Finding
Resolution K-Transformations

Diplomarbeit
Jan-Georg Smaus

Saarbrücken, Februar 1996

Author's Address:

Jan-Georg Smaus
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Germany
E-mail: smaus@mpi-sb.mpg.de

Advisor:

Priv. Doz. Dr. Hans Jürgen Ohlbach

Abstract

Resolution K-transformations are faithful transformations between clause sets. The aim is to remove clauses like symmetry or transitivity from a clause set in order to eliminate or reduce recursivity and circularity in this clause set. It is shown that for any set of such clauses, a resolution K-transformation is likely to exist and can be found automatically. Clause K-transformations may be applied to reduce the search space of theorem provers, to eliminate loops in logic programs, to parallelise closure computation algorithms and to support automated complexity analysis.

Declaration

Ich erkläre, die vorliegende Arbeit selbständig im Sinne der Diplomprüfungsordnung erstellt und ausschließlich die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Saarbrücken, den 06.02.1996

Jan-Georg Smaus

Acknowledgements

Mein größter Dank gilt meinem Betreuer und Erstkorrektor Priv. Doz. Dr. Hans Jürgen Ohlbach, der sich stets Zeit für meine Fragen und Probleme nahm und mir durch seine kritischen Kommentare und Verbesserungsvorschläge sehr geholfen hat.

Herr Prof. Dr. Jörg Siekmann hat sich freundlicherweise bereiterklärt, die Zweitkorrektur der Arbeit zu übernehmen.

Ich danke den Mitarbeiterinnen und Mitarbeitern des Max-Planck-Instituts für die angenehme Atmosphäre, in der ich hier arbeiten durfte, und für die große Hilfsbereitschaft bei fachlichen Problemen. Besonders erwähnen möchte ich Peter Barth, Peter Graf, Hubert Baumeister und Luca Viganò.

Ich danke Peter Leven, Alexander Bach, Christoph Meyer, Abdelwaheb Ayari und allen anderen Kommilitonen am Max-Planck-Institut für die Zusammenarbeit und die schöne gemeinsam verbrachte Zeit.

Vor allem danke ich meinen Eltern für ihre fortwährende Unterstützung.

Contents

Chapter 1

Introduction

When we develop and investigate logical systems, there are two closely related aspects. One aspect is to make statements *about* the system. The other aspect is to understand a new logical system in terms of another familiar system.

Thus we have two logical systems $\mathcal{L}_1$ and $\mathcal{L}_2$ and a transformation $\Upsilon : \mathcal{L}_1 \to \mathcal{L}_2$. We would like to be able to solve a problem in $\mathcal{L}_1$ by transforming it to $\mathcal{L}_2$, finding the solution there, and then transforming this solution back to $\mathcal{L}_1$.

We can divide this requirement into two parts: On the one hand, there is *soundness*. If we use Υ to transform a problem in $\mathcal{L}_1$ into a problem in $\mathcal{L}_2$, find the solution there and then transform the solution back to $\mathcal{L}_1$, it must be guaranteed that this is indeed a solution for the original problem.

On the other hand, there is completeness. Completeness means that for every problem in $\mathcal{L}_1$, we can find a solution by transforming the problem to $\mathcal{L}_2$ and finding the solution there.

Let us be a bit more specific and assume that the problems we want to solve are theorem proving problems. Then the above requirements translate into:

$$\begin{array}{rcl} \mathcal{L}_1 : Assumption & \Rightarrow & Conclusion \\ & \text{iff} & \\ \mathcal{L}_2 : \Upsilon(Assumption) & \Rightarrow & \Upsilon(Conclusion) \end{array} \tag{1}$$

Why should we want to transform a problem in a logical system $\mathcal{L}_1$ into a problem in $\mathcal{L}_2$? Well, the transformation should simplify the solution of the problem in some sense. A simplification would be that properties that have to be formulated explicitly in $\mathcal{L}_1$ hold "automatically" in $\mathcal{L}_2$. Thus our aim is to eliminate properties ϕ of $\mathcal{L}_1$ where $\Upsilon(\phi)$ is a tautology or in some other sense redundant in $\mathcal{L}_2$.

Various transformations of this kind have been introduced in [**?**]. We want to focus here on transformations of clause sets into clause sets. Thus the logical systems $\mathcal{L}_1$ and $\mathcal{L}_2$ are not really different.

The aim of this transformation is to get rid of *self-resolving* clauses. A self-resolving clause is one that can be resolved with a variable renamed copy of itself. If such a clause is part of a clause set that is to be refuted by an automated theorem prover, or if it is part of a logic program, it is usually a major source of problems. In both cases the clause may be resolved with itself over and over again, with no end.

The basic idea is to turn this infinite sequence of resolutions into a finite one. We would like to add a finite number of resolvents of the self-resolving clause, and remove it afterwards. The newly added clauses should not be self-resolving.

Of course the clause set should not lose any of its “meaning” by this transformation. So actually we must find criteria which resolvents of a self-resolving clause are needed, and which are not.

We shall see how such a transformation works in general. Of course the transformation depends on the clause we attempt to eliminate from the clause set. We shall speak of a transformation *for* a clause C if C is the clause we want to eliminate. The main focus of this paper is how a transformation for a clause can be found automatically.

1.1 Applications

There are at least four areas where interesting results can be obtained through the study of self-resolving clauses:

- Automated theorem proving. The search behaviour of a resolution-based theorem prover can be modified by transforming the clause set that is to be refuted.
- Parallelising of closure computation algorithms.
- Eliminating loops in logic programs. This may be done by replacing a self-resolving program clause with non-self-resolving clauses.
- Automated complexity analysis.

We shall not investigate here how the elimination of self-resolving clauses can be exploited for automated complexity analysis. This is explained in [**?**] and [**?**].

For the other three areas, I shall now give some examples.

1.1.1 Transitivity

The transitivity clause is a simple self resolving clause, and it will accompany us throughout this paper. It is defined as

$$C := \neg P(x, y) \vee \neg P(y, z) \vee P(x, z).$$

Transitivity triggers infinitely many resolutions. Consider the clause $P(a, b)$, for instance. Resolving on the first literal of C, we can generate the following infinite sequence of resolvents:

$$\begin{array}{l} \neg P(b, z) \vee P(a, z). \\ \neg P(b, z) \vee P(z, z') \vee P(a, z'). \\ \neg P(b, z) \vee P(z, z') \vee P(z', z'') \vee P(a, z'). \\ \ldots \end{array}$$

Brand has shown for the transitivity of the equality predicate that only the first of these resolvents is needed [**?**]. Actually this has nothing to do with a particular property of equality, but is true for any transitive relation. For any clause of the form $P(s, t)$, where P is a transitive relation, it is sufficient to add the resolvent between $P(s, t)$ and C on the first literal[1].

Let us look at a small example. Suppose we have a clause set that contains the transitivity clause and

[1] Alternatively, we might have chosen the second literal of C. However, we must do this consistently. Either we always take the first literal, or always the second.

$$\begin{array}{llll} C_1: & P(a,b) & \qquad C_2: & P(b,c) \\ C_3: & P(c,d) & \qquad C_4: & \neg P(a,d) \end{array}$$

This clause set is unsatisfiable. Now for each clause we add a single resolvent with the first literal of C. We get

$$\begin{array}{l} R_1: \neg P(b,z) \vee P(a,z) \\ R_2: \neg P(c,z) \vee P(b,z) \\ R_3: \neg P(d,z) \vee P(c,z) \end{array}$$

The clauses C_3, C_4, R_1, and R_2 are sufficient to derive the empty clause. The transitivity clause is not needed anymore.

This is not a coincidence. For the transitivity clause, it is sufficient to add resolvents as shown in the example. Afterwards, the transitivity clause can be removed.

We can say that the newly added clauses do not express transitivity in general, but rather express transitivity for this particular basic relation.

We shall look at this clause set more closely in Example **??**.

For other clauses than transitivity, this is generally not so simple. Nevertheless, for many self-resolving clauses it is possible to add a limited number of resolvents and remove the clause afterwards.

We shall see how such a transformation can be characterised in general, and how we can find a transformation automatically.

Note that all the original clauses except for the one expressing transitivity must remain in the clause set. Intuitively, this can be explained as follows: Each clause in a clause set expresses a certain property of a relation. By removing transitivity, we make the clause set *weaker*. Something that was expressed before is not expressed anymore. To make up for this loss of meaning, all the other clauses must *gain* meaning. But for this the least thing we must expect is that the original meaning of a clause D is not lost. That is, $\Upsilon(D)$ must imply D, and since $\Upsilon(D)$ is a clause set, this means that $\Upsilon(D)$ must contain D.

1.1.2 A 'Process View' of the Transformation

A clause like the transitivity clause can serve as a nucleus for a hyperresolution step. Hyperresolution with the transitivity clause takes two other clauses, the electrons, as input partners and generates the hyperresolvent as output. To continue the previous example, the transitivity clause takes $R(a,b)$ and $R(b,c)$ as input and produces $R(a,c)$ as output.

If we apply hyperresolution repeatedly, we successively compute the transitive closure of a basic relation. Actually we can identify the transitivity clause with a process that computes the transitive closure of a relation. From this point of view, the transitivity clause is the active part, and the other clauses are the data.

When we transformed the clause set of the previous example, we added three clauses R_1, R_2, R_3. These clauses can also serve as nuclei for a hyperresolution step. For example, R_1 may take $P(b,c)$ as input and produce $P(a,c)$ as output. Taking R_1, R_2, and R_3 together, the transitive closure can be computed without using the original transitivity clause.

So we might say that we have replaced the "transitivity"-process by three processes R_1, R_2, and R_3. One aspect of this is that we have turned the passive data contained in C_1, C_2, and C_3, into active processes. These processes are more "specialised", since they contain information about the data part.

The other aspect is parallelism. The three processes can work in parallel, so we have parallelised the closure computation process for the basic relation given by C_1, C_2, and C_3.

1.1.3 Loops in Logic Programs

Consider the Prolog program

```
married(heinz, hilde).
married(X, Y) :- married(Y, X).
```

For successful queries, say `married(hilde,heinz)`, this program works fine. But the query `married(heinz,gerda)` will cause the Prolog system to run into an infinite loop, since the second clause can be used (which means, resolved upon) over and over again.

The solution to this problem is exactly the same as for the transitivity example. The self-resolving clause '`married(X,Y) :- married(Y,X).`' is removed, and resolvents with all the fact clauses (in this case there is only one) are added instead.

For the little program shown above we have

```
married(heinz, hilde).
married(hilde, heinz).
```

The transformed program can not run into an infinite loop anymore.

1.2 Finding Transformations

So far we have not given any proof that the transformations for transitivity and symmetry shown above are sound and complete. We have not even formulated precisely how a transformation is defined, and what we mean by soundness and completeness. Nevertheless, the transformed Prolog program and the example of the transitive closure computation lead us to believe that the transformations are complete[2]. Why is that so? Essentially it is because we reason about the *semantics* of a clause. We are so familiar with a property like transitivity that we can easily tell that the newly added clauses are indeed sufficient to compute the transitive closure of a relation.

Of course this intuition is not enough. We shall have to turn our concept of transformation into a precise definition, and we shall have to prove that a transformation for a clause is sound and complete. More generally, we shall have to find criteria so we can tell whether a function that *might* be a sound and complete transformation for a clause really *is* a sound and complete transformation for that clause. All this is shown in [?]. We shall repeat the results, but not give the proofs.

But then how do we find a function that *might* be a transformation for a clause, a candidate, so to speak? Of course, this candidate should not be just any function, but one that has a good chance to meet our criteria.

Here again, semantic considerations and some good intuition may help. In [?] and [?], several transformations have been found this way. That is, a candidate is constructed using an informal argument, but then the criteria are tested formally.

From this point of view, finding a transformation and understanding why it works is essentially the same thing. Taking transitivity, for example, the process of *finding* a transformation for this clause is guided by the idea that after the transformation has been done, it should still be possible to compute the transitive closure.

[2]Soundness is trivial in these examples.

If we want to find a transformation for a clause *automatically*, this does no longer work. Unless we can describe our semantic considerations in an algorithmic way, we must abstract from the semantics of a particular clause. For an arbitrary clause we must enumerate candidates systematically, and then check whether a candidate is a transformation or not.

Finding transformations automatically is the main focus of this paper. We shall see that for clauses like the transitivity clause or the symmetry clause there is a *finite* set of candidates that is likely to contain a function which indeed is a sound and complete transformation for this clause. I have implemented this search, and we shall see several examples of transformations that have been found automatically.

For prominent clauses like the ones mentioned above, this is not so impressive because transformations for them have already been found by semantic considerations. However, we shall see the example of euclideanness where the transformation found automatically is simpler than the transformation found by a semantic consideration in [**?**].

It is also possible to eliminate several clauses at the same time. We shall treat this point, too, but throughout most of this paper, we will assume that we want to eliminate just one clause.

Chapter 2

Clause K-Transformations

2.1 Notions and Notation

We now introduce some basic notions and naming conventions.

If S is a finite set, we denote the cardinality of S by $\sharp S$ or $|S|$. If S is infinite, we write $\sharp S = \infty$.

The letters from the end of the alphabet usually denote *variables*. The letters from the beginning of the alphabet denote *constants*. f, g, h denote *functions*, s and t arbitrary *terms*. P and R denote *predicate symbols*.

If P is a n-ary predicate symbol and t_1,t_2,...,t_n are arbitrary terms, then $P(t_1, t_2, \ldots, t_n)$ is an *atom* .

If A is an atom, A and $\neg A$ are literals. A is a *positive* literal and $\neg A$ is a *negative* literal. Thus we do not say that an *atom* A is negative because it occurs in a formula with a $\neg$-sign. We are strict about this and say that $\neg A$ is a negative literal.

A *clause* is a disjunction of literals. If $\neg A_1, \neg A_2, \ldots \neg A_n$ are the negative literals and $B_1, B_2, \ldots B_m$ are the positive literals of a clause, there are three ways of writing this clause:

As disjunction:	$\neg A_1 \vee \neg A_2 \ldots \vee \neg A_n \vee B_1 \vee B_2 \ldots \vee B_m$
As implication:	$A_1 \wedge A_2 \ldots \wedge A_n \Rightarrow B_1 \vee B_2 \ldots \vee B_m$
As set:	$\{\neg A_1, \neg A_2, \ldots \neg A_n, B_1, B_2, \ldots B_m\}$

We assume that the variables of a clause are universally quantified. Thus a clause (as disjunction) is a closed formula. Therefore it is clear that two occurrences of the same variable in different clauses have nothing to do with each other.

However, when we come to resolution, it is better to enforce this condition in a technical way. This means that we imagine clauses to be available in infinitely many variable renamed copies. Each time a clause is "used" (we shall see later what "using clauses" means) for something, a new copy is taken.

A *unit clause* or *unary clause* is a clause that contains only one literal. We shall usually blur the distinction between a unary clause and the literal that is contained by this clause.

If A is an atom, we say that A is the *complement* of $\neg A$, and $\neg A$ is the complement of A. For a literal L we write $\bar{L}$ for its complement.

A *substitution* σ is an endomorphism on the free term algebra replacing a finite number of variables. We write substitutions as sets $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. A *variable renaming* is a substitution $\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$, where $y_i \neq y_j$ for $i \neq j$ and $y_i \neq x_j$ for all $i, j \in \{1, 2, \ldots n\}$.

We write $s\sigma$ instead of $\sigma(s)$ for the application of a substitution σ to a term s. $\sigma\tau$ denotes the composition of the substitutions σ and τ.

A substitution σ is called a *unifier* of two terms or atoms s and t if $s\sigma = t\sigma$. A substitution σ is called *more general* than τ if there is a substitution σ' such that $\tau = \sigma\sigma'$.

A *most general unifier* for two terms s and t is a unifier σ such that all other unifiers τ can be composed of σ and some other substitution. This means

$$\begin{array}{ll} s\sigma = t\sigma & \text{and} \\ s\tau = t\tau & \text{implies} \quad \tau = \sigma\sigma' \quad \text{for some } \sigma'. \end{array} \tag{2}$$

The most general unifier of two terms is unique up to variable renaming. We write $\mathrm{mgu}(s,t)$ for the most general unifier of s and t.

Two terms are *unifiable* if they have a unifier. We call two literals *complementary unifiable* if they have different signs and their atoms are unifiable.

A substitution μ is a *matcher* of s on t if $s\mu = t$.

Two terms s and t are called *variants* if they have no variables in common and there is a variable renaming η such that $s\eta = t$. We say that s and t are "equal up to variable renaming".

We say $s\sigma$ is an *instance* of s. If $s\sigma$ does not contain variables, we call $s\sigma$ a *ground instance* of s.

A clause C *subsumes* a clause D if there is a substitution μ such that $C\mu \subseteq D$. This means that C is an instance of a subclause of D. In this case C implies D. If $C\mu$ is a *proper* subset of D, we say that C subsumes D *properly*.

The standard inference rule for many theorem provers is *resolution*([**?**]). Let $C_1 = L_1 \vee L_2 \vee \ldots \vee L_n$ and $C_2 = K_1 \vee K_2 \vee \ldots \vee K_m$ be two clauses where L_1 and K_1 are complementary unifiable with mgu σ. Then the resolution rule is defined as follows:

$$\frac{\begin{array}{l} L_1 \vee L_2 \vee \ldots \vee L_n \\ K_1 \vee K_2 \vee \ldots \vee K_m \end{array}}{\sigma(L_2 \vee \ldots \vee L_n \vee K_2 \vee \ldots \vee K_m)} \tag{3}$$

This is to be understood in the sense that the clause below the line can be inferred from the two clauses above the line. Of course this rule reflects that the clause below the line is a logical consequence of the two clauses above the line.

L_1 and K_1 are the *resolution literals*. We say that we *resolve* C_1 and C_2 *on* the resolution literals. We write

$$\mathrm{Res}(C_1, C_2) = \sigma(L_2 \vee \ldots \vee L_n \vee K_2 \vee \ldots \vee K_m). \tag{4}$$

and call $\mathrm{Res}(C_1, C_2)$ a resolvent of C_1 and C_2.

In order for this notation to be unambiguous we must say that resolution is done on the *first* literal of C_1, C_2, respectively. Often we will not worry about this because we make statements that hold for *every* resolvent of two clauses!

We will sometimes say "C_1 is the *resolution partner* of C_2" (and vice versa).

2.2 Self Resolution

Self resolution means resolution between two variable renamed copies of a clause. Let us consider the transitivity clause, for example. The transitivity clause is defined as $\neg P(x,y) \vee \neg P(y,z) \vee P(x,z)$. Self resolution on this clause yields

$$
\begin{array}{ccccccccc}
\neg P(x,y) & \vee & \neg P(y,z) & \vee & P(x,z) & & & & \\
 & & & & | & & & & \\
 & & & & \neg P(x',y') & \vee & \neg P(y',z') & \vee & P(x',z') \\
\hline
\neg P(x,y) & \vee & \neg P(y,z) & \vee & & & \neg P(z,z') & \vee & P(x,z')
\end{array}
\tag{5}
$$

The resolvent obtained in a self resolution step is called ***self resolvent***. Coincidentally the transitivity clause has only one self resolvent, up to literal ordering.

The process of self resolution can be repeated. We give a formal definition.

Definition 2.2.1 The definition is an inductive one. Let C be a clause. As usual assume that clauses are variable renamed before resolution is performed.

1. C is a self resolvent of C.
2. If R and Q are self resolvents of C, then $\mathrm{Res}(R,Q)$ is a self resolvent of C.

$\diamond$

For certain clauses, finding a transformation requires computing self resolvents. If we want to find a transformation automatically, we must generate the self resolvents automatically, of course.

The definition says that resolution between arbitrary self resolvents results in another self resolvent. However it is not necessary to resolve arbitrary self resolvents with each other. All self resolvents can be obtained in resolution steps where one of the resolution partners is the original clause. This facilitates the computation.

To express this point precisely we need a definition.

Definition 2.2.2 Let C be a clause. We define a function $|.|$ on the self resolvents of C that we call "level":

1. $|C| = 0$.
2. $|\mathrm{Res}(R,Q)| = |R| + |Q| + 1$.

$\diamond$

The following corollary will show that it is sufficient to consider only resolvents where one resolution partner is the original clause.

Corollary 2.2.3 Let C be a clause and R, Q be self resolvents of C. Then there is a self resolvent P such that

$$\mathrm{Res}(R,Q) = \mathrm{Res}(P,C) \tag{6}$$

Proof:
The proof uses induction on $|\mathrm{Res}(R,Q)|$.

1. $|\mathrm{Res}(R,Q)| = 1$.
   This case is trivial; just take $P = R$.
2. $|\mathrm{Res}(R,Q)| = m > 1$.
   $|Q|$ is smaller than m, therefore by induction hypothesis Q can be obtained by resolution between some self resolvent Q' and C, that is $Q = \mathrm{Res}(Q',C)$.

   We write $Q' = Q_1 \vee Q_2 \vee Q_{\mathrm{rest}}$, $C = C_1 \vee C_{\mathrm{rest}}$ and $R = R_1 \vee R_{\mathrm{rest}}$. This means, we split the literals upon which we resolve from the rest of the respective clauses. This can easily be seen in Fig.**??**, which illustrates the last

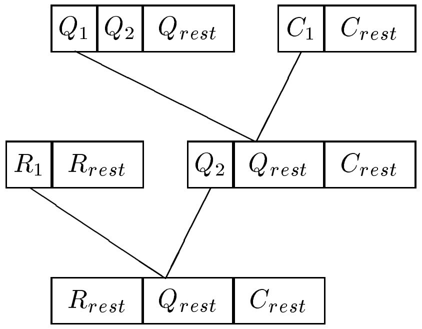

Figure 2.1: Original order

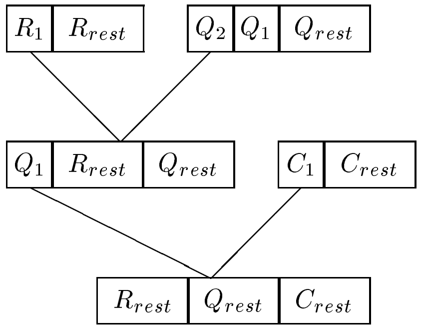

Figure 2.2: Interchanged order

two resolution steps that are performed to get $\mathrm{Res}(R, Q)$. Each node in this tree represents a clause. The arcs indicate which clauses are resolved with each other. We want to show that we can first resolve between R and Q' before resolving with C, that is, interchange the order of the last two resolutions.

Essentially we must argue that unifiability is not affected when the order of the resolutions is interchanged. In order to ensure that every resolution is done with a fresh copy of a clause, let us assume that the function mgu introduces fresh variables.

Let us define $\sigma := \mathrm{mgu}(Q_1, C_1)$ and $\tau := \mathrm{mgu}(R_1, Q_2\,\sigma)$. R_1 shares no variables with Q_1 or C_1. Therefore $R_1\,\sigma = R_1$ and $R_1\,\sigma\,\tau = Q_2\,\sigma\,\tau$. Thus $\sigma\,\tau$ is a unifier of R_1 and Q_2, but it cannot be more general than $\mathrm{mgu}(R_1, Q_2)$.

Furthermore $Q_1\,\sigma\,\tau = C_1\,\sigma\,\tau$. But then $Q_1\,\mathrm{mgu}(R_1, Q_2)$ and $C_1\,\mathrm{mgu}(R_1, Q_2)$ must be unifiable.

So we can first resolve between R and Q'[1] The substitution done in Q_1 will not affect unifiability with C_1. $\mathrm{Res}(R, Q')$ is the clause P we are looking for. In the following equation, substitutions are neglected for readability. Fig.**??** illustrates the resolution process when the last two resolutions are interchanged.

$$
\begin{aligned}
\mathrm{Res}(R, Q) &= \mathrm{Res}(R, \mathrm{Res}(Q', C)) \\
&= \mathrm{Res}(R_1 \vee R_{\mathrm{rest}}, \mathrm{Res}(Q_1 \vee Q_2 \vee Q_{\mathrm{rest}}, C_1 \vee C_{\mathrm{rest}})) \\
&= \mathrm{Res}(R_1 \vee R_{\mathrm{rest}}, Q_2 \vee Q_{\mathrm{rest}} \vee C_{\mathrm{rest}})) \\
&= R_{\mathrm{rest}} \vee Q_{\mathrm{rest}} \vee C_{\mathrm{rest}} \\
&= \mathrm{Res}(R_{\mathrm{rest}} \vee Q_1 \vee Q_{\mathrm{rest}}, C_1 \vee C_{\mathrm{rest}}) \\
&= \mathrm{Res}(\mathrm{Res}(R_1 \vee R_{\mathrm{rest}}, Q_2 \vee Q_1 \vee Q_{\mathrm{rest}}), C_1 \vee C_{\mathrm{rest}}) \\
&= \mathrm{Res}(\mathrm{Res}(R, Q'), C) \\
&= \mathrm{Res}(P, C) \qquad (7)
\end{aligned}
$$

So we have shown that $\mathrm{Res}(R, Q)$ can be obtained by resolution between an appropriate self resolvent of C and C itself. In other words, *Res* is associative.

□

The previous corollary says: If we want to compute the self resolvents of level n, it is sufficient to take all self resolvents of level $n-1$ and resolve them with the original clause.

[1] Q' on the second literal, however!

2.3 General Clause K-Transformations

We shall now define what a clause K-transformation is and see how it works. We start with an abstract definition. This definition captures the essential properties of clause K-transformations without describing how a clause K-transformation could be constructed.

However we have a much more concrete class of transformations in mind. So in Section **??** we shall look extensively at an example that will show how a clause transformation works. This example is for illustration, and we cannot be very formal at this point.

The following section takes us back on the formal trail. We shall become specific about the abstract definition given in the first section. Namely, we shall introduce the *resolution K-transformations*.

A clause K-transformation has to meet certain conditions. Making our definition specific means that we shall define a class of transformations that meets most of these conditions by construction.

We want to remove a clause C from a clause set Φ. The other clauses in Φ must be transformed such that satisfiability of Φ is preserved. The definition of *resolution K-transformations* says that an arbitrary clause D is transformed by generating resolvents between D and clauses of a clause set S_Υ. This set S_Υ depends on C, of course. S_Υ specifies the transformation Υ in a constructive way.

However, S_Υ has to meet one non-trivial condition. This is the so-called *test substitution set condition*. The verification of this condition involves theorem proving and thus is indeed non-trivial. Thus *finding* a clause set S_Υ that meets this condition is the difficult part.

Working with resolution K-transformations we shall be able to concentrate on this non-trivial condition. The other conditions hold automatically.

It is not known to me that there is a class of clause K-transformations that is really different from resolution K-transformations. Thus from our point of view the difference between the general definition of clause K-transformations and the definition of resolution K-transformations is a difference concerning the level of abstraction.

It is conceivable, however, that one could find a class of clause K-transformations that is not identical with the resolution K-transformations.

So we have a clause set Φ containing a certain clause C we would like to eliminate. To this end we must transform the other clauses of the clause set. The transformation Υ should be sound and complete. This means

$$\Phi \text{ is satisfiable} \qquad \textbf{iff} \qquad \Upsilon(\Phi \setminus C) \text{ is satisfiable} \tag{8}$$

At this point recall the following intuition: Removing C makes the clause set *weaker*. On the other hand, each transformed clause must be stronger than the original clause. When a clause D is transformed, D remains in the clause set, and possibly, further clauses are added. Thus $\Upsilon(D)$ *contains* D, which means that $\Upsilon(D)$ *implies* D.

The Eliminated Clause Must Become Redundant

An operational view will help us to get a first intuition for the properties a transformation must have.

Let us consider a very simple clause $A \Rightarrow B$ that is an element of some clause set Φ. A and B may contain variables, so that there may be many different instantiations of A and B. We can imagine this clause as an assertion, a simple statement of the fact that “A implies B”.

On the other hand, we can look at this clause as a procedure, or an operation. When the procedure finds an instance of A, it will generate an instance of B. This makes sense when we think of resolution based theorem proving. $A \Rightarrow B$ combined with the resolution technique forms a procedure. Whenever A occurs, it may be resolved with $A \Rightarrow B$ to give B as output. This is the original setting, without transformation involved.

Now let us look at the transformed clause set. In the transformed clause set, $A \Rightarrow B$ should no longer be needed. Neither should $\Upsilon(A \Rightarrow B)$ be needed, because we want to *eliminate* $A \Rightarrow B$ and not transform it. The first point is that we would still like to be able to derive B, provided that A is given. In the transformed clause set, we can make use of stronger preconditions than in the original set in order to derive B. To be precise, we can use $\Upsilon(A)$ instead of A. Thus it would be sensible to require that $\Upsilon(A) \Rightarrow B$ is a tautology. Then B could be derived and everything would be fine.

A closer look shows that this is not sufficient. If we said that in the transformed clause set, we can make use of stronger preconditions, then we must also make sure that these stronger preconditions are maintained for further inferences! Let us go back to the original setting: An instance of B, say B', is derived from an instance of A, say A'. B' however may be used for further inferences. In particular, B' itself could be an another instance of A, say A''. A'' is used by the clause $A \Rightarrow B$ to derive another instance of B, say B''.

All this must be simulated in the transformed setting. If all we have is that $\Upsilon(A) \Rightarrow B$ is a tautology, we can derive B'. But B' $(= A'')$ is not strong enough to derive B''. We need $\Upsilon(A) \Rightarrow \Upsilon(B)$ to be a tautology. Then it is possible to derive $\Upsilon(B')$ $(= \Upsilon(A''))$, and from this we can derive $\Upsilon(B'')$. This implies that we derived B'', because $\Upsilon(B'')$ contains B''.

This is illustrated by the following picture:

$$\text{Without Transformation: } \boxed{A'} \xrightarrow{\text{“}A \Rightarrow B\text{”}} \boxed{B' \ \ (= A'')} \xrightarrow{\text{“}A \Rightarrow B\text{”}} \boxed{B''}$$

$$\text{With Transformation: } \boxed{\Upsilon(A')} \xrightarrow{\text{taut}} \boxed{\Upsilon(B') \ \ (= \Upsilon(A''))} \xrightarrow{\text{taut}} \boxed{\Upsilon(B'')}$$

In [?] a definition of clause K-transformations is given capturing precisely the above condition.

The definition is more general than what we have considered until now. Rather than removing one clause C, we might also remove *several* clauses at the same time. That is, we remove a clause set $\mathcal{C}$. The following definition is only slightly more complicated than it would be if we restricted ourselves to removing only one clause.

Throughout most of this paper, we shall only consider clause K-transformations for *one* clause. In this case $\mathcal{C}$ is a singleton.

Definition 2.3.1 A function Υ mapping clauses to (possibly infinite) clause sets is called a *K-transformation for a clause set* $\mathcal{C}$ iff

1. $D \in \Upsilon(D)$ for all clauses D.

2. $\mathcal{C} \wedge D \Rightarrow \Upsilon(D)$ is a tautology for all clauses D.

3. $\Upsilon(D_1 \vee D_2) \Leftrightarrow \Upsilon(D_1) \vee \Upsilon(D_2)$ is a tautology for all *ground* clauses D_1 and D_2.

$$
\begin{tikzcd}
D \arrow[r, "\Upsilon"'] \arrow[d, "g"] & \Upsilon(D) \ni D_1 \arrow[dd, "g"'] \\
D_g \arrow[r, "\Upsilon"] & \Upsilon(D_g) \ni D_2 \arrow[d, "g"] \\
 & D'
\end{tikzcd}
$$

Figure 2.3: A lifting property for transformations

4. Let D be a clause and D_g be a ground instance of D. If D' is a ground instance of a clause in $\Upsilon(D_g)$, it is also a ground instance of a clause in $\Upsilon(D)$.

5. For all clauses $C = A_1 \wedge A_2 \ldots \wedge A_n \Rightarrow B_1 \vee B_2 \ldots \vee B_m$ in $\mathcal{C}$ and all ground instances $C\rho$ of C:

$$\vec{\forall}\Upsilon(A_1\rho) \wedge \ldots \wedge \vec{\forall}\Upsilon(A_n\rho) \Rightarrow \vec{\forall}\Upsilon(B_1\rho) \vee \ldots \vee \vec{\forall}\Upsilon(B_m\rho)$$

   is a tautology.

   $\vec{\forall}\Upsilon(\ldots)$ means the variables introduced by Υ are universally quantified.

For a clause set Φ let $\Upsilon(\Phi) := \cup_{D \in \Phi} \Upsilon(D)$.

$\diamond$

The above definition is not constructive. It states precisely the properties of clause K-transformations that are necessary to prove their soundness and completeness. Let us have a brief look at each of the conditions.

The first condition states that the original clause D is still necessary in the transformed clause set because it may be used in inferences with other clauses than the ones in $\mathcal{C}$.

Condition 2 guarantees soundness of the transformation. The transformed clauses must be implied by the original clauses.

Condition 3 relates transformations of single literals or parts of a clause with transformations of the whole clause. $\Upsilon(D_1) \vee \Upsilon(D_2)$ denotes the set of all disjunctions that can be formed taking one clause from $\Upsilon(D_1)$ and one clause from $\Upsilon(D_2)$, that is $\{E_1 \vee E_2 | E_i \in \Upsilon(D_i)\}$. The condition suggests that once we know how to transform literals, we know how to transform entire clauses. On the ground level this is indeed the case. A ground clause is transformed by forming the disjunction of all transformed literals. For non-ground clauses, we must be careful because transforming one literal may affect the instantiation of the variables in other literals.

Condition 4 states a certain homomorphism between instantiation and transformation, or in other words, a lifting property. It is illustrated in Fig. **??**. The horizontal arrows mean the transformation Υ, the vertical arrows stand for some ground substitutions. You can see in the graph that there are two paths leading from D to D'. The order of making a clause ground and transforming it may be interchanged in a certain sense.

Finally Condition 5 is the interesting one. It is the precise formulation of our intuitive argument that $\Upsilon(A) \Rightarrow \Upsilon(B)$ must be a tautology.

Theorem 2.3.2 [Soundness and Completeness of Υ] Let Φ is a clause set and $\mathcal{C} \subseteq \Phi$. Suppose Υ is a clause K-transformation for $\mathcal{C}$. Then Φ is satisfiable if and only if $\Upsilon(\Phi \setminus \mathcal{C})$ is satisfiable.

This theorem is proven in [**?**]. Soundness is easy. It follows directly from the second point of Def. **??**.

The completeness proof is very long and complicated. The basic idea is to transform refutation graphs. If the original clause set Φ is unsatisfiable, then the empty clause can be derived from Φ.

A refutation graph is a more abstract way to represent such a derivation. Each clause used in the derivation is represented by a node in the graph. Two literals are connected by an edge if they are resolved with each other in the derivation. Intuitively, if *each* literal is connected with another literal, this means that all literals are resolved away. Thus such a graph represents a derivation of the empty clause.

Now the completeness proof assumes that there is a refutation graph for the original clause set Φ. This graph is transformed by replacing each occurrence of a clause $C \in \mathcal{C}$ (and some of the neighbours of C) by an appropriate clause in $\Upsilon(\Phi \setminus \mathcal{C})$.

The new graph represents a derivation of the empty clause for the transformed clause set $\Upsilon(\Phi \setminus \mathcal{C})$.

2.4 An Example

Unfortunately we are not ready yet to introduce the transformations that we will eventually use, namely, the resolution K-transformations. We still need some formalism to give a precise definition. In order not to lose track of where we are heading, we shall give an example. Some of the concepts remain vague at this point, however.

If C is the clause to be removed, a clause D is transformed by adding resolvents between C and D. We will see that this can be understood as adding clauses that resemble C, but are more specific. Rather than having the general version of C, we have clauses that express everything about C that may ever be relevant for D. So much for intuition.

C itself may not be sufficient, however. In general, a transformation Υ is characterised by a *set* of clauses. We will call this set S_Υ. In order to transform a clause D, we have to add resolvents between D and all the clauses in S_Υ. This set may contain C itself as well as self resolvents of C and clauses that are subsumed by C. Finding such a set S_Υ is non-trivial in general, but we will start with an example where it is trivial. Namely, S_Υ contains C only.

The conditions 1-4 of Def. **??** are fulfilled by such a transformation. The difficult part is to find a set S_Υ such that Condition 5 is fulfilled.

Example 2.4.1 Consider $C = R(x,y) \wedge R(y,z) \Rightarrow R(x,z)$, i.e. the transitivity clause for the binary relation R. In this case the transformation is characterised by C itself, i.e. $S_\Upsilon = \{C\}$. We choose one *negative* literal of C that we call the *selected literal*. Transformation is done by resolution on this literal. Let us simply choose the first literal of C.

If we transform a unary clause $R(s,t)$, we get this clause and the resolvent between this clause and C.

$$\Upsilon(R(s,t)) = \{R(s,t),\ R(t,z) \Rightarrow R(s,z)\}$$

You see that for unit clauses, this transformation is very simple. Recall that z is implicitly universally quantified.

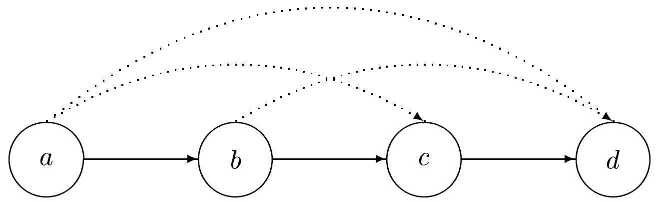

Figure 2.4: A simple basic relation and its transitive closure

Let us apply Υ to a concrete clause set Ψ consisting of unit clauses in order to get a feeling of how the transformation works and why it is sound and complete, and, last not least, why it may have a positive effect on the efficiency of theorem proving. Let

$$\Psi = \{R(a,b), R(b,c), R(c,d)\}.$$

Have a look at Fig.**??**. The solid lines depict the basic relation. The dotted lines depict the transitive closure, i.e. the unit clauses that can be inferred by resolution with the transitivity clause C.

Now we transform Ψ.

$$\begin{aligned} \Upsilon(\Psi) \quad = \quad & \{R(a,b), \forall z.R(b,z) \Rightarrow R(a,z) \\ & R(b,c), \forall z.R(c,z) \Rightarrow R(b,z) \\ & R(c,d), \forall z.R(d,z) \Rightarrow R(c,z)\} \end{aligned}$$

It can easily be seen that from $\Upsilon(\Psi)$, we can prove the ground unary clauses $R(a,c)$, $R(b,d)$, and $R(a,d)$, which is no more and no less than what can be proven from the original set $\Psi \cup \{C\}$.

Recall that the newly added clauses can be regarded as *processes*. For example $R(b,z) \Rightarrow R(a,z)$ generates $R(a,c)$ when it is given the input $R(b,c)$.

Replacing the "process" transitivity by three little "processes" has two main aspects.

One aspect is parallelism. If we imagine that in automated theorem proving one clause may be not used in several inferences at the same time, then our transformed clause set has an advantage compared to the original set. *Each* newly added clause can be involved in one inference, so we can conduct several inferences at the same time. As far as my work is concerned, the aspect of parallelism remains on the conceptual level. I propose a technique to transform clause sets in order to preprocess them for automated theorem proving. The theorem prover itself is not our concern here, but if a theorem prover indeed works in the described way, transforming may have a positive effect on efficiency.

The other aspect is removing redundancies. Using resolution as an inference rule, we can derive new clauses from a clause set as follows: For a clause set Φ, all possible resolutions between two clauses in Φ are performed. The new clauses are added to Φ. Then this process is repeated. When no more resolutions are possible, we are finished[2].

Suppose we want to compute the closure of $\Psi \cup \{C\}$ without using a resolution K-transformation. We write Ψ_i for the clause set in the i-th iteration.

[2]In general we may not expect termination, since predicate logic is not decidable.

$$
\begin{aligned}
\Psi_0 &= \{R(a,b),\ R(b,c),\ R(c,d),\ R(x,y) \wedge R(y,z) \Rightarrow R(x,z)\} \\
\Psi_1 &= \Psi_0 \cup \\
&\quad \{R(b,z) \Rightarrow R(a,z),\ R(x,a) \Rightarrow R(x,b), \\
&\quad\ R(c,z) \Rightarrow R(b,z),\ R(x,b) \Rightarrow R(x,c), \\
&\quad\ R(d,z) \Rightarrow R(c,z),\ R(x,c) \Rightarrow R(x,d)\} \\
\Psi_2 &= \Psi_1 \cup \\
&\quad \{R(a,c), R(b,d), \\
&\quad\ R(b,a) \Rightarrow R(a,b),\ R(b,b) \Rightarrow R(a,c),\ R(c,a) \Rightarrow R(b,b), \ldots\} \\
\Psi_3 &= \Psi_2 \cup \{R(a,d), \ldots\} \\
\Psi_4 &= \ldots
\end{aligned}
$$

The clauses in Ψ_1 can be resolved in 48 ways! Often two resolutions will have the same result. Furthermore, many of the clauses will never be used again. Take $R(b,b) \Rightarrow R(a,c)$, for example. Looking at Fig.**??** we can easily tell that we will never infer $R(b,b)$.

So Ψ_2 contains a lot of useless clauses, but it does not contain $R(a,d)$. We need a further resolution step to obtain the complete transitive closure. And even now, how do we know that we are finished?

Now in contrast suppose we had transformed Ψ first. Let us write Ψ' for $\Upsilon(\Psi)$.

$$
\begin{aligned}
\Psi'_0 &= \{R(a,b),\ R(b,c),\ R(c,d), \\
&\quad\ R(b,z) \Rightarrow R(a,z),\ R(c,z) \Rightarrow R(b,z),\ R(d,z) \Rightarrow R(c,z)\} \\
\Psi'_1 &= \Psi'_0 \cup \{R(a,c),\ R(b,d),\ R(c,z) \Rightarrow R(a,z),\ R(d,z) \Rightarrow R(b,z)\} \\
\Psi'_2 &= \Psi'_1 \cup \{R(a,d),\ R(d,z) \Rightarrow R(a,z)\}
\end{aligned}
$$

This is an exhaustive list of all the clauses that can be derived from Ψ' by resolution. Much fewer clauses are generated in order to compute the transitive closure[3] than if we consider Ψ. From Ψ, infinitely many clauses can be derived, and looking at Ψ_3, it is hard to see that we have already computed the transitive closure and may stop. We could just as well continue generating useless clauses forever. From Ψ'_2, in contrast, no further clauses can be inferred.

The trick is to resolve the transitivity clause itself only on the first literal. There are no *direct* resolutions with the transitivity clause on the second literal. Resolution on the second literal of the transitivity clause is done *indirectly*, that is, in later resolution steps.

But it is not trivial that we may do this, i.e. that completeness is guaranteed. We will see clauses other than transitivity where it is not sufficient to resolve on the first literal of the clause itself.

It is Condition 5 of Def. **??** that has to be checked to make sure that such a transformation is complete. This is proven in [**?**]. We shall check the condition for the transitivity clause. Take $P(a,b) \wedge P(b,c) \Rightarrow P(a,c)$ as a ground instance of C.

$$\vec{\forall}\Upsilon(R(a,b)) \wedge \vec{\forall}\Upsilon(R(b,c)) \Rightarrow \vec{\forall}\Upsilon(R(a,c))$$

is

[3] However it would not be fair to say that Ψ' is better than Ψ because we need only two iterations instead of three to compute the transitive closure. After all, we had to do the transformation first, which can be counted as an iteration, too. The advantage lies in the breadth of the search rather than in the depth.

$$
\begin{aligned}
&(R(a,b) \wedge (R(b,z) \Rightarrow R(a,z)) \\
&\wedge R(b,c) \wedge (R(c,z) \Rightarrow R(b,z))) \\
\Rightarrow\ &R(a,c) \wedge (R(c,z) \Rightarrow R(a,z))
\end{aligned}
$$

which is in fact a tautology.

This was one ground instance, but the transformation works for other ground instances just as well. The reason is that when we unify any ground literal $R(s,t)$ with $R(x,y)$ (the selected literal), only x and y are instantiated. The structure of s and t is irrelevant for the transformation. Therefore Condition 5 holds for all ground instances.

$P(a,b) \wedge P(b,c) \Rightarrow P(a,c)$ is a *prototypical* ground instance of C. It has all the properties relevant for the transformation itself and for the verification of soundness and completeness.

In other examples, the set of prototypical instances is not so simple.

We have seen how a resolution K-transformation is applied to *unary* clauses. We have also seen that to verify that a resolution K-transformation is complete, it must be applied to (prototypical) unary clauses. To round off the example, let us see how the transformation is applied to a clause of more than one literal.

Condition 3 of Def. **??** suggests how this must be done. We have seen there that we have to be careful about the instantiation of variables when transforming non-ground clauses. In the case of transitivity, we do not have this problem because the variables of the transformed clause are not instantiated.

Take $R(s,t) \vee R(q,r)$, for example.

$$
\begin{aligned}
\Upsilon(R(s,t) \vee R(q,r)) &= \Upsilon(R(s,t)) \vee \Upsilon(R(q,r)) \\
&= \{R(s,t),\ R(t,z) \Rightarrow R(s,z)\} \vee \\
&\quad\ \{R(q,r),\ R(r,z') \Rightarrow R(q,z')\} \\
&= \{R(s,t) \vee R(q,r), \\
&\quad\ \ R(r,z') \Rightarrow R(q,z') \vee R(s,t), \\
&\quad\ \ R(t,z) \Rightarrow R(s,z) \vee R(q,r), \\
&\quad\ \ R(t,z) \wedge R(r,z') \Rightarrow R(s,z) \vee R(q,z')\}
\end{aligned}
$$

Note that we used a variable renamed copy of C to transform $R(q,r)$.

We now leave our example and continue introducing formal concepts that will eventually allow us to generalise what we have seen here for the transitivity clause.

$\triangleleft$

2.5 Resolution K-Transformations

We have defined *clause K-transformations* by postulating certain properties (Def. **??**). The problem with this definition is that it is not constructive. We shall not solve this problem in the most general way. This means, there is no algorithm (known to me) that will, given an arbitrary clause, construct a transformation that meets the conditions of Def. **??**. But we shall:

1. present a definition of transformations such that conditions 1-4 will be met by construction. It has been said in Example **??** that such a transformation will be characterised by a clause set S_Υ. A clause is transformed by resolving it with the clauses in S_Υ. Thus all conditions will become trivial except for

Condition 5. This condition must be verified individually for each S_Υ. If the transformation characterised by S_Υ does meet Condition 5, we may call it a *resolution K-transformation.*

2. replace the test of Condition 5 by a more feasible test. This means testing Condition 5 for a limited set of substitutions rather than for all ground substitutions. This set can be enumerated systematically, and if we look at clauses without function symbols, it is even finite[4].

3. investigate different ways to find *candidates* for S_Υ such that the transformations Υ are likely to meet Condition 5.

Let us start with point **??**. The definition of the limited set of substitutions we have in mind is not so simple in general, and we must understand this definition before we can understand how it is verified that a clause set S_Υ characterises a transformation for a clause C.

From now on we shall write Υ for the transformation characterised by S_Υ, even though we do not know exactly yet *how* Υ is constructed from S_Υ.

2.5.1 Test Substitutions

The set of test substitutions actually depends on S_Υ. Assume that for each clause $S \in S_\Upsilon$, there is a literal $\neg L_S \in S$ that we call the *selected literal.* To make clear syntactically which literal is the selected literal, we shall always assume that the selected literal is the *first* literal.

Note that the selected literals are negative. Later we shall do unification, and for this purpose we need positive literals. Therefore we take the complements of the selected literals, and we call these *characteristic literals.* Call the set of characteristic literals L_Υ. It should be clear that L_Υ is derived from S_Υ in a trivial way.

For a literal L and a non-empty set of literals $\mathcal{K}$ we define $\mathrm{mgu}(L, \mathcal{K})$ as *simultaneous unifier* of L and all literals in $\mathcal{K}$. This means $L\ \mathrm{mgu}(L, \mathcal{K}) = K\ \mathrm{mgu}(L, \mathcal{K})$ for all $K \in \mathcal{K}$. If $\mathcal{K}$ is empty we define $\mathrm{mgu}(L, \mathcal{K})$ as the identity substitution.

Definition 2.5.1 Let $C = A_1 \wedge \ldots \wedge A_n \Rightarrow A_{n+1} \vee \ldots \vee A_m$ be a clause and L_Υ be a set of positive literals.

Starting with $\Sigma_0 := \{\epsilon\}$, where ϵ is the identity substitution, we iteratively define a set of *test substitutions.* For $i = 1, \ldots, m$

$$\Sigma_i := \bigcup_{\sigma \in \Sigma_{i-1}} \sigma \circ \{\mathrm{mgu}(A_i\,\sigma, \mathcal{K}) \mid \mathcal{K} \subseteq L_\Upsilon\} \tag{9}$$

where for a substitution σ and a set of substitutions Θ, $\sigma \circ \Theta$ denotes the set $\{\sigma \circ \theta \mid \theta \in \Theta\}$.

Finally we define

$$\Sigma := \Sigma_m. \tag{10}$$

If $\sigma \in \Sigma$, we call $C\,\sigma$ a *test instance.*

$\diamond$

The definition of the set of test substitutions is a bit complicated, and the set itself can be very large. However we shall see that for the clauses we want to investigate, the test substitutions turn out to be very simple. Let us state the following points about test substitutions:

[4] This also holds for certain clauses containing function symbols. However, the clause must not generate new *terms* through resolution. We shall see in Section **??** what it means for a clause to generate new terms.

- C must not share variables with any literal in L_Υ! This does not contradict Example **??** where the transformation was characterised by the clause itself. Recall what was said on page **??**: Each time a clause is "used" for something, a new copy is taken. In the example we had two variable renamed copies of C: One is an element of S_Υ and as such *characterises* the transformation, the other one is a copy we instantiate in some way or the other in order to *test* the transformation with this instance. Of course this point must be carefully considered when it comes to implementation.

- Furthermore, the variables of the characteristic literals in L_Υ must not share variables among each other.

- If in the course of the iterative construction of a $\sigma \in \Sigma$, some A_i is unified with a literal in L_Υ and later some A_j is unified with the same literal in L_Υ, then a *fresh* copy of the literal in L_Υ must be taken each time.

- The case $\mathcal{K} = \emptyset$ is important. By this it is ensured that one of the test substitutions is the identity substitution. We will mostly investigate examples where this is the only relevant test substitution.

- When computing the most general unifier of two terms, it happens frequently that we are free to choose whether to replace x by y or vice versa. Depending on how we do this, we will often end up with several test substitutions that instantiate C to clauses that are equal up to variable renaming. There is no need to consider these substitutions separately. A good way to think about it is the following: If in doubt instantiate into L_Υ rather than into C. If the substitution you receive does not instantiate any variables in C, you might as well forget it.

- We do not have to worry about how a test substitution instantiates the variables in L_Υ, because the test substitution is not applied to the literals in L_Υ, it is applied to the clause C. Therefore we can restrict the scope of a test substitution to the variables of C.

To illustrate these points we shall first look at a simple example, our well-known transitivity clause. Afterwards we shall look at an example that reveals how complicated the definition is, but that has nothing to do with a transformation that is investigated in this work. The reason is that for the transformations we encountered, the test substitution set was very simple.

Example 2.5.2 Let
$C = R(x,y) \wedge R(y,z) \Rightarrow R(x,z)$ and
$L_\Upsilon = \{R(u,v)\}$
Then Σ is computed as follows:

$$
\begin{aligned}
\Sigma_1 &= \epsilon \circ \{\mathrm{mgu}(R(x,y),\emptyset),\ \mathrm{mgu}(R(x,y),\{R(u,v)\})\} \\
&= \{\epsilon\}. \\
\Sigma_2 &= \epsilon \circ \{\mathrm{mgu}(R(y,z),\emptyset),\ \mathrm{mgu}(R(y,z),\{R(u',v')\})\} \\
&= \{\epsilon\}. \\
\Sigma_3 &= \epsilon \circ \{\mathrm{mgu}(R(x,z),\emptyset),\ \mathrm{mgu}(R(x,z),\{R(u'',v'')\})\} \\
&= \{\epsilon\}. \\
\Sigma &= \{\epsilon\}.
\end{aligned}
$$

Have another look at the points listed above and see how they are reflected in the example.

$\triangleleft$

Now we turn to a more complicated example.

Example 2.5.3 Let
$C = R(x, y) \vee R(1, y) \vee R(y, z)$ and
$L_\Upsilon = \{R(2, u), R(v, 3)\}$.
Then we have

$$
\begin{aligned}
\Sigma_1 &= \epsilon \circ \\
&\ \big\{\mathrm{mgu}(R(x, y), \emptyset),\ mgu(R(x, y), \{R(2, u)\}), \\
&\ \ \mathrm{mgu}(R(x, y), \{R(v, 3)\}),\ \mathrm{mgu}(R(x, y), \{R(2, u), R(v, 3)\})\big\}. \\
&= \big\{\epsilon, \{x \mapsto 2\}, \{y \mapsto 3\}, \{x \mapsto 2, y \mapsto 3\}\big\}. \\
\Sigma_2 &= \epsilon \circ \\
&\ \big\{\mathrm{mgu}(R(1, y), \emptyset),\ mgu(R(1, y), \{R(2, u')\}), \\
&\ \ \mathrm{mgu}(R(1, y), \{R(v', 3)\}),\ \mathrm{mgu}(R(1, y), \{R(2, u'), R(v', 3)\})\big\} \bigcup \\
&\ \{x \mapsto 2\} \circ \\
&\ \big\{\mathrm{mgu}(R(1, y), \emptyset),\ mgu(R(1, y), \{R(2, u')\}), \\
&\ \ \mathrm{mgu}(R(1, y), \{R(v', 3)\}),\ \mathrm{mgu}(R(1, y), \{R(2, u'), R(v', 3)\})\big\} \bigcup \\
&\ \{y \mapsto 3\} \circ \\
&\ \big\{\mathrm{mgu}(R(1, y), \emptyset),\ mgu(R(1, y), \{R(2, u')\}), \\
&\ \ \mathrm{mgu}(R(1, y), \{R(v', 3)\}),\ \mathrm{mgu}(R(1, y), \{R(2, u'), R(v', 3)\})\big\} \bigcup \\
&\ \{x \mapsto 2, y \mapsto 3\} \circ \\
&\ \big\{\mathrm{mgu}(R(1, y), \emptyset),\ mgu(R(1, y), \{R(2, u')\}), \\
&\ \ \mathrm{mgu}(R(1, y), \{R(v', 3)\}),\ \mathrm{mgu}(R(1, y), \{R(2, u'), R(v', 3)\})\big\} \\
&= \big\{\epsilon, \{x \mapsto 2\}, \{y \mapsto 3\}, \{x \mapsto 2, y \mapsto 3\}\big\}.
\end{aligned}
$$

You see that many of the most general unifiers that have to be considered in the above sets do not even exist. The ones that do exist are already contained in Σ_1, so that no new substitutions are added in this step.

Note that in the computation of Σ_2, we have renamed u and v to u' and v'. However u' and v' are used several times; they are not renamed for each mgu occurring in the above computation. This is safe because the substitutions in Σ_2 do not interfere *with each other*. Each substitution in Σ_2 is applied to C alternatively, not in composition.

$$
\begin{aligned}
\Sigma_3 &= \epsilon \circ \\
&\ \big\{\mathrm{mgu}(R(y, z), \emptyset),\ mgu(R(y, z), \{R(2, u'')\}), \\
&\ \ \mathrm{mgu}(R(y, z), \{R(v'', 3)\}),\ \mathrm{mgu}(R(y, z), \{R(2, u''), R(v'', 3)\})\big\} \bigcup \\
&\ \{x \mapsto 2\} \circ \\
&\ \big\{\mathrm{mgu}(R(y, z), \emptyset),\ mgu(R(y, z), \{R(2, u'')\}), \\
&\ \ \mathrm{mgu}(R(y, z), \{R(v'', 3)\}),\ \mathrm{mgu}(R(y, z), \{R(2, u''), R(v'', 3)\})\big\} \bigcup \\
&\ \{y \mapsto 3\} \circ \\
&\ \big\{\mathrm{mgu}(R(y, z), \emptyset),\ mgu(R(y, z), \{R(2, u'')\}),
\end{aligned}
$$

$$
\begin{aligned}
& \quad \mathrm{mgu}(R(y,z),\{R(v'',3)\}),\ \mathrm{mgu}(R(y,z),\{R(2,u''),R(v'',3)\})\} \bigcup \\
& \{x \mapsto 2, y \mapsto 3\} \circ \\
& \{\mathrm{mgu}(R(y,z),\emptyset),\ mgu(R(y,z),\{R(2,u'')\}), \\
& \quad \mathrm{mgu}(R(y,z),\{R(v'',3)\}),\ \mathrm{mgu}(R(y,z),\{R(2,u''),R(v'',3)\})\} \\
= & \{\epsilon, \{y \mapsto 2\}, \{z \mapsto 3\}, \{y \mapsto 2, z \mapsto 3\}\} \bigcup \\
& \{\{x \mapsto 2\}, \{x \mapsto 2, y \mapsto 2\}, \{x \mapsto 2, z \mapsto 3\}, \{x \mapsto 2, y \mapsto 2, z \mapsto 3\}\} \bigcup \\
& \{\{y \mapsto 3\}, \{y \mapsto 3, z \mapsto 3\}\} \bigcup \\
& \{\{x \mapsto 2, y \mapsto 3\}, \{x \mapsto 2, y \mapsto 3, z \mapsto 3\}\} \\
= & \{\{y \mapsto 2\}, \{z \mapsto 3\}, \{y \mapsto 2, z \mapsto 3\}, \{x \mapsto 2\}, \{x \mapsto 2, y \mapsto 2\}, \\
& \quad \{x \mapsto 2, z \mapsto 3\}, \{x \mapsto 2, y \mapsto 2, z \mapsto 3\}, \{y \mapsto 3\}, \{y \mapsto 3, z \mapsto 3\}, \\
& \quad \{x \mapsto 2, y \mapsto 3\}, \{x \mapsto 2, y \mapsto 3, z \mapsto 3\}\}.
\end{aligned}
$$

$\lhd$

Single Test Substitutions

Now we have seen a complicated example for test substitutions that should make clear how the definition works in principle. Fortunately this is not the typical case for the clauses we want to investigate. Rather, the first example (**??**) is typical. Let us see precisely why the test substitutions are so simple in that example.

If all characteristic literals ($\in L_\Upsilon$) have the form $R(x_1, x_2, \ldots x_n)$, where the variables x_i are all different, then $\mathrm{mgu}(A_i, \mathcal{K})$ is trivial.

For one thing all clauses in L_Υ are simultaneously unifiable in this case. This means there is a substitution σ such that $L\,\sigma = L'\sigma$ for all $L, L' \in L_\Upsilon$.

For all $\mathcal{K} \in L_\Upsilon$, $\mathrm{mgu}(A_i, \mathcal{K})$ does not need to substitute for variables in A_i. But we said earlier that it is only relevant what the test substitutions do to the variables of C, and in this sense, all test substitutions are the identity substitution.

It must not be neglected that for a literal $R(x_1, x_2, \ldots x_n)$ in L_Υ, all the x_i must be different! Otherwise, unification might force two arguments of A_i to be equal, and such a unifier would no longer be the identity substitution.

So let us state:

Corollary 2.5.4 [Corollary 4.9 in [**?**]] If all characteristic literals ($\in L_\Upsilon$) have the form $R(x_1, x_2, \ldots x_n)$, where the x_i are all different variables, then the test substitution set Σ contains the identity substitution only.

2.5.2 Simultaneous Resolution

We have seen in Example **??** that transforming a clause is done by resolving this clause with clauses contained in a set S_Υ that characterises the transformation. We shall now generalise our concept of transformations.

Recall what the restricted setting was: S_Υ is a singleton, and we transform only unary clauses (literals). To generalise means to consider the transformation of clauses of arbitrary length, where S_Υ may contain several clauses.

We define *simultaneous resolution* between a clause D and a clause set S_Υ. Simultaneous resolution is very similar to *hyperresolution*.([**?**]).

Definition 2.5.5 Let S_Υ be a set of clauses where each clause has at least one negative literal. For each clause $S \in S_\Upsilon$, there is a literal $\neg L_S \in S$ that we call the *selected literal*. To make clear syntactically which literal is the selected literal, we

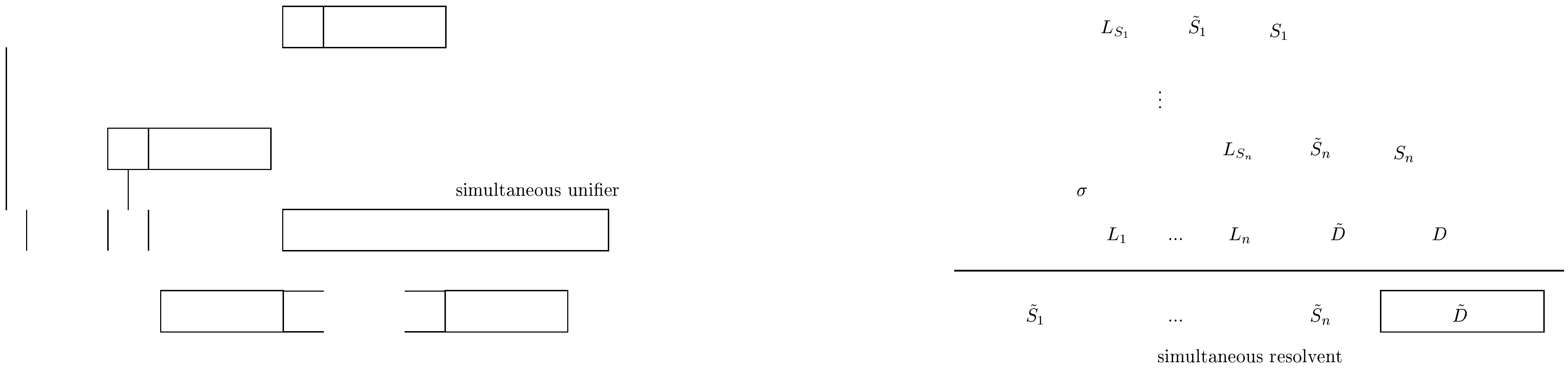

Figure 2.5: Simulteneous resolution between D and the clauses $S_1, \dots, S_n$

always assume that the selected literal is the *first* literal. Thus we can write S as $\neg L_S \vee \tilde{S}$.

Let D be a clause. Furthermore, let $\{L_1, L_2, \dots, L_n\} \subseteq D$ be a set of *positive* literals of D. Note that $L_i \neq L_j$ for $i \neq j$! Then we can write $D = L_1 \vee \dots \vee L_n \vee \tilde{D}$. Thus we separate the literals upon which we resolve from the rest clause $\tilde{D}$.

Now we pair each positive literal L_i with a clause $S_i \in S_\Upsilon$. Unlike the L_i, the S_i do not have to be all different[5].

If $(L_1, L_2, \dots, L_n)$ and $(L_{S_1}, L_{S_2}, \dots, L_{S_n})$ are unifiable with unifier σ, we define:

$$Res(D, L_1, \dots, L_n, S_1, \dots, S_n) \quad := \quad (\tilde{S_1} \vee \tilde{S_2} \vee \dots \vee \tilde{S_n} \vee \tilde{D})\sigma \tag{11}$$

If $(L_1, L_2, \dots, L_n)$ and $(L_{S_1}, L_{S_2}, \dots, L_{S_n})$ are not unifiable, $Res(D, L_1, \dots, L_n, S_1, \dots, S_n)$ is not defined.

We say: $Res(D, L_1, \dots, L_n, S_1, \dots, S_n)$ is the *simultaneous resolvent between D and the clauses $S_1, \dots, S_n$ on the resolution literals L_i and the selected literals L_{S_i}*. We call this operation *simultaneous resolution.*

This operation is illustrated by Fig. **??**.

unit=

◇

2.5.3 Definition of Υ

A resolution K-transformation is characterised by a clause set S_Υ as it has already occurred many times. For each clause, the selected literal must be fixed, which we point out by writing the selected literal first. This means, if S_Υ and $S_\Upsilon{}'$ are equal except for the order of literals in one clause, then S_Υ and $S_\Upsilon{}'$ represent different transformations!

Given S_Υ, we take the set of all simultaneous resolvents between D and clauses in S_Υ. This set is the result of the transformation. Formally:

Definition 2.5.6 Let $C := A_1 \wedge \dots \wedge A_n \Rightarrow A_{n+1} \vee \dots \vee A_m$ be a clause. Let S_Υ be a set of clauses and L_S defined as in Def. **??**. Let Σ be the set of test substitutions for S_Υ and C as defined by Def. **??**. For a clause D we define

[5]However we assume, as usual, that each copy of a clause in S_Υ is variable renamed.

[linestyle=dotted](140,645)(220,655)[fillstyle=solid,fillcolor=yellow](220,620)(300,640) [fillstyle=solid,fillcolor=yellow](280,580)(360,600)

$$\Upsilon(D) = \{Res(D, L_1, \ldots, L_n, S_1, \ldots, S_n) \mid n \geq 0, S_i \in S_\Upsilon, L_i \in D\} \tag{12}$$

It follows from the definition of simultaneous resolution that $L_i \neq L_j$ for $i \neq j$.

Once again, note that n can be 0, thus ensuring that $\Upsilon(D)$ contains D.

Υ is called a *resolution K-transformation for C* if it meets the following condition:

$$\vec{\forall}\Upsilon(A_1\,\sigma_{gr}) \wedge \ldots \wedge \vec{\forall}\Upsilon(A_n\,\sigma_{gr}) \Rightarrow \vec{\forall}\Upsilon(A_{n+1}\,\sigma_{gr}) \vee \ldots \vee \vec{\forall}\Upsilon(A_m\,\sigma_{gr}) \tag{13}$$

is a tautology for all $\sigma \in \Sigma$, where σ_{gr} is a grounded version of σ: all variables x in the codomain of σ are mapped to unique constants a_x.

This condition is called the *test substitution set condition.*

$\diamond$

We say "S_Υ characterises Υ".

Sometimes we speak of a "transformer" rather than a resolution K-transformation.

$\Upsilon(D)$ is a clause set, but we shall often interpret this clause set as a formula. This means that we regard the clauses in the set as joined by conjunction.

Let us illustrate how simultaneous resolution, and thus resolution K-transformations, work by looking at an example.

Example 2.5.7 Consider the following clause:

$$C := R(x,y) \wedge R(x,z) \Rightarrow R(z,y). \tag{14}$$

This clause formulates a property called "euclideanness". This clause cannot be eliminated in the same way the transitivity clause is eliminated. That means, it will not suffice to have S_Υ contain C itself. [**?**] proposes a transformation for this clause that is characterised by the following clause set:

$$\begin{aligned} S_\Upsilon \quad := \quad & \{R(x',y') \wedge R(x',z') \Rightarrow R(z',y'), \\ & R(x'',y'') \wedge R(y'',z'') \wedge R(y'',v'') \Rightarrow R(v'',z'')\} \end{aligned} \tag{15}$$

For each clause, the *first* literal is the *selected* literal as usual.

Let us transform $D := R(s,t) \vee R(q,r)$ with the transformer characterised by S_Υ. Fig. **??** shows some of the possible simultaneous resolution steps.

It has been said that one of the clauses generated when a clause D is transformed is D itself. This case is shown by the right picture of the second row of Fig. **??**. This is the marginal case, namely, that no resolution is done at all. This case must by no means be neglected.

How many simultaneous resolvents are there all together? Each positive literal of $R(s,t) \vee R(q,r)$ can be resolved with either literal of S_Υ, or not be resolved at all. Thus there are three possibilities per positive literal, resulting in $3^2 = 9$ different resolvents. Five of these are shown in Fig. **??**. The remaining four combinations should be easy to construct.

Each resolvent of Fig. **??** is a simultaneous resolvent, and the set of *all* simultaneous resolvents is the result of the transformation.

$\triangleleft$

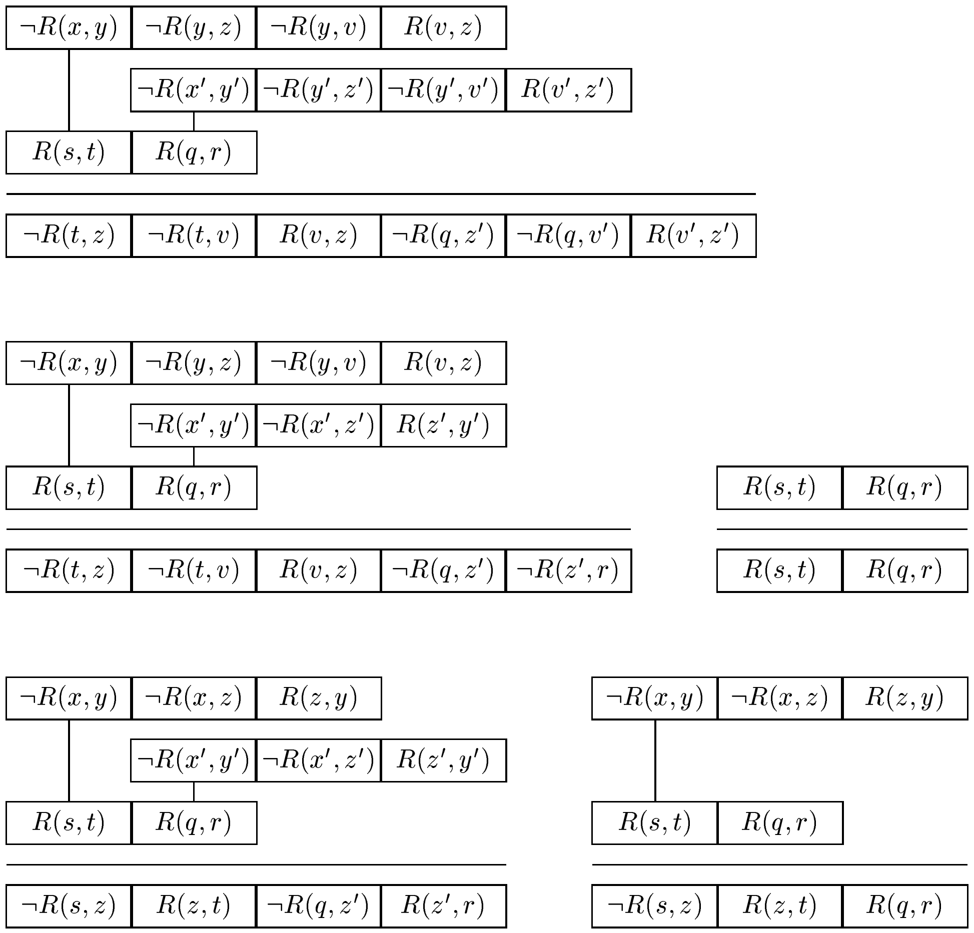

Figure 2.6: Some possible resolutions for $R(s,t) \vee R(q,r)$

2.5.4 Soundness and Completeness of Υ

We have now seen a constructive definition of transformations, characterised by a clause set S_Υ. The only condition that will have to be tested individually is the test substitution set condition (**??**).

Theorem 2.5.8 [Theorem 3.2 in [**?**]] Let Φ is a clause set and $C \in \Phi$. Suppose Υ is a resolution K-transformation for C. Then Φ is satisfiable if and only if $\Upsilon(\Phi \setminus \{C\})$ is satisfiable[6].

Thus if a resolution K-transformation satisfies condition (**??**), it is faithful.

2.6 Eliminating Clause *Sets*

We can generalise what we have seen above. Rather than eliminating just one clause C from a clause set Φ, we can eliminate a subset $\mathcal{C}$ of Φ.

Of course we could have presented all definitions and theorems up to here in the more general way, but I believe that the test substitutions and the test substitution set condition are already complicated enough as it stands. On the other hand, once

[6]In [**?**], this theorem is given for a clause *set* $\mathcal{C}$ rather than a clause C. This generalisation is treated in the next section.

we have understood how resolution K-transformations work for the elimination of one clause, it is easy to understand the general case.

Furthermore, we shall treat the issue of *finding* a resolution K-transformation. Most of the statements we make in this context only apply to the elimination of *one* clause. To be precise, we shall present a search procedure for the elimination of *one* clause that is complete in a certain sense. We shall also present a search procedure for the elimination of *several* clauses at the same time, but we shall waive completeness since the search space is too large.

Our description of the search for a transformation for *one* clause will be quite detailed, whereas in the case of *several* clauses, it will be rough.

For all of these reasons, we shall treat the issue of eliminating *several* clauses separately.

Now let us continue to do so.

If we eliminate a subset $\mathcal{C}$ of Φ, the remaining clauses (that is, $\Phi \setminus \mathcal{C}$) have to be transformed such that Φ is satisfiable if and only if $\Upsilon(\Phi \setminus \mathcal{C})$ is satisfiable. Nothing new up to here.

As before, the transformation is characterised by a clause set S_Υ. Typically this set consists of the clauses in $\mathcal{C}$ plus some of their self resolvents, resolvents, and subsumed clauses.

The test substitution set condition must be generalised, but this is very easy:

> If a clause set S_Υ meets the test substitution set condition for each clause $C \in \mathcal{C}$, then S_Υ characterises a resolution K-transformation for $\mathcal{C}$.

Recall that the test substitution set is constructed by unifying the literals of a clause C with literals from a set L_Υ, the set of *characteristic* literals. L_Υ depends on S_Υ in a straightforward way (see page **??**).

When we check the test substitution set condition for a clause set $\mathcal{C}$, we must compute an individual test substitution set for each $C \in \mathcal{C}$. This test substitution set may be more complicated than in the examples before, but this is only because S_Υ will typically be larger, since it is supposed to characterise a transformation for a clause set $\mathcal{C}$ rather than a single clause C.

The construction of the test substitution set has not changed. Each test substitution set depends on *one* clause $C \in \mathcal{C}$ and a clause set S_Υ. There is no interdependence between the clauses in $\mathcal{C}$ as far as the test substitutions are concerned.

Definition 2.6.1 Let $\mathcal{C}$ and S_Υ be sets of clauses. For each $C \in \mathcal{C}$, let us write Σ_C for the set of test substitutions for C and S_Υ (see Def. **??**).

S_Υ characterises a resolution K-transformation for $\mathcal{C}$ if for all $C \in \mathcal{C}$ and all $\sigma \in \Sigma_C$:

$$\vec{\forall}\Upsilon(A_1\,\sigma_{gr}) \wedge \ldots \wedge \vec{\forall}\Upsilon(A_n\,\sigma_{gr}) \Rightarrow \vec{\forall}\Upsilon(A_{n+1}\,\sigma_{gr}) \vee \ldots \vee \vec{\forall}\Upsilon(A_m\,\sigma_{gr}) \qquad (16)$$

is a tautology.

$\diamond$

So far eliminations for clause sets do not seem to be much more complicated than eliminations of a single clause. However this only concerns the verification that a clause set S_Υ characterises a transformation for a clause set $\mathcal{C}$.

Finding this set S_Υ is a completely different matter! We have said that "typically this set consists of the clauses in $\mathcal{C}$ plus some of their self resolvents, resolvents, and subsumed clauses". Even if we impose restrictions such as limiting the length of these clauses, usually this is still a huge set! In order to even have a chance of finding a clause set S_Υ that does the job we must be very careful about checking simple candidates for S_Υ before the more complicated candidates.

We shall turn to this question in the next chapter, where we describe the search for a clause set S_Υ, and in Chapter **??**, where we treat the implementation of this search.

2.7 Summary

In this chapter, we have seen what a clause K-transformation is, we have introduced the criteria that must be fulfilled for a clause K-transformation, and we have seen how a clause K-transformation eliminates redundancies in an inference system. We have introduced all formal concepts we need.

This formalism has been developed in [**?**], although the definitions are slightly different sometimes. For example, [**?**] says explicitly that $D \in \Upsilon(D)$, whereas our definition of resolution K-transformations contains this case implicitly.

When we have a clause set S_Υ, we know how to check whether the transformation characterised by S_Υ is faithful. But we do not know much about what might be a reasonable candidate for S_Υ. This is important whether we want to find a transformation by hand or automatically. In the next chapter we shall investigate the question how clause K-transformations can be *found*.

Chapter 3

Finding Resolution K-Transformations

In [?] it is said that S_Υ is "usually C together with some of its self resolvents and subsumed clauses".

For most of the clauses I investigated, my experience is that S_Υ should contain clauses that are subsumed by C. I have not found much evidence that it is worthwhile to investigate self resolvents. Self resolvents are important in at least two contexts, but not in a way that could easily be exploited for our purposes. We shall treat this aspect in Section **??**.

Until now we should say that finding a set S_Υ and selected literals for a clause C such that (**??**) holds is a creative process. For many clauses C however there is a *finite* set in which we are likely to find a set S_Υ that will yield a sound and complete transformer. This set was discovered investigating a particular clause from the viewpoint of semantics, and therefore I shall introduce my technique with this example.

3.1 Semantic Consideration

Let us consider a clause with a 3-place predicate, namely:

$$C := R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z). \tag{17}$$

We want to find a clause set S_Υ that characterises a transformation for this clause. We have seen how this works for the transitivity clause (see Example **??**). Therefore our first try would be to take $S_\Upsilon = \{C\}$, where $\neg R(w,x,y)$ is the selected literal.

The selected literal has only variables as arguments. All variables are different. Thus Theorem **??** applies and we only need to test one single ground instance. $\Upsilon_1(R(a,b,c)) \wedge \Upsilon_1(R(b,c,d)) \Rightarrow \Upsilon_1(R(a,b,d))$ is

$$\begin{array}{rcl} (R(a,b,c) & \wedge & (\forall x.R(b,c,x) \Rightarrow R(a,b,x)) \\ \wedge R(b,c,d) & \wedge & (\forall x.R(c,d,x) \Rightarrow R(b,c,x))) \\ \Rightarrow (R(a,b,d) & \wedge & (\forall x.R(b,d,x) \Rightarrow R(a,b,x))). \end{array} \tag{18}$$

At this point we should give some intuition for this relation. We start off with a binary relation R' and define: $R(a,b,c)$ iff $R'(a,b)$ and $R'(b,c)$. If we consider the graph of R', then $R(a,b,c)$ means intuitively: There is a path from a to c leading

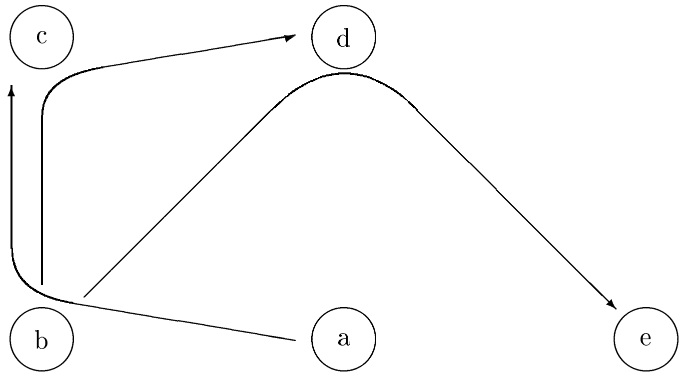

Figure 3.1: The basic relation given by $R(a,b,c)$, $R(b,c,d)$ and $R(b,d,e)$

through b and no other point. If R is closed under (**??**), $R(a,b,c)$ means: There is a path from a to c leading first through b and then possibly through other points. Note that the path from a to b must be direct, that is, an edge.

We must be careful about this intuition. For every binary relation R', we can indeed define R in the way described above.

Vice versa it does not always make sense. That means, we can construct a 3-place relation R for which there is no binary relation R' in the sense explained above. As long as we stick to the intuition, we are misled to believe that Υ_1 is already the transformer we are looking for.

This is not the case. (**??**) is not a tautology, and we can also give an example to see that Υ_1 is not a complete transformation.

Example 3.1.1 Let $\{R(a,b,c), R(b,c,d), R(b,d,e)\}$ be our basic 3-place relation. It is depicted by Fig.**??**. At this point we see that the idea of a binary relation R' of which R is derived is not appropriate here. For the idea to be appropriate, the basic relation would have to contain $R(a,b,d)$. However, $R(a,b,d)$ can be derived using (**??**). In a further step, we can derive $R(a,b,e)$. This is the whole closure of the basic relation under (**??**).

Now in contrast suppose we had transformed the basic relation using Υ_1. We have

$$
\begin{array}{lcl}
C1 & : & R(a,b,c) \\
C2 & : & R(b,c,d) \\
C3 & : & R(b,d,e) \\
R1 & : & R(b,c,x) \Rightarrow R(a,b,x) \\
R2 & : & R(c,d,x') \Rightarrow R(b,c,x') \\
R3 & : & R(d,e,x'') \Rightarrow R(b,d,x'')
\end{array}
$$

From this we can derive $R(a,b,d)$, but we cannot derive $R(a,b,e)$.

$\lhd$

The aim of a resolution K-transformation is to remove a clause from a clause set and replace it by more concrete clauses. This must be done such that we can still derive all consequences of the original clause set. Now what does Υ_1 do? If $R(a,b,c)$ is part of the basic relation, Υ_1 adds $R(b,c,x) \Rightarrow R(a,b,x)$ to the clause set. However the example showed that evidently this clause is *too* concrete. Clauses of this kind will not be general enough to derive all the original consequences.

One natural idea is to make $R(b,c,x) \Rightarrow R(a,b,x)$ more general. This clause has a negative literal and a positive literal. From the operational view this clause gets an instance of $R(b,c,x)$ as input and produces an instance of $R(a,b,x)$ as output. If you consider that we input unary ground clauses into $R(b,c,x) \Rightarrow R(a,b,x)$ to get other unary ground clauses as output, then it is intuitive that the *input* literal of $R(b,c,x) \Rightarrow R(a,b,x)$ must be generalised. After all, it is $R(b,c,x)$ that has to be unified with a ground literal, not $R(a,b,x)$.

Another way to look at it is to start with the original clause C and make it more specific. Let us look at C from a semantic point of view. Comparing the first and the last literal of C we note that they both start with '$R(w,x$'. Having only unary ground clauses and C, this implies an invariant of the resolution process: In order to derive $R(r,s,t)$, the basic relation must contain a literal of the form $R(r,s,u)$ for some constant term u. Therefore it is probably safe to instantiate C in these very arguments. Transforming $R(a,b,c)$, for example, we add $R(a,b,y) \wedge R(b,y,x) \Rightarrow R(a,b,x)$ to the clause set.

If we still want to express the transformation in terms of a clause set S_Υ, we have to construct a clause whose resolvent with $R(a,b,c)$ is $R(a,b,y) \wedge R(b,y,x) \Rightarrow R(a,b,x)$. This clause is

$$R(w,x,v) \wedge R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z) \tag{19}$$

where $R(w,x,v)$ is the selected literal.

These were intuitive ideas and by no means precise arguments. We have to prove formally that the intuition was right.

If we take $S_{\Upsilon_2} = \{R(w,x,v) \wedge R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)\}$ and select $R(w,x,v)$ we have to test
$\Upsilon_2(R(a,b,c)) \wedge \Upsilon_2(R(b,c,d)) \Rightarrow \Upsilon(R(a,b,d))$
which is

$$\begin{aligned}
(R(a,b,c) \quad &\wedge \quad (\forall x,y.R(a,b,x) \wedge R(b,x,y) \Rightarrow R(a,b,y)) \\
\wedge R(b,c,d) \quad &\wedge \quad (\forall x,y.R(b,c,x) \wedge R(c,x,y) \Rightarrow R(b,c,y))) \\
\Rightarrow (R(a,b,d) \quad &\wedge \quad (\forall x,y.R(a,b,x) \wedge R(b,x,y) \Rightarrow R(a,b,y))).
\end{aligned} \tag{20}$$

This is in fact a tautology, so we have found a sound and complete transformer for $R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)$.

3.2 Eliminating Clauses by Adding Instances

The transformer Υ for the previous example clause $C = R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)$ was found by a semantic consideration. We looked at some basic relation and asked : "What kind of conclusions must we be able to draw in our transformed clause set?" With some intuition we guessed an S_Υ that did the job. Now there are two things to be remarked about this transformer:

- The original clause C is not even needed in S_Υ.
- The clauses that Υ adds when transforming unit clauses[1] happen to be *instances* of C where some of the variables of C are replaced by terms occurring in the transformed clause.

I said "happen to be instances", but could this not be the general case?

[1] basic facts, so to speak

We eliminate a clause C by adding more or less concrete instances of C to the clause set. How concrete? The test substitution set condition gives us the criterion for this question.

There is a similar example in [**?**]. A transformer for the euclideanness clause is found by a semantic consideration much like the one we have made here.

In the previous example as well as in the euclideanness example in [**?**] we are completely dumbfounded at first that prefixing the clause with a literal that contains variables that do not occur elsewhere should yield a transformer for this clause. After all, we are making C *longer*, we are working with a clause that is *subsumed* by C. And where do these new variables suddenly come from?

If we look at *resolution K-transformations* as adding instances of C, these questions become quite clear. Prefixing a clause C with a literal and resolving with a unit clause is just a *technique* to obtain instances of C. Some of the variables in this literal are fresh, while others occur in C. The latter variables will be instantiated by terms in the unit clause, thus determining in which way C is instantiated. The former variables will be instantiated, too, but as they do not share with C, C will not be affected by this.

Prefixing a clause C with a literal will be used so often from now on that it deserves a formal definition.

Definition 3.2.1 Let C be a clause that has a predicate symbol P of arity m. Then

$$\neg P_1(x_1, x_2, \ldots, x_m) \tag{21}$$

is a *prefix literal* for C.

$\diamond$

The idea is that the x_i may or may not occur in C, but “may or may not” is not a formal condition, of course.

It may well be that a clause has more than one predicate symbol. Any of these can be used to create prefix literals.

Thus C extended by a prefix literal gives us a candidate for S_Υ.

3.2.1 One Extreme Case

For a clause C with an n-place predicate P, we can always find a transformer that is just as sound and complete as it is useless. Take

$$S_\Upsilon := \{\neg P(x_1, x_2, \ldots, x_n) \vee C\} \quad \text{where} \quad x_1, x_2, \ldots, x_n \notin C \tag{22}$$

Have a look at (**??**). For each $A_i\,\sigma_{gr}$, Υ adds C and, as always, $A_i\,\sigma_{gr}$. Thus

$$\Upsilon(A_i\,\sigma_{gr}) = \{A_i\,\sigma_{gr},\ C\} \tag{23}$$

Now the test substitution requires

$$(A_1\,\sigma\,\psi \wedge C) \wedge \ldots \wedge (A_n\,\sigma\,\psi \wedge C) \Rightarrow (A_{n+1}\,\sigma\,\psi \wedge C) \vee \ldots \vee (A_m\,\sigma\,\psi \wedge C) \tag{24}$$

to be a tautology. We make some simple equivalence transformations

$$\begin{aligned} & C \wedge A_1\,\sigma\,\psi \wedge \ldots \wedge A_n\,\sigma\,\psi \Rightarrow (C \wedge (A_{n+1}\,\sigma\,\psi \vee \ldots \vee A_m\,\sigma\,\psi)) \equiv \\ & (C \wedge A_1\,\sigma\,\psi \wedge \ldots \wedge A_n\,\sigma\,\psi \Rightarrow C) \wedge \\ & \qquad (C \wedge A_1\,\sigma\,\psi \wedge \ldots \wedge A_n\,\sigma\,\psi \Rightarrow (A_{n+1}\,\sigma\,\psi \vee \ldots \vee A_m\,\sigma\,\psi)) \equiv \\ & C \Rightarrow (A_1\,\sigma\,\psi \wedge \ldots \wedge A_n\,\sigma\,\psi \Rightarrow A_{n+1}\,\sigma\,\psi \vee \ldots \vee A_m\,\sigma\,\psi) \end{aligned} \tag{25}$$

$(A_1\,\sigma\,\psi \wedge \ldots \wedge A_n\,\sigma\,\psi \Rightarrow A_{n+1}\,\sigma\,\psi \vee \ldots \vee A_m\,\sigma\,\psi)$ is an *instance* of C and thus follows from C. Thus (**??**) is indeed a tautology.

Intuitively this transformation is complete because C is not really eliminated. If there is any unit clause $P(s_1, s_2, \ldots, s_n)$ the transformer will add a trivial instance of C, namely C itself, to the clause set.

Of course, this is not the kind of transformation we look for, but it is conceptually important to understands what a transformer does in the worst case.

3.2.2 The Other Extreme Case

We have seen the transitivity example (see Example **??**), where transformation works by resolving with the transitivity clause itself, and euclideanness, where transformation works by resolving with the euclideanness clause itself plus one other clause. How does this fit into the scheme presented above? Quite well, as we shall see. In these cases we were lucky enough to find a prefix literal that is identical to a negative literal that already occurs in the clause.

It will suffice to look at transitivity to make this point clear.

We have seen in Example **??** that $S_\Upsilon = \{R(x,y) \wedge R(y,z) \Rightarrow R(x,z)\}$ with $R(x,y)$ as selected literal gives us a transformation for C. For a unit clause $R(a,b)$ we have

$$\Upsilon(R(a,b)) = \{R(a,b), \forall z.R(b,z) \Rightarrow R(a,z)\}. \tag{26}$$

Now we present a slightly different transformer:

$$S_{\Upsilon_2} = \{R(x,y) \wedge R(x,y) \wedge R(y,z) \Rightarrow R(x,z)\} \tag{27}$$

where the first $R(x,y)$ is selected.

If we look at clauses as sets then there is no difference between the two transformers, but for now just accept that the clause has two occurrences of the same literal. Let us transform $R(a,b)$ again

$$\Upsilon_2(R(a,b)) = \{R(a,b), \forall z.R(a,b) \wedge R(b,z) \Rightarrow R(a,z)\}. \tag{28}$$

There is not much of a difference between (**??**) and (**??**); the two clause sets are equivalent. If this does not convince you, it must certainly convince you that

$$\begin{array}{l} (R(a,b) \wedge (\forall z.R(a,b) \wedge R(b,z) \Rightarrow R(a,z)) \\ \wedge R(b,c) \wedge (\forall z.R(b,c) \wedge R(c,z) \Rightarrow R(b,z))) \\ \Rightarrow R(a,c) \wedge \forall z.R(a,c) \wedge R(c,z) \Rightarrow R(a,z) \end{array}$$

is a tautology.

3.2.3 The General Case

Of course I am not really suggesting to prefer (**??**) over (**??**). I am just suggesting a way to look *at* transformers and to look *for* transformers. To find a transformer for C, systematically generate all potential prefix literals $\neg L$. There are only finitely many of them[2]. This will yield a *finite* set $\mathcal{S}_\Upsilon$ of candidates $\neg L \vee C$ for S_Υ. The selected literal is always the prefix literal. Now inspect every subset of $\mathcal{S}_\Upsilon$ to see whether the test substitution holds for this set. If it does, we have found a transformer.

If a prefix literal happens to be identical with a literal in C, an *optimisation* is possible. Instead of taking $\neg L \vee C$ take $\neg L \vee (C \setminus \{\neg L\})$ as a candidate for S_Υ. This is what is done in the transitivity example.

We have seen the extreme cases: The prefix literal shares *no* variables with C vs. the prefix literal is identical to another negative literal of C. We have also seen

[2]modulo renaming of the variables that do not occur in C

that the former case is definitely *not* what we are looking for, whereas the latter case is ideal.

It should be no surprise that between these extreme cases, we would like to have a prefix literal that shares as many variables as possible with C. This yields rather *concrete* instances of C. The more concrete the instances of C are, the more the transformer helps to reduce the number of possible inferences in automated theorem proving.

This point will be reflected when we turn to implementation. We shall attempt to find prefixed clauses where the prefix literal is as concrete as possible.

3.2.4 Excluding Positive Literals

One further observation will help to reduce the number of candidates for S_Υ. If we want to transform a clause $C = A_1 \wedge \ldots \wedge A_n \Rightarrow A_{n+1} \vee \ldots \vee A_m$, no prefix literal should equal any A_i for $i = n+1, \ldots m$.

The reason is the following: If a literal L is transformed, an instance of C would be added that has the form $(\ldots \Rightarrow \ldots L \ldots)$. As always, L remains in the clause set, too, giving us something like $L \wedge (\ldots \Rightarrow \ldots L \ldots)$, which is equivalent to L. But let us formulate this idea precisely.

Corollary 3.2.2 Let $C = A_1 \wedge \ldots \wedge A_n \Rightarrow A_{n+1} \vee \ldots \vee A_m$ be a clause and assume that for some $k \in \{n+1, \ldots, m\}$ and some clause set $S_{\Upsilon'}$

$$S_\Upsilon := S_{\Upsilon'} \cup \{A_k \wedge A_1 \wedge \ldots \wedge A_n \Rightarrow A_{n+1} \vee \ldots \vee A_m\} \tag{29}$$

characterises a resolution K-transformation Υ for C. Then Υ' is also a resolution K-transformation for C.

Proof:
Consider the test substitution set condition (**??**). In order to check this conditon, the transformation has to be applied to positive ground literals only. Thus if we show that

$$\Upsilon'(L) \quad \text{is equivalent to} \quad \Upsilon(L) \tag{30}$$

for any positive ground literal L, we are done. Since C is a tautology (it contains the literals $\neg A_k$ and A_k), resolution on the literals $\neg A_k$ and L results in a clause that contains L, and thus is subsumed by the unit clause L. $\Upsilon(L)$ contains this new clause, whereas $\Upsilon'(L)$ does not.

As always, the unit clause L is contained in $\Upsilon(L)$, so the new clause containing L can be removed without changing satisfiability, resulting in $\Upsilon'(L)$. Thus $\Upsilon'(L)$ and $\Upsilon(L)$ are equivalent.

□

3.2.5 Another Restriction for Prefix Literals

I have described the way to find prefix literals: instantiate the prefix literal with the variables of the clause in all possible ways. For the examples I have studied, I observed however that if a transformer was found for some clause, there was also some transformer where the prefix literal was a generalisation of a literal of the clause.

This means that for a clause $C = A_1 \vee \ldots \vee A_n$ and a prefix literal L, there should be a matcher μ such that $L\,\mu = A_i$ for some i. μ may only instantiate the variables that do *not* occur in C.

Take $C = p(x, y) \wedge p(y, z) \Rightarrow p(x, z)$, for example. Following the restriction described above, we would allow $p(x, v)$ as prefix literal, but we would not allow $p(v, x)$, because no literal in C has x as second argument.

This restriction reduces the search space, and I have not encountered an example where we fail to find any transformer because of this restriction. However we may fail to find a *particular* transformer because of this restriction. Furthermore, so far I have no proof that we cannot fail to find any transformer because we impose this restriction.

In Chapter **??** we shall see how we can automate the process of finding a transformation. The program leaves the choice whether or not to restrict the search for prefix literals to the user.

3.3 Euclideanness Revisited

In the last section I presented a heuristic for finding a transformer for a clause C. This has been implemented so that this kind of transformer can be found automatically, if it exists. The program is called TRANSFORMATOR. In order to test TRANSFORMATOR I took another look at an example that was already examined by Ohlbach, Gabbay, and Plaisted ([**?**]): euclideanness. We have already seen euclideanness. It is defined by (**??**).

The transformer presented in [**?**] is shown in (**??**).

This transformer is found by a semantic consideration that is analogous to the one in Section **??**. A graph is drawn for some simple example and the operational "behaviour" of certain clauses is investigated in order to add the right "arrows" into the graph.

Of course I was very interested to see whether TRANSFORMATOR would find this transformer automatically.

Well, it probably would have enumerated this transformer some time, but first it came up with

$$S_\Upsilon = \{R(v, z) \wedge R(x, y) \wedge R(x, z) \Rightarrow R(z, y)\}. \tag{31}$$

Let us have a look at the test substitution set condition.
$\Upsilon(R(a, b)) \wedge \Upsilon(R(a, c)) \Rightarrow \Upsilon(R(c, b))$ is

$$\begin{aligned} (R(a,b) \quad &\wedge \quad (\forall xy.R(x,y) \wedge R(x,b) \Rightarrow R(b,y)) \\ \wedge R(a,c) \quad &\wedge \quad (\forall xy.R(x,y) \wedge R(x,c) \Rightarrow R(c,y))) \\ \Rightarrow (R(c,b) \quad &\wedge \quad (\forall xy.R(x,y) \wedge R(x,b) \Rightarrow R(b,y))). \end{aligned} \tag{32}$$

This is indeed a tautology, so Υ is a transformer for the euclideanness clause. This transformer is characterised by a clause set S_Υ that is a singleton, and thus it is much simpler than the one presented in [**?**], where S_Υ consists of two clauses.

3.4 Horn Clauses without Function Symbols

The clauses we have examined contain no function symbols. They match the following pattern

$$\begin{aligned} & R(x_1^1, x_1^2, \ldots, x_1^r) \wedge R(x_2^1, x_2^2, \ldots, x_2^r) \wedge \ldots \wedge R(x_n^1, x_n^2, \ldots, x_n^r) \\ \Rightarrow \quad & R(x_{n+1}^1, x_{n+1}^2, \ldots, x_{n+1}^r) \end{aligned} \tag{33}$$

where not all variables have to be different[3].

Actually we could also allow function symbols. For example, we might have $R(f(x,y)) \Rightarrow R(f(y,x))$. The essential point is that such a clause does not generate new *terms* through self resolution. This is different from a clause like condensed detachment that may generate an infinite sequence of new terms (see Section **??**). In our context, a clause like $R(f(x,y)) \Rightarrow R(f(y,x))$ is essentially the same as $R(x,y) \Rightarrow R(y,x)$. Therefore it would only complicate the matter if we considered function symbols here.

Symmetry, transitivity, reflexivity, euclideanness, certain permutation properties are all expressed by clauses of this kind. I only claim so far that my method works well for this kind of clauses.

We shall now look at a finite subclass of these clauses exhaustively. This is to illustrate that the technique for finding transformers I propose is an effective one. We could easily extend this study to larger clauses.

Some simple and natural properties binary relations can have are expressed by clauses of the form

$$R(u,v) \wedge R(w,x) \Rightarrow R(y,z) \quad \text{or} \quad R(w,x) \Rightarrow R(y,z). \tag{34}$$

The variables here are meta-variables, which means that in concrete examples usually some variables occur more than once. Up to variable renaming, there are only finitely many such clauses. The most naïve consideration would yield $2^6 = 64$ or $2^4 = 16$ clauses, respectively, but some simple observations will reduce this number tremendously. These are

- Clauses that can be obtained from each other by exchanging two variables should not be considered separately. This is not precisely the same thing as renaming, but almost.
- The order of the two negative literals is irrelevant.
- Clauses that can be obtained from each other by exchanging the first with the second argument in each literal should not be considered separately.
- Tautologies should not be considered at all.
- Clauses that are equivalent to shorter clauses should not be considered.

For each clause in this clause, I tried to find a transformation automatically. First I wanted to demonstrate that my heuristic is a suitable one. Furthermore, some of these clauses represent common properties of relations. Others, of course, seem quite artificial.

Table **??** shows the results. In all cases the first literal is the selected literal.

Clause no.6 is the euclideanness clause, no.7 is symmetry, no.11 is transitivity. No.12 is also a nice clause, we might call it "circularity". Its transformer was by far the hardest to find, but was still found in few seconds. It is the only one of the above examples where S_Υ consists of more than one clause.

The other clauses are not so intuitive.

It is not accidental that for clause no.16, S_Υ contains the clause itself, but the order of the literals is interchanged. Coincidentally this clause had been input to TRANSFORMATOR as $R(x,x) \wedge R(y,z) \Rightarrow R(z,y)$. The program interchanged the literals because the *second* literal of the original clause must be selected for the transformation to be complete. Of course we could have interchanged the literals in the original clause so that S_Υ literally contains the clause itself. However it is

[3]If they are all different, this would yield a very trivial example.

	Clause	S_Υ **for this clause**	**pr.**
1	$R(x,x) \wedge R(x,y) \Rightarrow R(y,z)$	$\{\boldsymbol{R(x,x)} \wedge R(x,y) \Rightarrow R(y,z)\}$	no
2	$R(x,x) \wedge R(x,y) \Rightarrow R(z,y)$	$\{\boldsymbol{R(x,y)} \wedge R(x,x) \Rightarrow R(z,y)\}$	no
3	$R(x,x) \wedge R(x,y) \Rightarrow R(y,x)$	$\{\boldsymbol{R(x,y)} \wedge R(x,x) \Rightarrow R(y,x)\}$	no
4	$R(x,x) \Rightarrow R(x,y)$	$\{\boldsymbol{R(x,x)} \Rightarrow R(x,y)\}$	no
5	$R(x,x) \wedge R(y,y) \Rightarrow R(x,y)$	$\{\boldsymbol{R(x,x)} \wedge R(y,y) \Rightarrow R(x,y)\}$	no
6	$R(x,y) \wedge R(x,z) \Rightarrow R(y,z)$	$\{\boldsymbol{R(v,y)} \wedge R(x,y) \wedge R(x,z) \Rightarrow R(y,z)\}$	yes
7	$R(x,y) \Rightarrow R(y,x)$	$\{\boldsymbol{R(x,y)} \Rightarrow R(y,x)\}$	no
8	$R(x,y) \Rightarrow R(x,x)$	$\{\boldsymbol{R(x,y)} \Rightarrow R(x,x)\}$	no
9	$R(x,y) \wedge R(y,x) \Rightarrow R(x,z)$	$\{\boldsymbol{R(x,y)} \wedge R(y,x) \Rightarrow R(x,z)\}$	no
10	$R(x,y) \wedge R(y,x) \Rightarrow R(x,x)$	$\{\boldsymbol{R(x,y)} \wedge R(y,x) \Rightarrow R(x,x)\}$	no
11	$R(x,y) \wedge R(y,z) \Rightarrow R(x,z)$	$\{\boldsymbol{R(x,y)} \wedge R(y,z) \Rightarrow R(x,z)\}$ or $\{\boldsymbol{R(y,z)} \wedge R(x,y) \Rightarrow R(x,z)\}$	no
12	$R(x,y) \wedge R(y,z) \Rightarrow R(z,x)$	$\{\boldsymbol{R(x,v)} \wedge R(x,y) \wedge R(y,z) \Rightarrow R(z,x)$, $\boldsymbol{R(v,x)} \wedge R(x,y) \wedge R(y,z) \Rightarrow R(z,x)\}$	yes
13	$R(x,x) \wedge R(y,z) \Rightarrow R(x,y)$	$\{\boldsymbol{R(y,z)} \wedge R(x,x) \Rightarrow R(x,y)\}$	no
14	$R(x,x) \wedge R(y,z) \Rightarrow R(x,z)$	$\{\boldsymbol{R(y,z)} \wedge R(x,x) \Rightarrow R(x,z)\}$	no
15	$R(x,x) \wedge R(y,z) \Rightarrow R(y,y)$	$\{\boldsymbol{R(y,z)} \wedge R(x,x) \Rightarrow R(y,y)\}$	no
16	$R(x,x) \wedge R(y,z) \Rightarrow R(z,y)$	$\{\boldsymbol{R(y,z)} \wedge R(x,x) \Rightarrow R(z,y)\}$	no

Table 3.1: Transformers for clauses of length 2 and 3 with binary predicates

instructive to see that in some examples, the transformation is characterised by the clause itself, but the choice of the selected literal is significant!

The last column in the table says whether the prefix literal was really needed, or whether it happened to be identical with one literal already appearing in the clause. There are two examples where the prefix literal is actually needed, namely euclideanness and what we called "circularity". This is not much, but at least these two examples are among the more interesting ones in the list. Apart from that, there are other examples when we take predicates with more than two arguments or clauses with more than three literals (e.g. see Section **??**). So it can be claimed that systematically generating prefix literals is an effective way to find transformers.

For clause no. 11 (transitivity), two transformers are given. This is not to say that transitivity is the only clause for which more than one transformer exists. The table just shows the first transformer TRANSFORMATOR found for each clause.

I made sure however, that if the transformer given requires a prefix literal in the strict sense, then there is no transformer that does not.

I emphasise that the above list is exhaustive[4]. Among all clauses of this class there was not a single one for which no proper (i.e. not of the kind of Section **??**) transformer was found!

We could easily expand this class of clauses by taking predicates with more than two arguments or longer clauses. Of course this makes it much harder to find a transformation.

3.5 The Role of Self Resolvents

We have defined self resolution in Section **??**. In this chapter we have neglected it so far. In [**?**] there is a strong emphasis on self resolvents. To find a clause set S_Υ that characterises a transformation for a clause C, it is proposed to take C and some of its self resolvents. The set of self resolvents may be infinite, but it can be

[4] modulo the mentioned "equivalences"

enumerated. Thus one could systematically enumerate all sets of self resolvents and check for each set whether it fulfills the test substitution set condition.

Nobody claims that this procedure *must* yield a transformer for every clause. Evidence that this procedure is effective would have to come from examples, that is, clauses where this technique found a transformation. This is no different from the heuristic I propose (prefixed clauses), and for this reason I presented the examples of Section **??** to demonstrate that my heuristic is effective. The examples of Section **??** make it easy to believe that we could go on like this for bigger clauses, even if the complexity would certainly become a problem then.

As far as self resolvents are concerned, my experience is that they are important in two contexts:

3.5.1 Condensed Detachment

This is a clause containing a function symbol. We shall look at condensed detachment in Section **??**. It is also thoroughly investigated in [**?**]. There S_Υ contains *infinitely* many self resolvents of the condensed detachment clause. Thus the transformation may add infinitely many clauses. This is not a suitable input for an automated theorem prover. In [**?**] it is shown how these infinitely many clauses can be transformed such that they can be input to an automated theorem prover. This may result in dramatic improvements of efficiency.

Condensed detachment is an important clause, actually the standard example of a simple clause making automated theorem proving extremely hard. Thus finding a transformer for this clause is in itself an impressive result. However, it requires a very sophisticated argument to cope with the infinitely many clauses the transformer generates at first.

Thus it may well be that self resolvents are important to find transformers for other clauses similar to condensed detachment, but I would not know how one could automate the process of coping with infinity, such that the process of finding such a transformer is completely automated. I do not say that it is not possible, however.

3.5.2 Permutation Properties

A permutation property is expressed by a clause of the form

$$C := R(x_1, x_2, \ldots, x_n) \Rightarrow R(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}) \tag{35}$$

where the x_i are all different variables and $\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ is a permutation. Such a clause is called a *permutation clause*.

In a beginner's textbook (e.g. see [**?**]) about algebra we find that a permutation has a degree d, such that

$$\pi^d = \mathrm{id}$$

This means, permuting a tuple d times we will return to the original tuple.

The following corollary says how permutation clauses can be transformed.

Corollary 3.5.1 A transformation for a permutation clause C is characterised by all self resolvents of C from level[5] 0 to level $d - 2$.

Proof:

Let us write S_i for the self resolvent of level i. Then the self resolvents up to the

[5]The "level" was defined in Def. **??**.

$(d-2)$th level are

$$\begin{array}{l} S_0 = R(x_1, x_2, \ldots, x_n) \Rightarrow R(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}) \\ S_1 = R(x_1, x_2, \ldots, x_n) \Rightarrow R(x_{\pi^2(1)}, x_{\pi^2(2)}, \ldots, x_{\pi^2(n)}) \\ S_2 = R(x_1, x_2, \ldots, x_n) \Rightarrow R(x_{\pi^3(1)}, x_{\pi^3(2)}, \ldots, x_{\pi^3(n)}) \\ \vdots \\ S_{d-2} = R(x_1, x_2, \ldots, x_n) \Rightarrow R(x_{\pi^{d-1}(1)}, x_{\pi^{d-1}(2)}, \ldots, x_{\pi^{d-1}(n)}) \end{array} \tag{36}$$

$S_\Upsilon := \{S_0, S_1, \ldots S_{d-2}\}$. For the test substitution set condition replace each x_1 by a_i.

$$\Upsilon(R(a_1, a_2, \ldots, a_n)) \Rightarrow \Upsilon(R(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}))$$

is

$$\left(\begin{array}{l} R(a_1, a_2, \ldots, a_n) \wedge \\ R(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}) \wedge \\ R(a_{\pi^2(1)}, a_{\pi^2(2)}, \ldots, a_{\pi^2(n)}) \wedge \\ \vdots \\ R(a_{\pi^{d-1}(1)}, a_{\pi^{d-1}(2)}, \ldots, a_{\pi^{d-1}(n)}) \end{array} \right) \Rightarrow \left(\begin{array}{l} (R(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}) \wedge \\ R(a_{\pi^2(1)}, a_{\pi^2(2)}, \ldots, a_{\pi^2(n)}) \wedge \\ \vdots \\ R(a_{\pi^{d-1}(1)}, \ldots, a_{\pi^{d-1}(n)}) \wedge \\ R(a_{\pi^d(1)}, R(a_{\pi^d(2)}, \ldots, a_{\pi^d(n)}) \end{array} \right)$$

It can easily be seen that the right and the left side are equal up to the order of literals. The essential point is that the last line of the right side equals the first line of the left side, because $\pi^d = \mathrm{id}$.

□

Thus in order to eliminate permutation clauses, S_Υ should contain self resolvents, but in a predictable way. With the above corollary we have solved the problem of finding a transformation for permutation clauses once and for all.

By the way, the heuristic I proposed, that is, generating prefixed literals as candidates for S_Υ, finds the transformers for permutation clauses, too. At least in theory.

Consider $R(x_1, x_2, x_3, x_4, x_5) \Rightarrow R(x_2, x_3, x_4, x_5, x_1)$. There are 1546 prefixed clauses for this clause. Do not worry about how I came up with this number. Roughly speaking, five of these are the self resolvents that characterise the transformation. It would be an understatement to say that finding a five element subset having a certain property in a 1546 element set is like finding a needle in a haystack...

Conclusion

Thus we have seen two contexts in which self resolvents are useful, one of which is so complicated that it is hard to see that a transformation of this kind could be found automatically, whereas the other is trivial in the sense that the result can be given once and for all.

I have not encountered any other contexts where self resolvents are useful for finding transformations. Of course this is not to say that there might not be other contexts.

In Chapter **??** we shall treat the issue of implementation. Should there be any other examples where a transformation for a clause can be characterised by a finite number of self resolvents: My program (TRANSFORMATOR) is prepared for it. That is, I have implemented a search process that systematically tries sets of self resolvents of a clause C as candidates for S_Υ, where S_Υ characterises the transformation. Actually this had been implemented before the idea of *prefixed clauses* was born.

I emphasise that the example of condensed detachment is in itself very important. It is definitely much more important than most of the examples of Section **??**, which are rather artificial.

I only say that it seems to be very difficult to generalise the argument in [**?**] to find transformers for other clauses than condensed detachment. I concentrated mainly on finding transformers automatically. Therefore the relevance of self resolvents is rather limited for our purposes.

3.6 Condensed Detachment

Now we will look at an important clause containing a function symbol, namely, condensed detachment. Refuting clause sets that contain the condensed detachment clause can be surprisingly difficult. This is not alone for the fact that the condensed detachment clause contains a function symbol. The condensed detachment clause may generate new terms for ever and ever by continued resolution. We shall soon see why.

We have not addressed the question yet whether the concept of eliminating a clause by adding instances of it also works for such a clause.

Unfortunately our result for the condensed detachment clause is a negative one, but not in the sense that we cannot find a transformer for the condensed detachment clause, but rather that the transformer is pathological. In the typical setting where the condensed detachment clause occurs this transformer adds the condensed detachment clause itself, rather than more specific instances of condensed detachment. Thus nothing is gained.

We present these results anyway for two reasons. First, also a negative result is instructive in some way. Secondly, the approach presented here might be useful for other clauses containing function symbols.

Hans Jürgen Ohlbach presents a transformer for the condensed detachment clause that is indeed useful. For some clause sets containing the condensed detachment clause dramatic improvements in the performance of automated theorem proving have been achieved using this transformer.

Condensed detachment is the predicate logic encoding of the modus ponens rule. "Predicate logic encoding" means that the *terms* in predicate logic are interpreted as *formulæ* in some object logic, say propositional logic for our purposes. Predicate logic is our meta logic. So we make *statements in* predicate logic *about* propositional logic. $t(x)$ means "x is a true formula", $t(i(x,y))$ means "x implies y". Then modus ponens writes as

$$C := t(w) \wedge t(i(w,x)) \Rightarrow t(x). \tag{37}$$

Its meaning is simply the following: If w is true and if it is true that w implies x, then x is also true.

Let us see why the condensed detachment clause generates new terms. Suppose we have two clauses $t(a)$ and $t(b)$. Resolving each of these with the condensed detachment clause on the first literal we get the clauses

$$t(i(a,x)) \Rightarrow t(x) \quad \text{and} \quad t(i(b,x)) \Rightarrow t(x) \tag{38}$$

Now we can resolve these two clauses with each other, which yields

$$t(i(a,i(b,x))) \Rightarrow t(x) \tag{39}$$

$i(a,i(b,x))$ is a new term generated through resolution. We could go on like this forever.

Naïve Approach

Let us first try the method introduced in Section **??** in the most naïve way. The variables occurring in (**??**) are w and x, so the candidates for prefix literals are $t(w)$ and $t(x)$. Let us look first at $S_\Upsilon = \{t(w) \wedge t(w) \wedge t(i(w,x)) \Rightarrow t(x)\}$. Then $\Upsilon(t(a)) \wedge \Upsilon(t(i(a,b))) \Rightarrow \Upsilon(t(b))$ becomes

$$\begin{array}{rcl} (t(a) & \wedge & (\forall x.t(a) \wedge t(i(a,x)) \Rightarrow t(x)) \\ \wedge t(i(a,b)) & \wedge & (\forall x.t(i(a,b)) \wedge t(i(i(a,b),x)) \Rightarrow t(x))) \\ \Rightarrow (t(b) & \wedge & (\forall x.t(b) \wedge t(i(b,x)) \Rightarrow t(x))) \end{array} \tag{40}$$

This is not a tautology. So let us try $S_\Upsilon = \{t(x) \wedge t(w) \wedge t(i(w,x)) \Rightarrow t(x)\}$ instead. $\Upsilon(t(a)) \wedge \Upsilon(t(i(a,b))) \Rightarrow \Upsilon(t(b))$ becomes

$$\begin{array}{rcl} (t(a) & \wedge & (\forall x.t(x) \wedge t(i(x,a)) \Rightarrow t(a)) \\ \wedge t(i(a,b)) & \wedge & (\forall x.t(x) \wedge t(i(x,i(a,b))) \Rightarrow t(i(a,b)))) \\ \Rightarrow (t(b) & \wedge & (\forall x.t(x) \wedge t(i(x,b)) \Rightarrow t(b))) \end{array} \tag{41}$$

This is not a tautology either. $S_\Upsilon = \{\ t(w) \wedge t(w) \wedge t(i(w,x)) \Rightarrow t(x),$ $t(x) \wedge t(w) \wedge t(i(w,x)) \Rightarrow t(x)\ \}$ will not lead to success either, as can easily be inspected.

Finding Υ by an Operational View

The least thing that we can say is that my technique for finding prefix literals is not able to find a transformer for the condensed detachment clause. But we should not yet give up the idea that eliminating a clause works by adding instances of it. In order to get a first idea what kind of instances are needed, we shall make the simplifying assumption that we only transform unit clauses. In the formal proofs, we shall drop this assumption, of course.

We shall take an operational view of the condensed detachment clause. Consider a unit clause $t(i(s_1,s_2))$, where s_1, s_2 are arbitrary terms. From this clause, $t(s_2)$ might be derived through condensed detachment. If s_2 has the form $i(s_3,s_4)$, s_4 might be derived in a later step and so forth. Any clause that is derived might be used as “input” for the condensed detachment clause. If we consider the clauses in our original clause set as “trivially derived”, we can also say that evidently nothing else but a derived clause is ever used as input to the condensed detachment clause. So in order to get rid of the general condensed detachment clause, we must make sure that appropriate instances of the condensed detachment are available instead.

The condensed detachment clause has two negative literals $t(w)$ and $t(i(w,x))$, that is two “inputs”, from the operational point of view. Any clause that is derived might serve as $t(w)$ or as $t(i(w,x))$, we cannot predict that. Does this mean that we have to add two instances of the condensed detachment clause for each such clause? No, we can choose with which literal we want to instantiate as long as we do it in the same way for all transformed clauses. After all, deriving a unit clause by condensed detachment requires *two* other unit clauses, say $t(s_1)$ and $t(i(s_1',s_2))$. Without transformation, $t(w)$ is unified with $t(s_1)$, and $t(i(w,x))$ is unified with $t(i(s_1',s_2))$. To this end, s_1 and s_1' must be unifiable. Now, in contrast, suppose we transform the clause set before starting our derivations. Depending on which literal we choose, either $t(s_1) \wedge t(i(s_1,x)) \Rightarrow t(x)$ or $t(s_1') \wedge t(i(s_1',s_2)) \Rightarrow t(s_2)$ will be added to the clause set. But *one* of these clauses will suffice to replace the general condensed detachment clause, we do not need both.

First we try first to instantiate $t(w)$. The following three examples illustrate how Υ should work.

$$\begin{aligned}
\Upsilon(t(a)) &= \{t(a), \\
&\quad (\forall x.t(a) \wedge t(i(a,x)) \Rightarrow t(x))\} \\
\Upsilon(t(i(a,b))) &= \{t(i(a,b)), \\
&\quad (\forall x.t(i(a,b)) \wedge t(i(i(a,b),x)) \Rightarrow t(x)), \\
&\quad \wedge(\forall x.t(b) \wedge t(i(b,x)) \Rightarrow t(x))\} \\
\Upsilon(t(i(a(i(b,c))))) &= \{t(i(a(i(b,c)))), \\
&\quad (\forall x.t(i(a,i(b,c))) \wedge t(i(i(a,i(b,c)),x)) \Rightarrow t(x)), \\
&\quad (\forall x.t(i(b,c)) \wedge t(i(i(b,c),x)) \Rightarrow t(x)), \\
&\quad \wedge(\forall x.t(c) \wedge t(i(c,x)) \Rightarrow t(x))\}
\end{aligned}$$

Looking at the first and second element in each of the above sets we see that we could easily simplify these sets in the same way that $p(a,b) \wedge (\forall x.p(a,b) \wedge p(b,x) \Rightarrow p(a,x))$ can be simplified to $p(a,b) \wedge (\forall p(b,x) \Rightarrow p(a,x))$ (see Section **??**). We shall not do so at this point for the sake of conceptual clarity.

Of course we can define Υ formally.

$$\Upsilon(t(s)) := \{t(s)\} \cup \Upsilon'(t(s)). \tag{42}$$

The auxiliary function Υ' is defined as follows:

$$\begin{aligned}
\Upsilon'(t(s)) &:= \{\forall x.t(s) \wedge t(i(s,x)) \Rightarrow t(x)\} && \text{if } s \text{ is a variable or constant} \\
\Upsilon'(t(i(s_1,s_2))) &:= \{\forall x.t(i(s_1,s_2)) \wedge t(i(i(s_1,s_2),x)) \Rightarrow t(x)\} && \\
&\quad \cup \Upsilon'(t(s_2)) && \text{else}
\end{aligned} \tag{43}$$

For every positive literal (unit clause), Υ gives us a *finite* set. This is because Υ is defined inductively on the structure of terms, and there is no such thing as an infinite term in our setting.

Describing Υ by a Clause set

The above definition is easy to understand, but it is not obvious that a resolution K-transformation is defined. For this we would need a clause set S_Υ together with some selected literals. Let us define

$$\begin{aligned}
S_\Upsilon \quad := \quad & \{t(y) \wedge t(y) \wedge t(i(y,z)) \Rightarrow t(z), \\
& t(i(v_1,y)) \wedge t(y) \wedge t(i(y,z)) \Rightarrow t(z), \\
& t(i(v_1,i(v_2,y))) \wedge t(y) \wedge t(i(y,z)) \Rightarrow t(z), \\
& t(i(v_1,i(v_2,i(v_3,y)))) \wedge t(y) \wedge t(i(y,z)) \Rightarrow t(z), \\
& \ldots\}
\end{aligned} \tag{44}$$

This set suggests already that we will have to extend the definition of prefix literals, but we are not ready to do so yet.

Note that the suffix of each clause is the condensed detachment clause, but with variables y and z instead of w and x as before. This should help to avoid confusion when we unify between S_Υ and C in order to verify the test substitution set condition.

S_Υ is *infinite*, and we will have to argue carefully why it still characterises the same transformation as (**??**).

A literal $t(i(s_1, i(s_2, \ldots, i(s_n, a) \cdots)))$, where s_i are arbitrary terms and a is a constant, will only unify with finitely many prefix literals in S_Υ; to be precise, with all prefix literals from $t(y)$ through $t(i(v_1, i(v_2, \ldots, i(v_n, y) \cdots)))$. In this case $\Upsilon(t(i(s_1, i(s_2, \ldots, i(s_n, a) \cdots))))$ is certainly finite.

In contrast, a literal $t(i(s_1, i(s_2, \ldots, i(s_n, w) \cdots)))$, where w is a variable, will unify with all prefix literals in S_Υ. But for all $m \geq n$, the resolvent between

$$t(i(s_1, i(s_2, \ldots, i(s_n, w) \cdots))) \tag{45}$$

and

$$t(i(v_1, i(v_2, \ldots, i(v_m, y) \cdots))) \wedge t(y) \wedge t(i(y, z)) \Rightarrow t(z) \tag{46}$$

is always some variable renamed copy of

$$t(y) \wedge t(i(y, z)) \Rightarrow t(z). \tag{47}$$

We consider these variable renamed copies as one single clause, and it is in this sense that

- $S_\Upsilon(C)$ is finite for any clause C.
- S_Υ characterises the transformation Υ as defined by (**??**).

The Test Substitution Set Condition

Now that we have a candidate for a resolution K-transformation we must check the test substitution set condition.

At this point we will drop the assumption that we only transfrom unit clauses! The test substitution set condition allows no such restriction.

First we must compute the test substitution set. We have

$$C = t(w) \wedge t(i(w, x)) \Rightarrow t(x) \tag{48}$$

and

$$L_\Upsilon = \{t(y), t(i(v_1, y)), t(i(v_1, i(v_2, y))), t(i(v_1, i(v_2, i(v_3, y)))), \ldots\}. \tag{49}$$

From this we get

$$\begin{aligned}
\Sigma_1 &= \{w \mapsto y, w \mapsto i(v_1, y), w \mapsto i(v_1, i(v_2, y)), w \mapsto i(v_1, i(v_2, i(v_3, y))), \ldots\} \\
\Sigma_2 &= \Sigma_1 \circ \\
&\quad \{x \mapsto y', x \mapsto i(v_1', y'), x \mapsto i(v_1', i(v_2', y')), x \mapsto i(v_1', i(v_2', i(v_3', y'))), \ldots\} \\
\Sigma_3 &= \Sigma_2 \\
\Sigma &= \Sigma_3
\end{aligned} \tag{50}$$

For a test substitution $\sigma \in \Sigma$, $C\sigma$ will have the form

$$\begin{aligned}
&t(i(v_1, i(v_2, \ldots, i(v_n, y) \cdots))) \wedge \\
&t(i(i(v_1, i(v_2, \ldots, i(v_n, y) \cdots)), i(v_1', i(v_2', \ldots, i(v_m', y') \cdots))) \\
&\Rightarrow t(i(v_1', i(v_2', \ldots, i(v_m', y') \cdots)))
\end{aligned} \tag{51}$$

Now we must replace all variables by constants. Then we have to check whether

$$
\begin{array}{ll}
\Big(t(i(a_1,i(a_2,\ldots i(a_n,a_{n+1})\cdots)))\wedge & (1)\\
\forall y.(t(i(a_1,i(a_2,\ldots i(a_n,a_{n+1})\cdots)))\wedge & (2)\\
\quad t(i(i(a_1,i(a_2,\ldots i(a_n,a_{n+1})\cdots)),y))\Rightarrow t(y))\wedge & \\
\forall y.(t(i(a_2,i(a_3,\ldots i(a_n,a_{n+1})\cdots)))\wedge & (3)\\
\quad t(i(i(a_2,i(a_3,\ldots i(a_n,a_{n+1})\cdots)),y))\Rightarrow t(y))\wedge & \\
\qquad\qquad\qquad\vdots\quad\vdots & \\
\forall y.(t(i(a_n,a_{n+1}))\wedge t(i(i(a_n,a_{n+1}),y))\Rightarrow t(y))\wedge & (n+1)\\
\forall y.(t(a_{n+1})\wedge t(i(a_{n+1},y))\Rightarrow t(y))\Big) & (n+2)\\
\bigwedge & \\
\Big(t(i(i(a_1,i(a_2,\ldots i(a_n,a_{n+1})\cdots)),i(b_1,i(b_2,\ldots i(b_m,b_{m+1})\cdots))))\wedge & (n+3)\\
\forall y.(t(i(i(a_1,i(a_2,\ldots i(a_n,a_{n+1})\cdots)),i(b_1,i(b_2,\ldots i(b_m,b_{m+1})\cdots))))\wedge & (n+4)\\
\quad t(i(i(i(a_1,\ldots i(a_n,a_{n+1})\cdots),i(b_1,\ldots i(b_m,b_{m+1})\cdots)),y))\Rightarrow t(y))\wedge & \\
\forall y.(t(i(b_1,i(b_2,\ldots i(b_m,b_{m+1})\cdots)))\wedge & (n+5)\\
\quad t(i(i(b_1,i(b_2,\ldots i(b_m,b_{m+1})\cdots)),y))\Rightarrow t(y))\wedge & \\
\forall y.(t(i(b_2,i(b_3,\ldots i(b_m,b_{m+1})\cdots)))\wedge & (n+6)\\
\quad t(i(i(b_2,i(b_3,\ldots i(b_m,b_{m+1})\cdots)),y))\Rightarrow t(y))\wedge & \\
\qquad\qquad\qquad\vdots\quad\vdots & \\
\forall y.(t(i(b_m,b_{m+1}))\wedge t(i(i(b_m,b_{m+1}),y))\Rightarrow t(y))\wedge & (n+m+4)\\
\forall y.(t(b_{m+1})\wedge t(i(b_{m+1},y))\Rightarrow t(y))\Big) & (n+m+5)\\
\Longrightarrow & \\
\Big(t(i(b_1,i(b_2,\ldots i(b_m,b_{m+1})\cdots)))\wedge & (n+m+6)\\
\forall y.(t(i(b_1,i(b_2,\ldots i(b_m,b_{m+1})\cdots)))\wedge & (n+m+7)\\
\quad t(i(i(b_1,i(b_2,\ldots i(b_m,b_{m+1})\cdots)),y))\Rightarrow t(y))\wedge & \\
\forall y.(t(i(b_2,i(b_3,\ldots i(b_m,b_{m+1})\cdots)))\wedge & (n+m+8)\\
\quad t(i(i(b_2,i(b_3,\ldots i(b_m,b_{m+1})\cdots)),y))\Rightarrow t(y))\wedge & \\
\qquad\qquad\qquad\vdots\quad\vdots & \\
\forall y.(t(i(b_m,b_{m+1}))\wedge t(i(i(b_m,b_{m+1}),y))\Rightarrow t(y))\wedge & (n+2m+6)\\
\forall y.(t(b_{m+1})\wedge t(i(b_{m+1},y))\Rightarrow t(y))\Big) & (n+2m+7)
\end{array}
$$

Figure 3.2: The test substitution set condition for the first transformer

$$
\begin{array}{l}
\Upsilon(t(i(a_1,i(a_2,\ldots,i(a_n,a_{n+1})\cdots))))\wedge\\
\Upsilon(t(i(i(a_1,i(a_2,\ldots,i(a_n,a_{n+1})\cdots))),i(b_1,i(b_2,\ldots,i(b_m,b_{m+1})\cdots))))\\
\Rightarrow \Upsilon(t(i(b_1,i(b_2,\ldots,i(b_n,b_{m+1})\cdots))))
\end{array}
\tag{52}
$$

is a tautology. This is the monstrous formula of Fig.**??**.

Line $(n+m+6)$ can be derived from (1),(2), and $(n+3)$. The lines $(n+m+7)$ through $(n+2m+7)$ can be derived because they are identical to lines $(n+5)$ through $(n+m+5)$. This is also true if either n or m are zero. Thus the test substitution set condition holds.

Corollary 3.6.1 The clause set S_Υ as defined by (**??**) characterises a resolution K-transformation for the condensed detachment clause.

So we have found a sound and complete transformer for the condensed detachment clause, and in spite of the fact that S_Υ is infinite, Υ will only add a *finite* number of clauses when transforming a clause set. Of course the transformation could be implemented more elegantly if it could be described in terms of a finite clause set, but at least it is clear that it *can* be implemented. We will not need to argue about the structure of some infinite result, or represent it in some finite way. The result is finite!

Disadvantage of This Transformer

Unfortunately, there is a serious problem with this transformer. We argued on page **??** that even if a positive literal unifies with infinitely many prefix literals in S_Υ, the resolvent would always be the same, namely (**??**). But (**??**) is nothing else but the condensed detachment clause! This means that if one of the clauses in the clause set we want to transform has the form $t(i(s_1, i(s_2, \ldots, i(s_n, w)\cdots)))$, we gain nothing by the transformation.

This insight is not completely new, however. Let us reconsider the euclideanness clause, for example. If we transform some unit clause that has a variable as second argument, say $R(a, x)$, the transformation will add the euclideanness clause itself, which is what we wanted to eliminate. This is true whether we take the transformer (**??**) or the transformer (**??**).

Intuitively one might say that transforming a very general clause will not yield a very specific result.

In the context of the condensed detachment clause, transforming unit clauses of the form $t(i(s_1, i(s_2, \ldots, i(s_n, w)\cdots)))$ is something we will frequently have to do. Typically we want to show that some propositional logic formula is a tautology by using some *axiom schemes* together with the condensed detachment clause. As we already mentioned, the terms of our meta logic (predicate logic) represent formulæ in our object logic (propositional logic). Axiom *scheme* means nothing else but that such a term may contain (predicate logic) variables that stand for some propositional formulæ.

A Slightly Better Transformer

There is one other thing we have not tried out yet. On page **??** it is said that we can either add instances of the condensed detachment clause where $t(w)$ is instantiated, or where $t(i(w, x))$ is instantiated. We have seen the former, but not the latter.

The transformation is characterised by

$$\begin{aligned} S_\Upsilon \quad := \quad & \{t(i(y,z)) \wedge t(y) \wedge t(i(y,z)) \Rightarrow t(z), \\ & t(i(v_1, i(y,z))) \wedge t(y) \wedge t(i(y,z)) \Rightarrow t(z), \\ & t(i(v_1, i(v_2, i(y,z)))) \wedge t(y) \wedge t(i(y,z)) \Rightarrow t(z), \\ & \ldots\} \end{aligned} \tag{53}$$

Compared to (**??**), $t(y)$ is replaced by $t(i(y, z))$ in the prefix literals. The transformation characterised by S_Υ meets the test substitution set condition, too, but we refrain from proving this. The proof is much like for the first transformation.

If we transform a unit clause $t(i(s_1, i(s_2, \ldots i(s_{n-1}, s_n)\cdots)))$ with this transformer, we will get n clauses. The first transformer generated $n+1$ clauses. Technically this is for the following reason: In both cases the prefix literals form an infinite chain of terms $t(i(x_1, i(x_2, \ldots i(x_{i-1}, s_i)\cdots)))$, but the second transformer starts this chain at a later point; it has no prefix literal $t(x)$.

There is another way to understand this. The first transformer will generate an instance

$$t(i(s_1, i(s_2, \ldots i(s_{n-1}, s_n) \cdots))) \wedge t(i(i(s_1, i(s_2, \ldots i(s_{n-1}, s_n) \cdots)), y)) \Rightarrow t(y) \quad (54)$$

even if the nesting depth of all other clauses rules out that a clause unifiable with $t(i(i(s_1, i(s_2, \ldots i(s_{n-1}, s_n) \cdots)), y))$ will ever be derived. In this sense, the second transformer is most likely superior to the first.

There is a second advantage. In order for this transformer to generate the condensed detachment clause itself, in which case it would turn useless, it is necessary that there is a unit clause $t(i(x_1, i(x_2, \ldots i(s_n, i(w_1, w_2)) \cdots)))$ among the clauses to be transformed (w_1 and w_2 are variables). With the first transformer, it was sufficient that the *last* term was a variable in order for this worst case to happen. If w_1 is a term other than a variable, the second transformer will *not* generate the condensed detachment clause, whereas the first will.

Thus we still have the problem that our transformer might be pathological in the sense that it does not eliminate the clause it is supposed to eliminate, but we reduced the cases in which this will happen.

As far as the condensed detachment clause is concerned, it must be admitted that the second transformer is as pathological as the first. However, the general idea might work for other clauses that generate new terms by resolution, and we have worked out that the choice of the selected literal may be of importance.

3.7 Finding Transformations for Clause *Sets*

It has been said on page **??** that rather than eliminating one clause C, we can also eliminate a clause set $\mathcal{C}$. In this case S_Υ should contain clauses in $\mathcal{C}$ plus some of their self resolvents, resolvents, and subsumed clauses[6].

We shall now become more specific about finding a clause set S_Υ that characterises a transformation for a clause set $\mathcal{C}$.

Recall that when we looked for a transformer for one clause C, we considered a set $\mathcal{S}_\Upsilon$ (see page **??**). Let us rather write $\mathcal{S}_C$ here, to emphasise that this set depends on C. This set contains clauses that *might* go into S_Υ. That is, S_Υ is a subset of $\mathcal{S}_C$. One possibility is that $\mathcal{S}_C$ is the set of self resolvents of C (as proposed in [**?**]), or else it may consist of prefixed clauses.

Suppose we have decided on the way to construct this candidate set $\mathcal{S}_C$. We shall now describe how we could enumerate clause sets S_Υ that are likely to characterise a resolution K-transformation for a clause set $\mathcal{C}$.

So we have a clause set $\mathcal{C} := \{C_1, \ldots, C_n\}$ that we want to eliminate.

1. For each C_i, construct the set $\mathcal{S}_{C_i}$ that contains clauses that might characterise a transformation for C_i. Thus we have a set $\{\mathcal{S}_{C_1}, \ldots, \mathcal{S}_{C_n}\}$.

2. Since each $\mathcal{S}_{C_i}$ contains clauses that might characterise a transformation for C_i, it is reasonable to assume that a clause set S_Υ that characterises a transformation for $\mathcal{C}$ should contain clauses from the $\mathcal{S}_{C_i}$.

   Therefore the next step is to generate (small) subsets S_i of each $\mathcal{S}_{C_i}$. This gives us a set $S_1, \ldots, S_n$. No S_j should be the empty set, because this would mean that the clause C_j (which is in $\mathcal{C}$) has not contributed to the construction of S_Υ at all.

3. Now we might generate a few resolvents between the clauses from the $\mathcal{S}_{C_i}$. For each resolution one partner should be in some S_i and the other partner in some S_j, where $i \neq j$.

[6]Actually it is not necessary that S_Υ contains all clauses in $\mathcal{C}$. We shall see an example where S_Υ does not contain all clauses in $\mathcal{C}$.

4. Take the union of all S_i together with the generated resolvents.

5. For each clause in this set, select a negative literal. The resulting clause set (where the selected literals are marked) is a candidate for S_Υ.

A clause set S_Υ generated in the described way indeed consists of "clauses in $\mathcal{C}$ plus some of their self resolvents, resolvents, and subsumed clauses."

The number of candidates that can be generated in this way is astronomic, and still this procedure does not enumerate all S_Υ that consist of "clauses in $\mathcal{C}$ plus some of their self resolvents, resolvents, and subsumed clauses."

Actually I have restricted the search described above even further. The program TRANSFORMATOR restricts the number of resolvents that may be added (see point **??**) to $\binom{n}{2}$. The idea is to allow one resolution between each pair (S_i, S_j).

We shall see, however, that the practical results are not too bad. I have implemented this search, and the transformers presented in the following examples were found by my program TRANSFORMATOR.

Example 3.7.1 The first example is to illustrate that the search space is gigantic.

Consider

$$\begin{aligned} C_1 &:= R(x,y) \wedge R(y,z) \Rightarrow R(x,z) \\ C_2 &:= R(x,y) \wedge R(x,z) \Rightarrow R(y,z) \\ \mathcal{C} &:= \{C_1,\ C_2\} \end{aligned} \tag{55}$$

Suppose we try to find a transformation that is characterised by self resolvents. We shall only take self resolvents up to level 1 into account.

The first clause has one direct self resolvent, the second clause has two. Together with the original clauses we get that $\mathcal{S}_{C_1}$ has 2 elements, and $\mathcal{S}_{C_2}$ has 3 elements.

There are $2^2 - 1 = 3$ subsets of $\mathcal{S}_{C_1}$ of at least one element; likewise, there are $2^3 - 1 = 7$ subsets of $\mathcal{S}_{C_2}$. Thus there are $3 * 7 = 21$ ways to generate $\{S_1, S_2\}$.

Now suppose that for a given $\{S_1, S_2\}$, we allow to add one resolvent. Only one! There are at most 2 clauses in S_1, and at most 3 clauses in S_2. Thus there are at most $2 * 3 = 6$ ways to choose the two resolution partners.

Assuming that we have chosen two resolution partners, each partner has at most 3 negative literals and exactly one positive literal, resulting in at most $3 + 3 = 6$ possibilities for resolution.

S_Υ consists of the union of S_1 and S_2, and possibly the added resolvent. S_1 and S_2 together have at most 5 elements. The added resolvent has 5 negative literals, whereas the original clauses C_1 and C_2 have two negative literals, and their self resolvents have 3 negative literals. Recall that for each clause in S_Υ, a negative literal must be selected. There are at most $5 * 2^2 * 3^3 = 540$ ways to do so.

Now of course, to get an upper bound for the number of candidates for S_Υ under these restricted conditions, we must multiply the numbers of possibilities at each stage, that is

$$21 * 6 * 6 * 540 = 408240 \tag{56}$$

Our bounds were very rough. Actually TRANSFORMATOR comes up with a number of 40289 candidates for S_Υ. This is still terrible enough.

It should be clear that one has to be very careful about the order in which the elements of a set of this size are inspected. The above computation of an upper bound shows that there are several choice points. If we represent each choice point as an edge and the choices as arcs of a tree, we could say that each of the 40289 candidates is a leaf in that tree . TRANSFORMATOR inspects this tree in a way that resembles breadth-first-search.

$\triangleleft$

Example 3.7.2 In spite of the exploding size of the search space, we can often find transformations for several clauses. Let us see how transitivity and symmetry can be removed at the same time.

$$\begin{aligned}
\mathcal{C} &:= \{R(x,y) \wedge R(y,z) \Rightarrow R(x,z), \\
&\quad\; R(x,y) \Rightarrow R(y,x)\} \\
S_\Upsilon &= \{\boldsymbol{R(x,y)} \wedge R(y,z) \Rightarrow R(x,z), \\
&\quad\; \boldsymbol{R(x,y)} \Rightarrow R(y,x), \\
&\quad\; \boldsymbol{R(y,x)} \wedge R(y,z) \Rightarrow R(x,z)\}
\end{aligned} \tag{57}$$

The first two clauses in S_Υ are the original clauses, the third is obtained by resolving transitivity and symmetry. Coincidentally the resulting clause is the euclideanness clause.

This transformation is presented in [?], but was also easily found by Transformator. It took about 0.25 seconds to find this transformer.

$\triangleleft$

Example 3.7.3 We shall now give an example that illustrates that our search for a transformation is not complete. Consider

$$\begin{aligned}
C_1 &:= R(x,y) \Rightarrow R(y,x) \\
C_2 &:= R(x,y) \wedge R(y,z) \Rightarrow R(z,x) \\
\mathcal{C} &:= \{C_1,\ C_2\}
\end{aligned} \tag{58}$$

C_1 is symmetry, C_2 is what we called "circularity". Transformator found the following transformation:

$$\begin{aligned}
S_\Upsilon = \{&\boldsymbol{R(x,y)} \Rightarrow R(y,x), \\
&\boldsymbol{R(x,w)} \wedge R(x,y) \wedge R(y,z) \Rightarrow R(z,x), \\
&\boldsymbol{R(w,x)} \wedge R(x,y) \wedge R(y,z) \Rightarrow R(z,x)\}
\end{aligned} \tag{59}$$

Recall that C_1 and C_2 occur in Table **??**. C_1 is no. 7, C_2 is no. 12. If we look at the transformations presented there, we see that we must simply take the union of the two clause sets that represent transformations for C_1 and C_2, respectively. This union is the set S_Υ that characterises a transformation for both clauses at the same time.

Actually Transformator found S_Υ in another way. It only considered *one* prefixed clause for C_2. The other one was added because, coincidentally, it is a resolvent of the prefixed clause and the symmetry clause.

Now there is an interesting point about C_1 and C_2. By one resolution step, we can obtain the transitivity clause. On the other hand, if we resolve transitivity and symmetry, we can obtain C_2. Therefore we would expect that the transformation for symmetry and transitivity together of Example **??** is also a transformation for $\mathcal{C}$. This is indeed the case.

Transformator cannot find this transformation, however. The reason is that the S_Υ of Example **??** consists of C_1 plus *two* resolvents between C_1 and C_2. One of these happens to be the transitivity clause, the other one happens to be the euclideanness clause. Transformator allows only $\binom{2}{1}$, that is, *one* resolvent between C_1 and C_2 to be added to S_Υ.

Of course we could have omitted this restriction, but then again the search space would have been even larger. Improving this search for arbitrary sets of clauses would certainly require a thorough theoretic analysis aimed to reduce the search space.

$\lhd$

Example 3.7.4 Our last example is a transformation for euclideanness and "circularity".

$$\begin{aligned}
\mathcal{C} \quad &:= \quad \{R(x,y) \wedge R(y,z) \Rightarrow R(z,x), \\
& \qquad R(x,y) \wedge R(x,z) \Rightarrow R(y,z)\} \\
S_\Upsilon \quad &= \quad \{\boldsymbol{R(x,y)} \wedge R(y,z) \Rightarrow R(z,x), \\
& \qquad \boldsymbol{R(w,z)} \wedge R(x,y) \wedge R(x,z) \Rightarrow R(y,z), \\
& \qquad \boldsymbol{R(z,v)} \wedge R(v,w) \wedge R(x,y) \wedge R(x,z) \Rightarrow R(y,z)\}
\end{aligned} \tag{60}$$

$\lhd$

3.8 Complexity

In this chapter we have described how a resolution K-transformation can be found. I have implemented these ideas, which will be the subject of the next chapter. My program is called TRANSFORMATOR and it can be used both for *finding* a transformation and for *using* it.

In this section we shall investigate a few aspects of the complexity of *finding* a transformation and *using* a transformation. This is done here rather than in the next chapter because our analysis is essentially a theoretical one. We shall make statements about the size of the sets that have to be searched for a transformation. This means that the statements have little if anything to do with the actual implementation of TRANSFORMATOR.

We shall argue that other aspects of complexity, namely those concerning the implementation of TRANSFORMATOR, are negligible.

We must make clear which aspects of complexity we want to investigate. Possible questions would be:

- How complex is it to *find* a resolution K-transformation for a clause of a given size? This could be divided into different parts such as: How complex is it to compute self resolvents, to compute prefixed clauses, to compute Υ for the test substitution candidates etc. Finally, how complex is it to prove the test substitution set condition?

- How complex is it to *transform* a clause set of a given size using a transformation characterised by a certain S_Υ?

3.8.1 Finding a Transformation

Actually we shall neglect most of the questions listed above, and the justification for this is an empirical one. It results from the experience with my Prolog program TRANSFORMATOR, which will be described in Chapter **??**.

I have conducted several profiling experiments. A *profile* of a Prolog program gives information about how often a predicate was called during execution, how much time was used up in the calls etc. From these experiments it became absolutely clear that in the process of finding a resolution K-transformation, almost all of the CPU time is used up in the attempts to prove the test substitution set condition.

Less than two percent of the CPU time is used up in predicates not directly related to the proofs. Considering how complex automated theorem proving is it seems likely that for more complex clauses, this result should be even clearer. In other words, the length of the proof attempts grows worse than anything else.

It is not very meaningful to attempt to express this more precisely. After all, how much time is used in the proofs depends on certain parameters determined by the user. The user can set these such that most of the time is used for other things than for proving. But then he will probably never find a transformer! This will become clear in Chapter **??**.

So as far as *finding* a transformation for a clause C is concerned, almost everything depends on three aspects:

1. How many candidates for S_Υ have to be tested?

2. How many test substitutions are there?

3. How complex is a candidate for S_Υ, and, as a consequence, how complex is formula (**??**)?

We shall now look at these questions.

How many candidates for S_Υ have to be tested?

Let us consider a clause

$$C := R(x_{11},\ldots,x_{1n}) \vee R(x_{21},\ldots,x_{2n}) \vee \ldots \vee R(x_{m1},\ldots,x_{mn}) \qquad (61)$$

We have neglected here that some of the atoms may be negated because it is of no importance for our analysis. So we consider a clause of m literals. There is only one predicate symbol, which has arity n. If we had several predicate symbols, we would simply have to consider them separately.

Some of the variables may be identical, of course.

Assume we want to find a resolution K-transformation for C by investigating prefixed clauses. We have seen in Section **??** that there are two ways of doing this. We shall now consider the more general way.

Corollary 3.8.1 Suppose we consider the following prefix literals: A prefix literal is a literal that has R as predicate symbol, and some arguments of the literal are instantiated with variables occurring in C.

Let V be the number of variables occurring in C. Then there are at most $(V+1)^n$ prefix literals, which is less or equal $(m*n+1)^n$.

Proof:

This is a simple combinatoric argument. We take a fresh instance of a literal with R as functor. Now we can choose the number j of arguments we want to instantiate ($j < n$). Once we have chosen j, we have to choose *which* j arguments to instantiate. There are n arguments, so there are actually $\binom{n}{j}$ possibilities for this choice. Once we have chosen *which* arguments we instantiate, we must for each argument choose a variable from C used in this instantiation. There are V variables, so there are V^j possibilities to do so[7].

Taking the sum for all choices of j we get

$$\sum_{j=0}^{n} \binom{n}{j} V^j 1^{n-j} = (V+1)^n \qquad (62)$$

[7]If we allow double occurrences of variables, that is. Otherwise we would get $\frac{V!}{(V-j)!}$ possibilities, and V^j would be an upper limit for this. Actually TRANSFORMATOR does not consider double occurrences of variables

The factor 1^{n-j} was added so it could be seen easily that the term on the left side of the equation is simply the binomial expansion of the term on the right side.

All in all, C has $n*m$ arguments, so V can be $n*m$ at most. Thus we get

$$(V+1)^n \leq (m*n+1)^n \tag{63}$$

□

Now in contrast suppose we restrict the number of prefix literals as it has been proposed in Section **??**.

Corollary 3.8.2 Suppose we consider the following prefix literals: A prefix literal is found by taking a fresh instance of an atom with R as functor and partially unifying this atom with a literal occurring in C.

Then there are at most $m*2^n$ prefix literals.

Proof:
So we have a fresh instance of an atom with R as functor. Now we can choose from m literals of C that we may use for partial instantiation. Once we have done this choice, we may choose for each of the n arguments whether or not to instantiate it. There are 2^n possibilities to do so. Thus we have at most

$$m*2^n \tag{64}$$

prefix literals.

□

Both of the above corollaries only give rough upper bounds for the number of prefix literals. First, we neglected that we exclude prefix literals that are equal to a positive literal occurring in C (see Section **??**).

Furthermore, in the case of Corollary **??**, the number of variables V will usually be significantly smaller than $m*n$. For many prominent clauses like transitivity, euclideanness, and the permutation clauses V actually equals $\frac{m*n}{2}$. We can also reduce the number of prefix literals by excluding double occurrences of a variable in a prefix literal.

In the case of Corollary **??**, there is another reason why the upper bound is rough. Many of the prefix literals in consideration actually turn out to be equal. Take the euclideanness clause (**??**), for example. If we take the fresh atom $R(v,w)$ and unify it with $R(x,y)$ in the first argument, we will get $R(x,w)$. If we unify $R(v,w)$ with $R(x,z)$ in the first argument, we will also get $R(x,w)$. The upper bound of $m*2^n$ given in Corollary **??** assumes that this prefix literal is enumerated twice.

Anyway, one point should be clear: If n is greater than 2, $(m*n+1)^n$ grows much faster than $m*2^n$. Since my experience suggests that we do not fail to find transformers by restricting the prefix literals in the proposed way, it is strongly recommended to do so.

Example 3.8.3 We give a little example to get a feeling for the upper bounds and the actual number of prefix literals. Consider

$$C := R(w,x,y) \land R(x,y,z) \Rightarrow R(w,x,z). \tag{65}$$

We have already seen this clause in Section **??**. Here we have $n=3$, $m=3$, and $V=4$. According to Corollary **??**, we have at most $(4+1)^3$ prefixed clauses, which is 125. The actual number of prefixed clauses is 71.

If we impose the mentioned restriction on prefix literals, Corollary **??** says there will be at most $3*2^3$ prefixed clauses, which is 24. The actual number of prefixed clauses is 16.

$\triangleleft$

So far we have only considered how many prefix literals there are. This is, we have said how many members the set $\mathcal{S}_\Upsilon$ has (see page **??**). A candidate for S_Υ is a subset of this set. Thus there are actually $2^{|\mathcal{S}_\Upsilon|}$ candidates for S_Υ.

This is not as bad as it sounds, however. We shall see that the search for a transformer is done such that candidates for S_Υ that are singletons are preferred and investigated first. Often Υ can be characterised by a singleton.

How Many Test Substitutions are there?

For the clauses we investigated, there was usually only one test substitution. Corollary **??** explains why this was so. When we look at clauses not containing function symbols, the only case where we have to consider several test substitutions is that the prefix literal contains the same variable more than once.

In principle the number of test substitutions can be enormous.

Corollary 3.8.4 Consider the clause $C := A_1 \wedge \ldots \wedge A_n \Rightarrow A_{n+1} \vee \ldots \vee A_m$. Let $l =| L_\Upsilon |$ (recall that L_Υ is the set of characteristic literals, see page **??**). Then there are at most

$$(2^l)^m \tag{66}$$

test substitutions.

Proof:

Have a look at equation (**??**).

We give an inductive argument. There is one substitution in Σ_0, and $1 = (2^l)^0$.

Now suppose that Σ_i contains at most $(2^l)^i$ substitutions. For each $\sigma \in \Sigma_i$, the atom $A_{i+1}\sigma$ can simultaneously be unified with any subset $\mathcal{K}$ of L_Υ. There are 2^l such subsets, so that Σ_{i+1} contains at most $(2^l)^i * (2^l)$ substitutions.

Σ is defined as Σ_m. Thus Σ contains at most $(2^l)^m$ substitutions.

$\square$

Again things are not as bad as it seems at first. Often L_Υ is a singleton, and some of the test substitutions are unnecessary because they result in tautologies, which are of no use in S_Υ. Furthermore, some of the resulting clauses may be equivalent.

Example 3.8.5 Consider

$$C := R(x,x) \wedge R(x,y) \Rightarrow R(y,z) \tag{67}$$

This is clause no.1 of Table **??**. Corollary **??** gives us an upper bound of $(2^1)^3 = 9$ for the number of test substitutions. Actually there are only three relevant test substitutions in this example.

$\triangleleft$

How Complex is the Test Substitution Set Condition?

To compute (**??**), Υ is computed separately for each literal of C. Thus it is obvious that the size of (**??**) depends on the length of C in a linear way.

Now for the transformation of each literal A_i. A_i is transformed by resolving it with one of the literals in S_Υ, or with none of them. Thus $\Upsilon(A_i)$ contains $| S_\Upsilon | + 1$ clauses.

We can not say in general how many clauses S_Υ contains for an arbitrary clause C, but we can say that the size of formula **??** depends on the cardinality of S_Υ in a linear way.

We could not have expected any better result than this, but unfortunately, it is bad enough, considering that the complexity of proving a formula is exponential.

3.8.2 The Complexity of the Transformation Itself

We shall now turn to a completely different question. Once we have found a transformer, how complex is it to use it?

The last section has given us a hint to this question. For one thing, a clause set is transformed by transforming each clause in this clause set separately. That is, there is no dependency between the different clauses in this set. Thus we get an obvious and very important result:

> The time complexity of transforming a clause set Ψ is linear in the number of clauses of Ψ.

Another question is: How complex is it to transform a single clause D? Recall Def. **??**. To generate a simultaneous resolvent, each positive literal of D is resolved with one clause in S_Υ, or not resolved at all. Thus for each positive literal of D, there are $|S_\Upsilon| + 1$ choices.

Corollary 3.8.6 Let D be a clause and p be the number of positive literals of D. Let Υ be the transformation characterised by the clause set S_Υ. Then we have

$$|\Upsilon(D)| = (|S_\Upsilon| + 1)^p \tag{68}$$

Proof: $\Upsilon(D)$ is defined as the union of all simultaneous resolvents between D and S_Υ. Each positive literal can be resolved with any clause in S_Υ, or not be resolved at all. This gives $(|S_\Upsilon| + 1)^p$ possibilities.

□

Conclusion

We have only touched a few aspects of the complexity of clause K-transformations. It has become clear that finding a transformation can be very complex in principle.

On the other hand, applying the transformation is harmless. There is a linear relation between the size of the original clause set and the size of the transformed clause set.

This suggests that for a useful clause K-transformation, the clause to be removed must be small relative to the clause set that is transformed. Clauses like transitivity and euclideanness are small by all means and finding transformations for them is easy.

Chapter 4

Implementation

In the previous chapters we have introduced different algorithms or heuristics, we might say, for finding resolution K-transformations. Now of course we would like to automate this process.

To be precise, we want to automate the following:

- Given a clause set S_Υ, we want to transform a clause set Ψ with the transformer that is characterised by S_Υ. Afterwards we can use the transformed clause set as input for a theorem prover.
- Given a clause set S_Υ and a clause C (or a clause set $\mathcal{C}$), we want to verify whether S_Υ characterises a resolution K-transformation for C (or $\mathcal{C}$). This involves automated theorem proving, and therefore no guarantee for termination can be given.
- Given a clause C (or a clause set $\mathcal{C}$), we want to find a transformer for C (or $\mathcal{C}$). This search requires checking the test substitution set condition for different transformations.

The tasks are listed in hierarchical order, so to speak. The second task includes the first, and the third task includes the second.

I have written a Prolog program that performs these tasks. The program is called TRANSFORMATOR. TRANSFORMATOR can be used on the Prolog interpreter level. If you want to find a transformer for some clause or just verify that a transformer is complete, it is convenient to use TRANSFORMATOR this way. Of course this requires some basic knowledge of Prolog.

Using TRANSFORMATOR on the interpreter level is certainly not convenient if you want to transform (large) clause sets. For this reason, I wrote a file interface. Thus TRANSFORMATOR can be called on the operating system level. The input file contains a clause set where the clause to be removed is marked in some way. The output file contains the transformed clause set. Using TRANSFORMATOR in this way requires no knowledge of Prolog.

I chose Prolog because we deal with logic formulæ (clauses etc.), and these can be expressed very naturally in Prolog term syntax. Furthermore, it lies in the nature of our problem that we deal with multiple solutions. For a clause C, there may be several transformers. Prolog is very suitable for enumerating these solutions.

Outline of this Chapter

Section **??** gives a very short overview of what TRANSFORMATOR computes.

The following sections describe in some detail *how* TRANSFORMATOR works. There are several options the user can set, and for understanding the options it is necessary to understand how TRANSFORMATOR works. On the other hand, to use TRANSFORMATOR, it is not necessary to understand all options that can be set. If an option is not set, (hopefully) reasonable default values are assumed.

The description of TRANSFORMATOR is intended to abstract from the actual programming language, Prolog. Algorithms are shown using the usual "pseudo-Pascal"-notation. Therefore we shall not speak of *predicates*, but rather of *functions*, although this is not technically correct. A reader not famliliar with logic programming might not know what a predicate is, but she would certainly have an idea of what a function is.

There is a problem about this, however. Prolog predicates are ideal for enumerating *multiple* solutions. On each subsequent call, a predicate is backtracked and the next solution is computed. This does not readily translate into the concept of a function. In these cases I have not translated the predicate into a function. We use the more general notion of an algorithm then, and rather than *returning* a result, we shall simply say that we *output* the result. The algorithm may then continue and compute the next solution.

The algorithm descriptions are simple enough so this should not be difficult to understand.

Sections **??** and **??** give an exhaustive and technically precise description of how TRANSFORMATOR is *used*. But once again, it is not possible to understand in all detail how TRANSFORMATOR is used without understanding how it works.

For the reader who does not want to go into detail, it is recommended to read Section **??**, skip over Sections **??** through **??** and continue with Section **??** or **??**. These last sections contain references to the previous sections so that it is easy to go into more detail as it is desired.

Typographic Conventions

For text that occurs literally in TRANSFORMATOR, for text that is typed to the terminal, and for filenames we use `typewriter font`.

On the meta level, that is to denote a predicate or other object in TRANSFORMATOR in an abstract way, we use *italic font*. For example we write "*option* is called" if *option* stands for *some* option.

We use sans serif font for function names. Functions do not appear in the actual code, but only in the "pseudo-Pascal" algorithm descriptions.

Variable names occurring in these algorithm descriptions are written in *slanted font*.

4.1 Introduction to TRANSFORMATOR

We shall now look at the functions TRANSFORMATOR computes. That is, only the first three are functions. The fourth, find_upsilon is an algorithm that, in principle, never stops. It may compute more than one result, even infinitely many, but it may also compute no result at all.

In the actual Prolog program, find_upsilon is a predicate that enumerates multiple solutions on backtracking. This means of course that a solution that is found can be used before other solutions are produced.

Transforming Clauses: upsilon

upsilon(D,S_Υ) computes $\Upsilon(D)$. Thus it returns a set of clauses. Here D is a clause and S_Υ is a set of clauses that characterises the transformation Υ.

Transforming Clause Sets: upsilon_list

While upsilon is used to transform one clause, upsilon_list is used to transform *sets* of clauses. upsilon_list($\mathcal{D}$,S_Υ) calls upsilon for all clauses in $\mathcal{D}$ and returns the union of all the results.

Verifying the Test Substitution Set Condition: tss_condition

Since first order logic is not decidable, in general we may not expect that any proof procedure terminates. When we call tss_condition(C,S_Υ,*time*), a theorem prover will be given *time* milliseconds to prove that S_Υ characterises a resolution K-transformation for C. if tss_condition returns true, we know that S_Υ characterises a resolution K-transformation for C. **If it returns false, this does not imply that S_Υ does not characterise a resolution K-transformation for C.** It only means that this could not be proven in the given time.

tss_condition also works for clause *sets* $\mathcal{C}$ rather than a single clause C (see Section **??**).

Find a transformer: find_upsilon

find_upsilon can not be modeled as a function in a reasonable way, because it may compute anything from zero to infinitely many solutions. These solutions are "output", whatever that means. A solution is a clause set S_Υ that characterises a resolution K-transformation for the clause C.

In the previous chapter we have introduced two different ways of finding candidate clauses for S_Υ: either we create self resolvents, or we create prefixed clauses. Transformator allows the user to choose between the two ways. In either case the search space will be enlarged in steps or *iterations*, we might say. During the first iteration, the test whether some S_Υ characterises a resolution K-transformation is limited to a certain time. In the next iteration, this limit will be multiplied by some factor. This technique guarantees completeness in the sense that for a clause a transformer of the kind we described will eventually be found, if it exists. We shall consider this enumeration process in more detail later.

The time limit and the factor can also be determined by the user, thus controlling the search in a certain way.

find_upsilon also works for clause sets $\mathcal{C}$ rather than a single clause C (see Section **??**). However we shall not describe in detail how the search for S_Υ is realised for this case.

4.2 Structure of Transformator

The central module is called `killer`. In `killer` all functions supplied to the outside are defined[1]. Let us understand the structure of Transformator by sketching what happens when find_upsilon(C) is called:

1. A candidate clause set $\mathcal{S}_\Upsilon$ (see page **??**) is computed. We hope that some subset of $\mathcal{S}_\Upsilon$ will be the S_Υ that characterises a resolution K-transformation. This computation depends on an integer L. Initially it is 1. "L" stands for "limit". The computation can be done in two ways:

   (a) Look for self resolvents. $\mathcal{S}_\Upsilon$ consists of all self resolvents that can be obtained from C in L resolution steps. *Every* non-empty subset of $\mathcal{S}_\Upsilon$ is

[1] If Transformator is used through the file interface, there is an additional module `interface` that translates the commands in the input file into calls to `killer`.

a candidate for S_Υ. The predicates that are related to the computation of these self resolvents can be found in the module `selfres`.

(b) Look for prefixed clauses. $\mathcal{S}_\Upsilon$ consists of all prefixed clauses of C. On page **??** it is explained that for clauses not containing function symbols, $\mathcal{S}_\Upsilon$ is finite. Here the number L takes a slightly different role. Every non-empty subset of $\mathcal{S}_\Upsilon$ having at most L elements is a candidate for S_Υ.

2. For each candidate S_Υ the test substitution set condition is checked. Test instances $C\,\sigma$ are created. `upsilon` is called to compute $\Upsilon(A_i\,\sigma_{gr})$ for each atom A_i of C. The results are composed so that the formula of equation (**??**) is generated. Then a theorem prover is called to check whether this formula is a tautology.

   If it can be proven within a certain time that the candidate meets the test substitution set condition, output it (and try to find the next solution if desired). If not, try the next candidate.

3. If no candidate could be proven to meet the test substitution set condition, go back to **??**, but increment L and multiply the time limit by some factor (which would have to be specified).

Fig.**??** gives an overview of the modules that form TRANSFORMATOR. An arrow means that there are predicate calls from the pointing module to the pointed module. Now that we understand the basic structure of TRANSFORMATOR, we shall explain some modules in more detail.

4.3 Finding Candidates for S_Υ: `selfres`

The module `selfres` contains the predicates that are related to computing self resolvents of a clause C. We have seen in the previous chapter that generating self resolvents is one way to find transformers.

We know that there may be infinitely many self resolvents. Therefore we must enumerate them systematically. There is a function that takes a clause C and a number as arguments and computes all self resolvents up to this level. This function is shown in Fig.**??**.

self_resolvents(C, *level*) *level* = 0 [[C]] $[L_{level-1},\ldots,L_0]$, := self_resolvents(C, *level* -1) L_{level} := all_resolvents($C, L_{level-1}$) $[L_{level}, L_{level-1},\ldots,L_0]$ fuselfcompComputation of self resolvents

The function all_resolvents computes the list of all resolvents between C and the clauses in $L_{level-1}$. Corollary **??** says that this is sufficient to compute all self resolvents up to level *level*.

self_resolvents returns a *list of lists* of clauses. Note that this is necessary for the recursive call of self_resolvents.

The list $[L_{level},\ldots,L_1]$ does not contain clauses that are tautologies. These are filtered out. From Corollary **??** it is clear that tautologies are of no use in S_Υ.

When we form self resolvents in order to find transformers, there are two simplifications that can be made:

- Subsumed clauses can be removed.
- Clauses can be shortened using factoring

We shall now explain these simplifications.

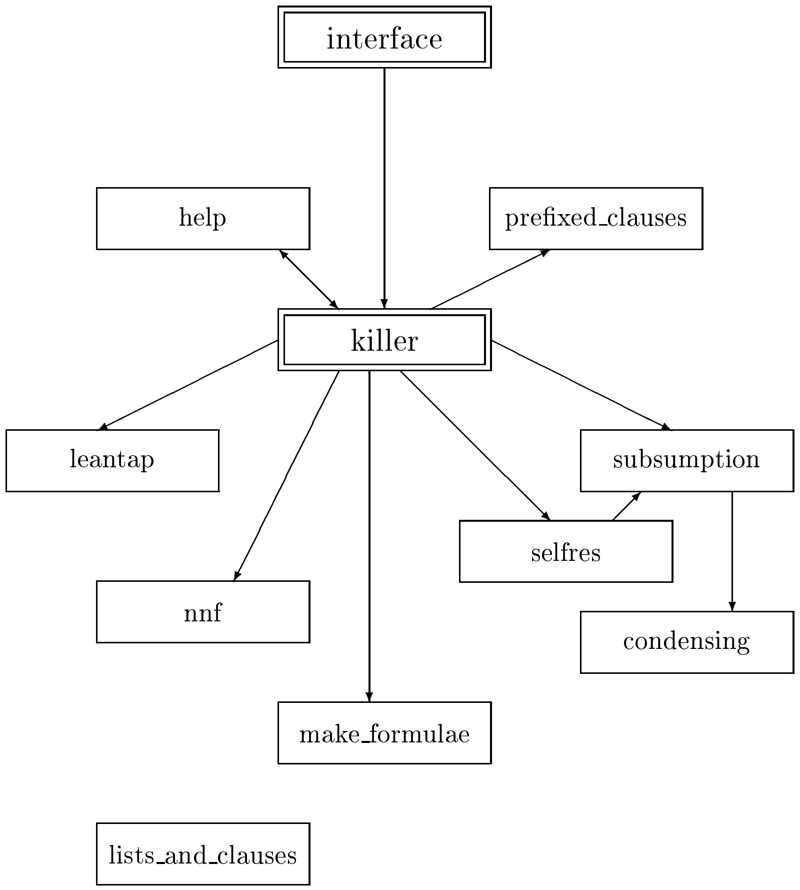

Figure 4.1: Overview of TRANSFORMATOR

4.3.1 Eliminating Subsumed Clauses: `subsumption`

The list of self resolvents may contain clauses that are subsumed by other clauses. Subsumption of clauses is defined on page **??**.

Consider the transitivity clause, for example. Resolution on the first negative literal will yield the same resolvent as resolution on the second literal. This is not the case with all clauses, however. I do not know how it could be decided looking at a clause C whether self resolution will result in identical clauses or not.

Therefore TRANSFORMATOR first generates all self resolvents, even if some of them happen to be identical.

The module `subsumption` contains predicates that have the task of removing such redundant clauses. Removing redundant clauses is expandable, of course. Therefore it is left as a choice to the user how much subsumption testing he wants to be performed.

The subsumption testing must be built into the process of generating self resolvents. Suppose we have already computed $[L_n, \ldots, L_0]$, that is, the self resolvents up to the nth level. Furthermore, we have computed the set of all resolvents of C with the clauses in L_n. Call this set L'_{n+1}. L'_{n+1} may contain duplicates and clauses that are subsumed by other clauses.

The user can choose between three levels of subsumption testing:

- No subsumption testing. L_{n+1} is L'_{n+1}. Thus the list of self resolvents computed in this iteration may contain duplicates.

- Subsumption testing within L'_{n+1}. L_{n+1} is instantiated to the list containing the members of L'_{n+1} that are not strictly subsumed by another clause in L'_{n+1}. If two clauses in L'_{n+1} are identical up to variable renaming, only one clause is taken into L_{n+1}. One would expect that proper subsumption cannot occur within self resolvents of the same level. It can occur however, if we modify each clause by *factoring* it. This will be explained later.

- "Global" subsumption testing. Clauses in L'_{n+1} are tested for subsumption by another clause in L'_{n+1} as well as for subsumption by a clause in a list in $[L_n, \ldots, L_1]$.

The second level is the default.

Now we turn our attention to another simplification used by `selfres`, called "factoring" or "condensing".

4.3.2 Shortening Clauses: `condensing`

"Condensing" or "factoring" is another simplification. It is not a simplification of a clause set, however, but of a single clause.

As an example consider the clause $P(x) \vee P(a)$. $P(a)$ implies $P(x) \vee P(a)$. Furthermore, $P(x)$ implies $P(a)$. Recall that x is considered as universally quantified. Therefore $P(x) \vee P(a)$ implies $P(a)$. Together this implies that $P(a)$ is equivalent to $P(x) \vee P(a)$.

Let us state this generally. Let C be a clause. If there are clauses C_1,C_2, and D and a substitution σ such that

$$\begin{aligned} C \quad &= C_1 \cup C_2 \cup D \\ C_1\,\sigma &= C_2 \\ C_2\,\sigma &= C_2 \qquad \text{and} \qquad D\,\sigma = D \end{aligned} \tag{69}$$

then $C_2 \cup D$ is equivalent to C. Therefore we can replace C with $C_2 \cup D$. We say that C is *condensed* to $C_2 \cup D$.

Condensing can be repeated, of course. The user can choose whether self resolvents should be condensed or not by calling one of the following predicates:

- Self resolvents are condensed as long as possible. This is the default.

- Self resolvents are not condensed.

Now that we have studied the module `selfres` and its children, we will turn to the second strategy for finding transformers, and to the module `prefixed_clauses`.

4.4 Finding Candidates for S_Υ: `prefixed_clauses`

In order to find for a clause set S_Υ that characterises a resolution K-transformation, I have proposed another technique than the one presented in [**?**]. I described this technique in Section **??**. All in all, this technique was simpler to implement than the technique of finding self resolvents.

When we transform clauses without function symbols, the set of prefix literals is finite. And it is also finite in a practical sense. In the whole process of finding S_Υ that characterises a resolution K-transformation, computing the candidates takes very little time! Practically all of the time is used up in the attempts to prove the test substitution set condition. Therefore it can be afforded to compute all candidates at the very beginning before any candidate is tested for (**??**).

The analogy of the function self_resolvents (see page **??**) is the function prefix_clauses. prefix_clauses(C) returns the set of all clauses composed of C plus a prefix literal. Each clause in this set has fresh variables!

There are two ways to find prefix literals (see page **??**):

- Take a fresh literal whose predicate occurs in the clause and partially instantiate this literal with variables occurring in the clause. All literals obtained in this way are candidates for a prefix literal.
- Take only literals that are generalisations of literals that actually occur in the clause. This is the default.

The way prefix literals are computed differs considerably depending on which prefix literals are taken into consideration.

4.4.1 All Prefixed Clauses are Considered

To compute a prefixed clause for a clause C, first we collect all variables of C. Then we take a fresh instance of a predicate symbol of C. Now we instantiate the arguments of this fresh literal with some variables of C. A literal generated in this way is a prefix literal. Attaching it as a negative literal to C gives us a prefixed clause.

`prefix_clauses` collects *all* prefixed clauses.

An important point is that this is done in such a way that in the list of all prefixed clauses, the prefixed clauses whose prefix literals share *many* variables with C occur *before* the prefixed clauses whose prefix literals share *few* variables with C.

Take $C = R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)$, for instance. $R(w,x,y) \wedge R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)$ will be enumerated before $R(w,x,v_1) \wedge R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)$, and the latter before $R(v_1,x,v_2) \wedge R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)$.

We have seen in Section **??** that the more variables the prefix literals share with the rest of the clause, the better the transformer is.

The search for a clause set S_Υ is done such that the clauses occurring early in the list of all prefixed clauses are preferred as candidates for S_Υ.

There are three other properties of `prefix_clauses` that reflect some of the points made in Section **??**

- The prefix literal must share at least one variable with the input clause. Otherwise our transformer is useless (see Section **??**).
- For example, $R(w,x,y) \wedge R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)$ is contracted to $R(w,x,y) \wedge R(x,y,z) \Rightarrow R(w,x,z)$. In general, if the prefix literal happens to be identical with a negative literal, this negative literal is removed. Technically, a clause containing the same literal twice will not be constructed in the first place. Conceptually, we should regard this as an optimisation (see Section **??**).
- A prefix literal must not equal a positive literal in the clause (see Section **??**).

These were all the prefixed clauses we can get, but we can impose a restriction such that fewer prefixed clauses will be computed.

4.4.2 Restricting the Set of Prefix Literals

So now we only consider prefix literals that are generalisations of literals occurring in the clause C. Thus the prefix literals are generated by taking a fresh instance of a literal having a predicate symbol of C and partially instantiating that instance with some literal of C.

As before this is done in such a way that in the list of all prefixed clauses, the prefixed clauses whose prefix literals share *many* variables with C occur *before* the prefixed clauses whose prefix literals share *few* variables with C.

4.5 The Main Module: `killer`

`killer` defines all predicates that are accessible from the outside. `killer` calls predicates defined in other modules. Fig. **??** gives an overview of the modules and their relationships.

We have said on page **??** what we expect TRANSFORMATOR to do. So `killer` supplies the function upsilon, which is used to compute a resolution K-transformation for a clause C. upsilon_list computes a transformation for clause lists instead of single clauses.

Furthermore, tss_condition is supplied to check the test substitution set condition.

And then, find_upsilon is defined to find a transformer for a clause automatically.

`killer` uses the predicates defined in `selfres` and `instances` to find candidates for S_Υ, generates logic formulæ and passes these to some automated prover in order to check the test substitution set condition.

Much in `killer` depends on the options set by the user. These concern mainly two aspects:

- Do we look for self resolvents or for prefixed clauses as candidates for S_Υ?
- Which automated theorem prover do we use?

In **??** we shall investigate the second aspect.

Now we look at the first aspect. Here we distinguish between two versions of find_upsilon:

- find_self_upsilon applies when we look for a transformation characterised by self resolvents.
- find_prefix_upsilon applies when we look for a transformation characterised by prefixed clauses.

I hope this avoids confusion.

4.5.1 S_Υ Consists of Self Resolvents

One way to look for transformers is to have S_Υ contain self resolvents of the clause to be removed. The user can set a flag to choose this kind of search. This is *not* the default.

find_upsilon was introduced on page **??**. Here we call it find_self_upsilon to emphasise that the transformation we look for is characterised by self resolvents. Fig.**??** sketches how this algorithm (or predicate) works.

find_self_upsilon(C,*initial*,*factor*) *level* := 0 ∞ *time* := time_limit(*level*, *initial*, *factor*)*SR* := self_resolvents(C, *level*)$\tilde{S}_\Upsilon \subset SR$ L := select_literals($\tilde{S}_\Upsilon$)$S_\Upsilon \in L$tss_condition(C, S_Υ, *time*) S_Υ fuselfFinding Υ, characterised by self resolvents

As the line $\mathit{level} := 0 \ \infty$ suggests, $\mathsf{find\_self\_upsilon}$ can find anything from zero to infinitely many solutions. Therefore it does not make much sense to model this predicate as a function.

In reality the Prolog predicate `find_upsilon` *enumerates* the solutions one by one. In the real code the forall- and for-statements are backtrackable predicates. I wrote $\mathit{level} := 0 \ \infty$ rather than $\mathit{level} \in I\!N$ because this correctly reflects how the enumeration takes place: Small solutions are enumerated first!

There is an important point not reflected in a forall-statement occurring in this algorithm description, and also in others: Whenever subsets are enumerated, this is done such that small subsets come before large subsets[2].

The function $\mathsf{time\_limit}$ can be described by the following simple formula:

$$\mathsf{time\_limit}(\mathit{level}, \mathit{initial}, \mathit{factor}) := \mathit{initial} * \mathit{factor}^{\mathit{level}} \tag{70}$$

This means that for the self resolvents of level 0, the time limit has the value *initial*. Whenever the level is increased by one, the time limit is multiplied by *factor*. *factor* should be greater than 1, of course.

The function $\mathsf{self\_resolvents}$ computes the set of all self resolvents of C up to level *level*[3]. I emphasise "up to"! It does not only compute the self resolvents that have exactly this level.

$\tilde{S}_\Upsilon$ is a subset of SR, thus a set of self resolvents. However, it is not specified what the selected literals are. This is what $\mathsf{select\_literals}$ is used for. It returns a set containing several copies of $\tilde{S}_\Upsilon$. In each copy, other literals are marked as selected literals.

For example, if $\tilde{S}_\Upsilon$ is a set of three clauses and each clause has two negative literals, $\mathsf{select\_literals}$ returns $2^3 = 8$ copies of $\tilde{S}_\Upsilon$, because there are 8 ways to choose the selected literals.

$\mathsf{tss\_condition}$ returns *true* if it could be proven in *time* milliseconds that S_Υ characterises a resolution K-transformation for C.

Infinity

We can spot two sources of infinity in Fig.**??**. The one is the line $\mathit{level} := 0 \ \infty$. The other source is that testing the test substitution set condition requires automated theorem proving. The only way we could control this infinity is having an argument *time* for the function $\mathsf{tss\_condition}$ that limits the time for the proof.

Note that the actual Prolog predicate `find_upsilon` *enumerates* one S_Υ after the other. We shall now have another close look at how this enumeration is done.

Consider Fig.**??**. On the horizontal axis there are the candidates for S_Υ. Considering that we can only handle finite sets S_Υ, the set of all potential S_Υ, that is, the set of all finite subsets of the set of self resolvents, is infinite but *enumerable*.

On the vertical axis there is the proof length (or proof time, if you want).

Each vertical line stands for a proof attempt. Some of the proof attempts are not finite. The candidates for S_Υ become larger and larger, therefore the proofs that succeed become longer and longer. Not necessarily, of course, but as a tendency.

We guarantee completeness by a sort of *diagonalisation*.

First we cover the proofs below the lowest diagonal line, then below the second, and so on. Our strategy is not only complete, but it also attempts to find a small S_Υ. First we expect (**??**) to be easier to prove for a small S_Υ, and second we expect that a small S_Υ characterises a better transformer.

[2]Actually I had to modify the Prolog predicate `sublist` for this purpose.

[3]When we introduced $\mathsf{self\_resolvents}$ in Fig.**??** we said that it returns a list of lists. This was necessary for the recursive call. From now on we shall neglect this minor point.

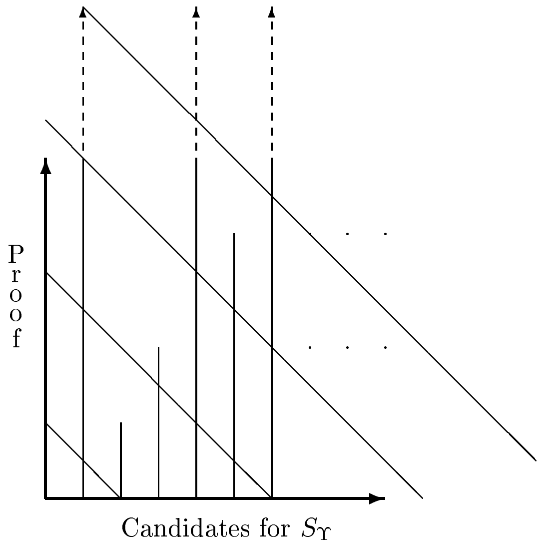

Figure 4.2: Self resolvents

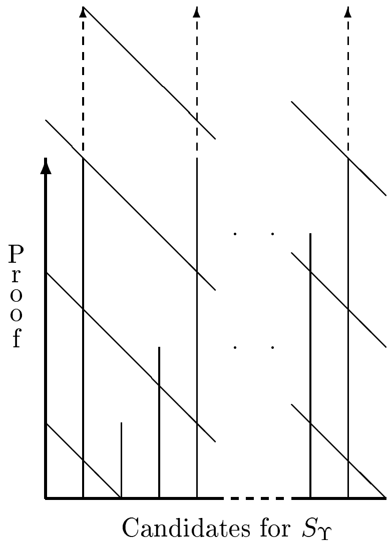

Figure 4.3: Prefixed literals

The performance of this strategy will be very sensible with respect to the initial CPU time limit and the factor by which this limit is multiplied in each iteration. Ideally, they should be such that the set S_Υ is caught in the first try. The number of candidates for S_Υ grows exponentially[4] with the level of self resolvents. Thus if S_Υ escapes the first time due to a time-out, it has to be re-found in a much larger set.

On the other hand, we should not waste too much CPU time trying to prove formulæ that are not tautologies. My experience with the leanT^4P theorem prover is that these calls of a proof procedure will not terminate unless timed out. The Otter prover, however, will usually terminate before being timed out.

Example 4.5.1 Let us look at permutation clauses. Take

$$C := R(x_1, x_2, \ldots, x_n) \Rightarrow R(x_n, x_1, x_2, \ldots, x_{n-1}) \tag{71}$$

We have seen in Corollary **??** that the transformer for this clause is characterised by the set of all self resolvents of C up to the $(n-2)$th level.

For $n = 2$, the leanT^4P theorem prover needed about 0.15 milliseconds to prove the test substitution set condition; for $n = 9$, it took about 2.6 milliseconds. An increase of n by one caused the proof to last around 1.5 times longer. Thus the example suggests that it is reasonable to let the CPU-time rise exponentially with the level of self resolvents.

Recall that the initial CPU time limit and the factor can be given as arguments. The defaults are 10 milliseconds and a factor of two.

◁

[4]Even if the number of self resolvents depends on the level only by a linear function, the number of *subsets* of the set of self resolvents grows exponentially.

4.5.2 S_Υ Consists of Prefixed Clauses

By default, we search for a transformer where S_Υ consists of prefixed clauses.

Here we shall speak of find_prefix_upsilon rather than find_upsilon to emphasise that we look for a transformation characterised by prefixed clauses. find_prefix_upsilon is shown in Fig.??.

find_prefix_upsilon(C,*initial*,*factor*) PC := prefix_clauses(C)*cardin* := 1 ∞ *time* := time_limit(*cardin*, *initial*, *factor*)S_Υ $\subset PC$ *where* $\#S_\Upsilon$ $\leq$*cardin* tss_condition(C, S_Υ, *time*) S_Υ fuprefFinding Υ, characterised by prefixed clauses

Let us compare Fig.?? and Fig.??. The most important difference is that prefix_clauses is called *before* time_limit. In contrast, the call to self_resolvents is nested into the for-loop. prefix_clauses is only called once. For a clause without function symbols, the set of prefixed clauses is finite. It is computed at the very beginning of the search.

However we prefer a small S_Υ to a big S_Υ. *cardin* stands for the cardinality of the present S_Υ candidate.

Infinity

When we look at clauses without function symbols, PC is a finite set. S_Υ is a subset of this set. Thus in contrast to ??, find_prefix_upsilon can not find infinitely many solutions. However, checking the test substitution set condition for some clause set S_Υ will still not terminate in general.

Furthermore, we are interested in finding a set S_Υ as small as possible that characterises a transformation. Therefore we start enumerating all subsets of PC having size 1. Again, we limit the CPU-time for checking the test substitution set condition. If this does not yield a transformer, we enumerate all subsets of size less or equal 2 and multiply the CPU-time limit by some factor, and so forth.

Fig.?? illustrates how this works. Compare this to Fig.??! We have only one source of infinity. The horizontal axis is finite. Nevertheless we apply a similar kind of diagonalisation for reasons of efficiency. We prefer a solution in the lower left corner. So this is where we start our search.

Example 4.5.2 The proofs of the test substitution set condition for the clauses of Table (??) took between 0.5 and 5 milliseconds, except for clause no.12 which took 34 milliseconds, but clause no.12 is the only clause for which S_Υ has two elements, so this is no surprise.

For the clause shown in Section ??, the proof took about seven milliseconds.

$\triangleleft$

No matter how we choose our candidates for S_Υ, our diagonalisation ensures that every S_Υ that fulfills the test substitution set condition will eventually be given enough time for the proof to succeed.

4.5.3 The Test Substitution Set Condition: `tss_condition`

We shall now look at tss_condition.

tss_condition(C,+S_Υ,*time*) serves to check the test substitution set condition. C is a clause, S_Υ is a list of clauses, and *time* is a limit for the CPU time (in milliseconds).

tss_condition works as is shown in Fig.??.

tss_condition(C, S_Υ, *time*) L_Υ := char_lits(S_Υ)one_σ_condition(L_Υ) *formula* := tssc_formula($C\psi$, S_Υ) prove(*formula*, *time*)Σ := test_substitutions(C, L_Υ)$\sigma \in \Sigma$ *formula* := tssc_formula($C\sigma$, S_Υ) $\neg$ prove(*formula*, *time*) *falsetrue*futssTesting the test substitution set condition char_lits computes the set of characteristic literals (see Def. ??).

one_σ_condition checks the condition of Corollary ??. If this condition is fulfilled, we must consider only one test substitution ψ. ψ simply "freezes" the variables of C. When we look at clauses without function symbols, this condition is fulfilled unless one variable occurs twice as argument of the same characteristic literal.

The then-Case

The function tssc_formula computes the formula (??). The function prove calls some automated theorem prover and gives this prover *time* milliseconds to prove that *formula* is a tautology. If the prover succeeds, S_Υ does indeed characterise a resolution K-transformation for C. Thus *true* is returned.

If the prover does not succeed, *false* is returned. This may be due to the time limit and thus by no means implies that S_Υ does not characterise a resolution K-transformation for C !

The else-case

If the condition of Corollary ?? is not fulfilled, test_substitutions is called to compute Σ (see Def. ??). Technically, not Σ, but the test instances are computed. At no time do we keep substitutions as a data structure.

If (??) fails for any test substitution, the loop is interrupted and *false* is returned. Otherwise, true is returned.

4.5.4 Otter or leanT^AP

The test substitution set condition is checked by a theorem prover, of course. The user can choose between two theorem provers. tss_condition works very differently depending on the proof procedure the user chooses. There are three possibilities:

- The leanT^AP prover is used. The leanT^AP prover is written in Prolog. It is very small, and for short clauses (transitivity, euclideanness etc.) it performs well. Apart from that, calling a Prolog predicate from a Prolog program seems a lot more trustworthy than making systems calls from a Prolog program, which we have to do if we want to use a non-Prolog prover. Therefore leanT^AP is used by default.

- The Otter theorem prover is used. Otter is very fast. However this shows only when proving formulas of a certain size. Usually a time span of, say, one second is neglectable when we are talking about theorem proving. For our purposes it is not! Typically we will test dozens or hundreds of formulæ before we encounter an S_Υ that characterises a transformation. The actual proof for S_Υ may only take a few milliseconds.

  Apart from that, the Otter theorem prover can not be forced to quit after less than one second. So it is pointless to use Otter if the proof is to be timed out after less than one second.

- Another possibility is to use leanT^AP as long as the CPU time limit is smaller than a certain number specified by the user, and use Otter as soon as the CPU time limit is at least this number. Recall that the CPU time limit is computed by a function time_limit (see page ??). Roughly speaking the CPU time increases with the size of the candidate for S_Υ.

  A reasonable choice would be to switch to Otter as soon as the proof time limit exceeds one second. For one thing, Otter can not be timed out earlier.

Furthermore, one second is about the order of magnitude where Otter starts to perform better than lean$T^A\!P$.

A lean$T^A\!P$ Pitfall

In theory S_Υ is a *set* of clauses. In TRANSFORMATOR, this set is represented as a *list*. In theory the elements of a set are in no order. In practice, the members of a list are in some order. Depending on this order, the transformation of a formula yields different results. Of course these differences are of a very trivial kind. Nothing is different except of the *order* of subformulæ in a conjunction.

The problem is that lean$T^A\!P$ sometimes reacts very, very sensibly to such differences! Recall the clause $R(x, y) \wedge R(y, z) \Rightarrow R(z, x)$ (see Table **??**). Here S_Υ has two elements. Depending on the order of these elements `tss_condition` will run several milliseconds or several hours. And one can easily imagine that the formulæ in question are not very large.

How can we solve this problem in general? It would be hopeless to predict in advance which permutation of S_Υ will result in the shortest running time of `prove` and permute S_Υ accordingly. In order to mitigate the problem we allow the user to have *all* representations of S_Υ tested rather than just one.

If S_Υ has two or three elements, testing all its permutations is not much of a problem. If S_Υ has more elements, it is absolutely out of question to test all the permutations.

By default, only one representation of S_Υ will be tested.

`tss_condition` and Otter

Otter is a very powerful theorem prover written by William McCune (see [**?**]). Otter is written in C and is basically non-interactive. Input and output is done via standard input and output. These can be redirected by UNIX© pipes as usual. So typically Otter would be called by something like

$$\texttt{otter} \quad < \quad input\!-\!file \quad > \quad output\!-\!file \tag{72}$$

Fortunately it is not necessary to create files for the communication between TRANSFORMATOR and Otter. The two programs communicate via *streams*. SICStus 3.0 provides predicates for stream input and output.

This starts with the call

$$\texttt{exec}(\texttt{otter}, [\texttt{pipe}(\texttt{In}), \texttt{null}, \texttt{null}], \texttt{PID}) \tag{73}$$

The variable `In` is instantiated to a stream that is used by Otter as input. The output and error streams are connected to `/dev/null`.

Otter does not start to work on this input before the stream is closed. Now TRANSFORMATOR writes the formula (**??**) (in Otter syntax) onto this stream. Furthermore, it writes some other things that are required for an Otter input file.

It is possible to set a flag in the input stream or file such that the CPU time for the proof search is limited. Thus TRANSFORMATOR delegates the responsibility of limiting the CPU time to Otter directly rather than interfering while Otter is running.

The only disadvantage is that the limit must be one second at least. However, for problems of a very small size lean$T^A\!P$ should be favoured anyway.

When the formula is written onto the stream, the stream is closed, and

$$\texttt{wait}(\texttt{PID}, 26368) \tag{74}$$

causes TRANSFORMATOR to wait until Otter has finished its work. The second argument[5] of `wait` is the exit status of Otter. It is 26368 iff Otter finds a proof. Termination of Otter is guaranteed because of the time limit.

4.6 Eliminating Clause *Sets*

tss_condition and find_upsilon also work for clause sets $\mathcal{C}$ rather than a single clause C. Thus we can verify that a clause set S_Υ characterises a transformation for a clause set $\mathcal{C}$, and we can also attempt to find a clause set S_Υ that characterises a transformation for a clause set $\mathcal{C}$.

The former is easy, whereas the latter is not, due to the gigantic search space. It has been shown in Section **??** how candidates for S_Υ can be generated. Of course it is crucial to enumerate simple candidates first. It would be hopeless to find a transformation by inspecting the candidates in a "depth-first-search"-way.

We shall not describe in detail how the search is realised. The basic idea is a diagonalisation as described on page **??**. From Section **??** it should be clear that this diagonalisation is much more complicated than in the case of the elimination of just one clause.

Furthermore, we do not claim any kind of completeness of this search. Of course it would have been possible, given a clause set $\mathcal{C}$, to enumerate all sets that consist of clauses in $\mathcal{C}$, self resolvents of clauses in $\mathcal{C}$, clauses subsumed by clauses in $\mathcal{C}$, and resolvents between all of the latter. Practically, it would be impossible to inspect all of these sets.

As long as we do not have any further theoretical criteria to reduce this search space, all we can do is impose certain restrictions that seem reasonable following our intuition. The examples of Section **??** show that this works quite well.

4.7 Other Modules

The module `make_formulae` contains predicates that actually generate the logic formulæ that have to be verified to check the test substitution set condition.

`lists_and_clauses` contains some auxiliary predicates that relate to lists and to the terms that represent clauses. The predicates are similar to usual list predicates like `member`, `append`.

The predicates in `lists_and_clauses` are used by all other modules. This is not depicted in Fig.**??** to keep the picture simple.

`help` defines some predicates that print help messages for the user.

`leantap` contains the lean$T^A\!P$ theorem prover. `nnf` contains a predicate that computes the negation normal form for an arbitrary formula. These two modules were designed by Bernhard Beckert and Joachim Posegga ([**?**]), with minor changes by myself.

Finally, `interface` defines some predicates that manage the file interface (see Section **??**).

4.8 Meta Predicates

Let us make some general remarks about the implementation of TRANSFORMATOR.

[5]The strange number of 26368 has to do with a peculiarity of UNIX© calls. The Otter manual says that Otter returns the status 103 iff a proof is found. However this only refers to the low-order eight bits of the exit status. So in order to translate the status obtained by `wait` into the status described in the Otter manual, we have to cut off the second byte, that is, divide by 256 (2^8). Note that $26368 : 256 = 103$ (see [**?**, p.198]).

TRANSFORMATOR uses meta predicates very often. A meta predicate is a predicate that deals with information about the execution state of the program. For example, it could be queried whether a variable is instantiated at a certain point of time. Such a thing cannot be expressed in Pure Prolog. Two other important examples are the use of `time_out` and the interaction with Otter.

One could name single reasons for each use of a meta predicate. On a more abstract level however there is one very important reason: the dual function of *Prolog variables* .

On the one hand Prolog variables serve the purpose that variables in a programming language always serve: to represent an *object*. In a sense which depends on the programming language the variables will finally be instantiated to the objects that are the result of the computation.

Now what is an *object* in our context? Essentially our domain is logic , and our objects are *logic formulæ*, clauses and clause sets in particular. In general, logic formulæ contain *logic variables*. In our representation of formulæ, logic variables and Prolog variables are identified. This is sensible even if it were only for the interaction with leanT^4P , which uses the same technique.

However this also constituted one of the difficulties of the programming task: keeping the two purposes of variables carefully apart, not allow logic variables to be instantiated by mistake and the like.

4.9 Using TRANSFORMATOR Directly

TRANSFORMATOR is written in the **Prolog** programming language, more precisely, in **SICStus Prolog 3.0**. For an introduction to Prolog see [?]. For the particularities of the SICStus 3.0 implementation see [?].

TRANSFORMATOR **only runs under SICStus 3.0, not under earlier versions!** I use several built-in predicates of SICStus 3.0, such as unification with occurs check.

This section explains how TRANSFORMATOR is used on the Prolog interpreter level. If TRANSFORMATOR is to be used to *find* a transformation for some clause or to *verify* that some clause set S_Υ characterises a transformation, this is a convenient way to use TRANSFORMATOR.

It is also possible to *transform* clauses and clause sets on the Prolog interpreter level. It is not very convenient, however. For this task, it is recommended to use the file interface described in the next section.

Using TRANSFORMATOR on the Prolog interpreter level requires some basic knowledge of Prolog, of course. Furthermore, it is necessary to understand how the objects we deal with are represented.

Following the convention used in Prolog manuals, and also in the language itself, *predicate/arity* means that the predicate *predicate* has arity *arity*. For example we might say "we used `append/3`" in order to express that we used `append` and that `append` has three arguments.

When no confusion can arise, we shall often identify a predicate with a call to that predicate. E.g. we say "the predicate succeeds" instead of "the call to this predicate succeeds".

4.9.1 Representation of Clauses and Clause Sets

Function, predicate, and constant symbols can be arbitrary Prolog atoms. Logic variables are represented as Prolog variables. Logic atoms are represented in a canonical way, that means, a logic atom composed of symbols following the above rules (Prolog atoms for non-variable symbols, Prolog variables for variables) is syntactically identical to the representation of this logic atom in TRANSFORMATOR.

At this point I should say that we must not confuse Prolog atoms and logic atoms! Prolog atoms will serve as function, predicate, or constant symbols. A logic atom is composed of a predicate symbol and argument terms (see page **??**).

Literals are not represented as such. Only in a clause can an atom be known as representing a positive or negative literal.

Clauses are represented by Prolog terms of the form

$$\texttt{clause}(P, N) \tag{75}$$

where P and N are Prolog lists . The elements of P and N are logic atoms. P stands for the set of positive literals and N stands for the set of negative literals. The only way to recognize the logic atoms in N as negative is their position in this list.

For example the transitivity clause for a predicate `p` is represented as

$$\texttt{clause}([\texttt{p}(\texttt{X},\texttt{Z})],[\texttt{p}(\texttt{X},\texttt{Y}),\texttt{p}(\texttt{Y},\texttt{Z})]) \tag{76}$$

Clause sets are represented as Prolog lists of clauses.

When we say "C is a clause", we mean "C is instantiated to a Prolog term representing a clause according to the above syntax".

4.9.2 Loading TRANSFORMATOR

TRANSFORMATOR is supplied as a directory containing 11 SICStus modules[6]. Recall that TRANSFORMATOR runs under the **SICStus 3.0** interpreter, usually invoked by `sicstus3`. Loading TRANSFORMATOR in a SICStus interpreter is done by the call

`use_module(killer).`

The other SICStus modules are called from the `killer` module automatically.

4.9.3 Predicates

It has been said that there are three (or if transforming clause *sets* is counted separately, four) tasks TRANSFORMATOR can perform. These are explained in Section **??**. For each of these tasks, there is a corresponding Prolog predicate.

Apart from these, the user can set certain options. Technically, setting an option is also done calling a predicate. But here we shall only treat the four main predicates supplied by TRANSFORMATOR.

Some predicates have several versions with different arities. The reason is very simple. There are some arguments that the user may, but does not need to give, to the predicate. If the argument is not supplied, a default value is assumed.

We shall now list the predicates, explain them shortly and say what arguments they require.

`upsilon/3`

`upsilon`(D,S_Υ,*transformed*) instantiates *transformed* to $\Upsilon(D)$. $\Upsilon(D)$ is a set of clauses represented as a list.

At the time of the call, D must be a clause. S_Υ must be a list of clauses where each clause has its own set of variables. D and S_Υ must not share variables. *transformed* must not be instantiated.

[6]and four shellscripts (see next section)

If these conditions are met `upsilon` always succeeds. It is basically a functional predicate.

upsilon_list/3

While `upsilon/3` is used to transform one clause, `upsilon_list/3` is used to transform *sets* of clauses. `upsilon_list(` DL,S_Υ,*transformed*) calls `upsilon` for all clauses in DL and appends the resulting clause lists to obtain *transformed*.

At the time of the call, DL must be a clause list. S_Υ must be a list of clauses where each clause has its own set of variables. DL and S_Υ must not share variables. Two clauses in DL may share variables, however. *transformed* must not be instantiated.

`upsilon_list` is also a functional predicate, of course.

tss_condition/2 /3

`tss_condition(`C,S_Υ,*time*) holds if S_Υ can be proven to characterise a resolution K-transformation for C in *time* milliseconds.

`tss_condition(`C,S_Υ) holds if S_Υ can be proven to characterise a resolution K-transformation for C in 20000 milliseconds.

At the time of the call, C must be a clause or list of clauses. S_Υ must be a list of clauses where each clause has its own set of variables. *time* must be a positive integer.

find_upsilon/2 /3 /4 /5

`find_upsilon(`C,S_Υ) instantiates S_Υ to a list of clauses that characterises a resolution K-transformation for C, if it succeeds.

`find_upsilon(`C,S_Υ,*total*) is the same as before, only that the argument *total* is a limit for the total time (in milliseconds) the search may last. If no transformer was found by that time, the predicate fails. The time limit refers to real time rather than CPU time. The limit is quite rough, because it would be anything but good programming style for a Prolog program to test a time limit every twenty milliseconds. Using time limits at all is bad enough! Practically this is not a problem.

When we call `find_upsilon(`C,S_Υ,*initial_time*,*factor*), the arguments *initial_time* and *factor* are used to control the search. This is explained in Section **??**. *initial_time* is the initial time *in milliseconds*.

Finally there is `find_upsilon(`C,S_Υ,*initial_time*,*factor*,*total*). This should be clear.

At the time of the call, C must be a clause or list of clauses. S_Υ must be uninstantiated. *initial_time* and *total* must be positive integers, *factor* must be a float or integer greater than 1.

When one of the versions of `find_upsilon` having two, three, or four arguments is called, this invokes a call to `find_upsilon/5`, where default values are assumed for *initial_time*, *factor*, and *total*. The default values are:

$$\begin{aligned} \textit{initial\_time} &= 10\,(milliseconds) \\ \textit{factor} &= 2 \\ \textit{total} &= 1000000000\,(milliseconds) \approx 11\,days \end{aligned}$$

Predicate	Description	page
instances	S_Υ consists of prefixed clauses **(default)**	??
selfres	S_Υ consists of self resolvents	??
leantap	The prover leanT^AP is used **(default)**	??
otter	The prover Otter is used	??
switch(ms)	The prover Otter is used for proofs whose time limit is at least ms milliseconds	??
noperms	Only one representations of S_Υ is checked **(default)**	??
perms	All permutations of S_Υ are checked	??
someprefixes	A prefix literal must be a generalisation of a literal occurring in the clause **(default)**	??
allprefixes	A prefix literal is a literal with at least one variable from the original clause	??
condense	Self resolvents are shortened by condensation **(default)**	??
nocondense	Self resolvents are not shortened by condensation	??
no_subsumption	No subsumption tests are done for S_Υ	??
min_subsumption	No self resolvents are considered that are subsumed by other self resovents of the same level **(default)**	??
max_subsumption	No self resolvents are considered that are subsumed by other self resovents of any level	??
silent	TRANSFORMATOR runs in silent mode	—
verbous	Proof attempts are reported **(default)**	—

Table 4.1: List of options

4.9.4 Options

Various options were introduced throughout this chapter. Technically these are realised by predicates of arity 0 or 1 that are asserted or retracted from the database. This is not done by the user directly. It would be awkward to have to retract the 'Otter'-flag in order to use leanT^AP , for example. The user just calls predicates with generic names. There are Prolog clauses in the database that will actually retract and assert the former predicates. Table **??** gives an overview of all options and defaults. The third column contains references to the pages where these options are explained in detail.

4.9.5 Help

Apart from the main predicates and the options, there are a few predicates related to help. These are

- Call `help/0` and you see which options you can set and which help messages you can obtain.
- Call `explain/0` to get a *very brief* description of TRANSFORMATOR.
- Call `flags/0` to get the current setting of all options.
- Call `defaults/0` to set all options to their default values.

4.10 Using the File Interface

This section explains how TRANSFORMATOR is used via the file interface. The structure of this section is similar to the last section.

The file interface is designed for interaction with the Otter theorem prover. This is not to be confused with the internal use of the Otter theorem prover in TRANSFORMATOR. The idea is that an Otter input file can be *pre-processed* using a resolution K-transformation.

Thus the input file for the killer transformation program is very much like an Otter input file, or in other words, an Otter input file can be used as input for TRANSFORMATOR after a little editing.

The Otter syntax is explained in detail in [**?**].

Essentially, an Otter input file contains a clause set. If we want to transform this clause set with a resolution K-transformation, we have to *mark* the clause we want to remove. Then we give this file to the transformation program. The output of TRANSFORMATOR contains the transformed clause set, which can be given to Otter for proving.

If we only want to *find* a transformation or *verify* that a clause set S_Υ characterises a transformation, this can also be done using the file interface, although this can more conveniently be done using the SICStus interpreter directly (see previous section). If the user wants to solve several problems, it is also faster to use TRANSFORMATOR on the interpreter level because the Prolog program has to be loaded only once, and then the user can make as many queries as desired.

Anyway, using the file interface, the syntax for the input file is always in an Otter-like fashion.

4.10.1 Input File Syntax

On page **??** it has been said that there are three tasks we want to perform. For each, a shellscript is provided so that actually there is a program for each.

It should be clear that the different tasks require different inputs. For example, it does not make sense to give the clause set S_Υ as input if this is exactly what we want TRANSFORMATOR to *find*.

However, the routine reading the input does not take this into consideration. Thus the syntax of the input file can be described in a uniform way, independent of which of the programs we use. If we give an input that does not make sense in the particular context, e.g. give the clause set S_Υ as input if we want to *find* S_Υ, it does not hurt. This input will simply be ignored.

Otter accepts *clause lists* as well as *logic formulæ* as input. However, TRANSFORMATOR computes a transformation of *clauses*. Therefore TRANSFORMATOR only accepts *clause lists*. Applying the transformation to general formulæ would require transforming the formula in clause normal form first.

TRANSFORMATOR requires the use of Prolog style variables, i.e., a variable starts with a capital letter or an underscore. Otter understands Prolog variables, but a flag has to be set.

Otter maintains three lists of clauses called `sos`, `usable`, and `passive`. It is not relevant for our purposes what these mean, only that TRANSFORMATOR *preserves* the distinction. That is, if a clause D is contained in the list `sos` in the input file, the result of transforming D is written to the list `sos` in the output file, and accordingly for the other lists.

Thus clause lists are input in real Otter syntax, and they are actually "understood" by TRANSFORMATOR. TRANSFORMATOR can read in the clause lists and transform the clauses contained in them.

Furthermore, there are a few commands, or options, especially designed for TRANSFORMATOR. In other words, these commands are *only* "understood" by TRANSFORMATOR. These are in a syntax similar to Otter syntax, but each option is preceded by '`killer_`' to avoid confusion with Otter commands.

Everything else is not "understood" by TRANSFORMATOR. Any line in the input file that is not "understood" will simply be *forwarded* to the output file. It is assumed that it is some Otter command. If it is not, this is Otter's problem!

The following lines in the input file are understood by TRANSFORMATOR:

- `killer_list(remove).` This marks the beginning of the list of clauses that are to be eliminated. Of course this list can consist of one clause only.

  The list has to be terminated by the line "`end_of_list.`" (this is usual Otter syntax).

- `killer_list(s_ups).` This marks the beginning of the list of clauses in S_Υ. Just as before, the list has to be terminated by the line "`end_of_list.`"

- `killer_set(`*flag*`).` This is to set the flag *flag*. The flags will be explained later.

- `killer_clear(`*flag*`).` This is to withdraw a flag.

- `killer_assign(`*parameter*`,`*value*`).` This is to assign a value to a parameter. The parameters will be explained later.

4.10.2 Commands

It has been said in the last section that TRANSFORMATOR is supplied as a directory containing 11 SICStus modules and four shellscripts (see next section). When we use the file interface, this directory must be our working directory.

When we use TRANSFORMATOR on the SICStus interpreter level, we have predicates for each task TRANSFORMATOR can perform. Using the file interface, we have a command for each task. Each command corresponds to a shellscript that starts Prolog and calls the according Prolog predicates.

In each case, the input file is given as an argument. The output is written to standard output and can be redirected using UNIX© pipes.

In the following list of commands, S_Υ stands for the clause list contained in the input file and started by the line `killer_list(s_ups).` $\mathcal{C}$ stands for the clause list following the line `killer_list(remove)`. This should be clear from the description of the input file syntax.

- `transform` transforms the clause lists in the input file using the transformation characterised by the clause set S_Υ. The transformed clause lists are written to standard output.

- `tss_condition` verifies that S_Υ characterises a resolution K-transformation for $\mathcal{C}$. Depending on the success of this verification, a message is written to standard output.

- `find_upsilon` tries to find a transformation for $\mathcal{C}$. The result is written to standard output.

- `find+transform` tries to find a transformation for $\mathcal{C}$ and transforms the other clause lists using this transformation. The transformed clause lists are written to standard output.

Parameter	Description	page
`switch`	The prover Otter is used for proofs whose time limit is at least the value assigned to this parameter (in milliseconds). Must be set to a positive integer.	??
`initial_time` or `time`	Sets the time limit (in milliseconds) for proofs during the first iteration. Must be set to a positive integer.	??
`total_time`	Sets the time limit (in milliseconds) for the whole process of finding a transformation. Must be set to a positive integer.	—
`factor`	Sets the factor by which the time limit is multiplied for subsequent iterations. Must be set to an integer or float > 1.	??
`solution_number`	When the command `find_upsilon` is used, this parameter can be used to choose the number of solutions for S_Υ that should be output. Must be set to a positive integer. If this option is not set, the number defaults to 1.	—

Table 4.2: List of parameters

4.10.3 Options and Parameters

Various options were introduced throughout this chapter. Furthermore, when we used the SICStus interpreter directly, some of the predicates had various arguments. When we use the file interface, all these options can be chosen and all the arguments can be given to the predicates just as well. Technically this is done in a uniform way, following the style of Otter flags and parameters.

All options listed in Table **??** except `switch(`*value*`)`, `silent`, and `verbous` can be set simply by having a line "`killer_set(`*option*`).`" in the input file. When this file is processed, this line is directly translated into a Prolog call to the predicate *option*.

It is also possible to *clear* an option, but not recommended, because it is confusing to do so. For example, `killer_clear(leantap)` will cause Otter to be used to test the test substitution set condition. However, it is much more sensible to use `killer_set(otter)` for this purpose.

A parameter is assigned a value using `killer_assign(`*parameter,value*`)`. The parameters are listed in Table **??**. The last column contains references to the pages where the meaning of the parameters is explained in more detail.

Example 4.10.1 A simple example will illustrate most of the points we have made in this section. Consider the file `example`:

```
set(hyper_res).
set(prolog_style_variables).
killer_assign(initial_time,50).
killer_assign(factor,5).
list(sos).
p(a,b).
p(b,c).
end_of_list.
list(usable).
p(d,e).
end_of_list.
```

```
p(k,l).
list(passive).
-p(a,c).
end_of_list.
killer_list(remove).
-p(X,Y) | -p(Y,Z) | p(X,Z).
end_of_list.
banana.
```

If we now type

```
find+transform example > example.out
```

on the shell, the output file `example.out` looks as follows:

```
set(hyper_res).
set(prolog_style_variables).
p(k,l).
banana.
list(usable).
-(p(e,_1984)) | p(d,_1984).
p(d,e).
end_of_list.
list(sos).
-(p(b,_2329)) | p(a,_2329).
p(a,b).
-(p(c,_2500)) | p(b,_2500).
p(b,c).
end_of_list.
list(passive).
-(p(a,c)).
end_of_list.
```

The clause lists `usable`, `sos`, and `passive` were transformed using the transformer for transitivity.

Note that in the input file, `p(k,l)` does not occur in a clause list. It is isolated. Therefore it is not recognised as a clause. Instead, it is forwarded to the output file. Neither is the string "`banana.`" "understood" by TRANSFORMATOR. It is not sure whether Otter can make anything out of it, but TRANSFORMATOR does not care!

Chapter 5

Conclusion and Outlook

Based on the definition of resolution K-transformations given by Ohlbach [**?**], I have shown how such transformations can be found automatically for arbitrary sets of clauses.

A resolution K-transformation for a clause set $\mathcal{C}$ is characterised by a clause set S_Υ, and clauses are transformed by resolving them with clauses in S_Υ. The definition of resolution K-transformation requires that S_Υ has to meet the *test substitution set condition*. A priori there is no restriction as to how a reasonable candidate for S_Υ is found.

Now I say that in general, S_Υ should consist of *prefixed clauses* of clauses in $\mathcal{C}$. A prefixed clause for a clause C is a clause that has an additional negative literal as first literal, and else is like C. Resolution is done on this literal. Thus if we transform unit clauses, the extra literal is resolved away, leaving an *instance* of C where some arguments are instantiated by terms occurring in the transformed unit clause. Thus we may look at resolution in this context as a *technique* to add more specific instances of a clause C.

This paper as well as [**?**] contain examples where a transformation for a clause C is characterised by C itself. We have seen that this case can be regarded as an *optimisation* of the general case.

For permutation clauses, saying that S_Υ should consist of prefixed clauses does not lead to reasonable results. Here *self resolvents* of the permutation clause are better. I gave a general description of these transformations that applies to all permutation clauses. So it is not necessary to test different sets of self resovents. For this class of clauses the result can be given once and for all.

I have implemented the search for a clause set S_Υ that characterises a transformation, both under the assumption that S_Υ should consist of prefixed clauses, or that it should consist of self resolvents. Looking back, I only found the former to be useful.

The program TRANSFORMATOR works for clause *sets* $\mathcal{C}$, which is a generalisation of the case that we want to eliminate only one clause. For the elimination of only one clause I claim a kind of completeness of the search for a transformation.

I have investigated a small class of clauses exhaustively and found a transformation for every clause in this class. This class includes the symmetry, transitivity, and euclideanness clauses.

I did not attempt to achieve completeness for the general case that $\mathcal{C}$ consists of *several* clauses. We assumed that in this case, S_Υ should consist of clauses from $\mathcal{C}$, self resolvents, subsumed clauses, and resolvents among all these. This characterisation is so general that it would have been hopeless to enumerate all such sets systematically. It would be interesting to develop the theory further to the end of having stronger criteria for S_Υ-candidates.

However, for examples like the set consisting of the euclideanness clause and the symmetry clause, TRANSFORMATOR succeeded in finding a transformation. Other examples were given.

This work mainly concerns clauses without function symbols. The results also apply for clauses that contain function symbols in a harmless way. "Harmless" means that no new terms are generated by resolution with the clause.

I have given a negative result for one clause containing a function symbol in a "non-harmless" way, namely, condensed detachment. The result is negative not in the sense that we cannot find a transformation. We can find a transformation according to our definition, but it is pathological. So another open question is whether the idea developed there may be useful for the elimination of other clauses of this kind.

I have not focussed on the *applications* of resolution K-transformations, but rather on the process of finding resolution K-transformations. It has been said that there are at least four areas where resolution K-transformations may be applied: reduce the search space of theorem provers, eliminate loops in logic programs, parallelise closure computation algorithms and support automated complexity analysis.

For all of these areas, it has been outlined in [**?**] how resolution K-transformations may be used, but especially the areas of parallelising closure computation algorithms and supporting automated complexity analysis still need much development.

My program TRANSFORMATOR may serve as a tool to study the area of automated theorem proving, since clause sets in Otter syntax can be transformed, and then the behaviour of Otter for the original clause set and the transformed clause set can be compared. It would probably not be difficult to do the same for other provers. The few experiments I have done suggest that Otter is not so easily fooled, whereas for leanT^4P, the transformation can mean the difference between a proof that can be found and a proof that cannot. But these results are by no means representative, and an extended series of experiments has yet to be done.

List of Figures

Bibliography

[BG95] David Basin and Harald Ganzinger. Automated complexity analysis based on ordered resolution. Technical report, Saarbrücken, 1995. www-addresses http://www.mpi-sb.mpg.de/guide/staff/hg/hg.html and http://www.mpi-sb.mpg.de/guide/staff/basin/basin.html

[BP95] Bernhard Beckert and Joachim Posegga. lean$T^{A}P$: Lean Tableau-Based Theorem proving. Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, Karlsruhe, Germany, 1995. www-address http://i12www.ira.uka.de/ posegga/leantap/leantap.html

[Bra75] Daniel Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4(4):412-430, 1975.

[CM87] W.F.Clocksin and C.S.Mellish. Programming in Prolog. Springer Verlag, 1987

[CW95] Mats Carlsson and Johan Widén. SICStus Prolog User's Manual. Swedish Institute of Computer Science, Kista, 1995. www-address http://www.sics.se/ps/sicstus.html

[Lam78] Erich Lamprecht. Einführung in die Algebra. Birkhäuser, 1978

[McC94] William McCune. OTTER 3.0 Reference Manual and Guide. ANL-94/6, Argonne National Laboratory, Argonne, 1994. www-address http://www.mcs.anl.gov/home/mccune/ar/otter/

[OGP94] Hans Jürgen Ohlbach, Dov Gabbay, David Plaisted. Killer Transformations. Technischer Bericht MPI-I-94-226, Max-Planck-Institut für Informatik, Saarbrücken, 1994. www-address http://www.mpi-sb.mpg.de/guide/staff/ohlbach/ohlbach.html

[Ohl96] Hans Jürgen Ohlbach. Elimination of Self-Resolving Clauses. Max-Planck-Institut für Informatik, Saarbrücken, 1996. www-address http://www.mpi-sb.mpg.de/guide/staff/ohlbach/ohlbach.html

[Rob65a] John Alan Robinson. Automated Deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1(3):227-234,1965

[Rob65b] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery (JACM)*,12(1):23-41, 1965

[Ste92] W.Richard Stevens. Advanced Programming in the UNIX© environment. Addison-Wesley, 1992