Quantifier Elimination
in
Second-Order Predicate Logic

Diplomarbeit

Thorsten Engel

Saarbrücken, 1996

Author's Address

Thorsten Engel
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken/Germany
E-mail: engel@mpi-sb.mpg.de

Advisors:

Prof. Hans Jürgen Ohlbach, Imperial College London
Prof. Dov Gabbay, Imperial College London

Declaration

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

Saarbrücken, den 20.10.1996

Thorsten Engel

Acknowledgments

Thanks to Hans-Jürgen, for his part of the patience this work demanded from both of us, and his belief that something would come out of it.
Thanks to Renate and Ullrich, who always had an open ear and a free half-an-hour for my just-a-second questions.
Thanks to Volker, for continuous supply with luxury goods, such as moral support and Darjeeling tea, and last-minute help.
Thanks to Kai, Maren and Hannah, to Hannah with a twinkle, for their help with this work, and anyway.
Finally, thanks to Frigga Engel and Horst Engel, for the least scientific and most natural contribution to this work, and for *some* support.

Contents

Chapter 1

Introduction

Mathematical logic is concerned with formalizing mathematical notions and reasoning about them. For this purpose, classical predicate logic has been developed. Now there is an ongoing dispute with a long tradition about the type of quantifiers a logic must provide to be able to express certain notions. Is first-order logic sufficient, or not even second-order logic?

Whatever the choice, there is a fatal trade-off between the expressive power of a logic and the complexity of deduction within this logic: For example, first-order logic is complete, but fails in characterizing, for example, the class of finite models. In second-order logic, on the other hand, this class can be characterized, and this can be done with a simple second-order expression (see chapter 3 for this example in depth). Likewise, many natural formulations of mathematical problems have a second-order form. However, there can be no complete deduction system for second-order logic, which has already been shown by Gödel ([Göd30], for a modern exposition see van Benthem and Doets [vBD84]).

In this situation, we might try to computationally decide whether a problem, formulated as a set of second-order formulas, also has a first-order formulation, and compute a first-order translation if it exists. We could then first describe a problem in a second-order language, translate it into first-order logic and then reason about it in a complete system for first-order logic. Furthermore, for certain problems arising for example in correspondence theory ([vB84]), sometimes the translation itself is the solution of a problem.

Of course, we would not seriously expect that there is any sound and complete algorithm able to make this decision for full second-order logic, and van Benthem and Doets show that these expectations can be proven correct ([vBD84]). Still, for subclasses of second-order logic essentially more expressive than first-order logic, powerful reduction algorithms have been developed.

1.1 Reduction of second-order formulas

Already Ackermann ([Ack35a], [Ack35b]) has proposed basically two algorithms for transforming certain second-order formulas into equivalent first-order formulas. One of his methods works by transforming formulas into a form that can be easily reduced. It is explained in detail and extended by Szalas in [Sza92]. The other method can be seen as a special case of a recent approach by Gabbay and Ohlbach ([GO92]). Both methods try to compute a set of first-order consequences of a second-order formula that is equivalent to this formula and free of redundancies. The formulation by Gabbay and Ohlbach is simpler and more direct than the one of Ackermann. Yet another approach by Nonnengart and Szalas [NS95] computes fixpoints of formula series to find reductions.

However, these algorithms only eliminate predicate quantifiers in second-order formulas. They work on normalized formulas containing skolem functions, and their output may also contain skolem functions. In this context, skolem functions must be interpreted as existentially quantified functions, so formulas containing them are still second-order formulas. None of the algorithms can be used in an obvious way to eliminate these functions, and only in [GO92] there is a sketch of how to do this. McCune addresses the problem of skolem function elimination in [McC88]. His algorithm eliminates skolem functions by reversing the well-known skolemization process. It only tries to preserve satisfiability of a formula, but can easily be proven to be equivalence preserving. However, it fails to eliminate functions for some formulas that have been obtained by skolemization, and has exponential complexity.

In this work, we present the algorithm UNSKOLEMIZE for function elimination that is also based on the idea of reversing skolemization. It reduces a larger class of formulas than the McCune algorithm, and solves a subproblem in quadratic time that the McCune algorithm computes in polynomial time in the worst case. Furthermore, we present the SCAN algorithm as developed by Gabbay and Ohlbach in [GO92]. We add some optimizations that enlarge the set of formulas for which the algorithm terminates. Besides, we describe a C++-implementation of the two algorithms based on the well-known theorem prover OTTER. On the theoretical side, we collect some known fundamental results relating second-order quantifier elimination and some basic observations on its computational aspects.

We will now shortly sketch the methods employed by the SCAN and the UNSKOLEMIZE algorithm.

1.2 Elimination of existentially quantified predicate symbols

Assume we are given the first-order formula

$$\phi \stackrel{\text{def}}{=} (P \vee Q) \wedge (\neg P \vee R)$$

and now consider the second-order formula

$$\Phi \stackrel{\text{def}}{=} \exists P \phi.$$

It is clear that any first-order equivalent of Φ must still imply all consequences of Φ. Especially, it must imply all first-order consequences, that is, consequences expressible as a first-order statement. In a naive approach, we try to generate all P-free first-order consequences of ϕ and then check whether they are already equivalent to Φ.

To this end, we only consider the first-order part $\psi \stackrel{\text{def}}{=} (P \vee Q) \wedge (\neg P \vee R)$ of ϕ, treat P as a simple first-order predicate and compute the set of first-order consequences by resolution. We obtain the single resolvent

$$Res \stackrel{\text{def}}{=} Q \vee R.$$

It is easily checked that $Res \equiv \Phi$: $\Phi \models Res$ holds because of correctness of resolution. For the other direction, given a model $\mathbf{M}$ that verifies Res, we have to choose an interpretation for P so that $\mathbf{M}$ verifies ϕ with this interpretation. Now if $\mathbf{M}$ verifies R, choose P to be true, and if $\mathbf{M}$ verifies Q, choose P to be false. Then $\mathbf{M}$ also verifies Φ.

Essentially this method of generating resolvents is extended in chapter 4, yielding the SCAN algorithm.

1.3 Elimination of existentially quantified function symbols

As is shown in chapter 5, the SCAN algorithm for predicate symbol elimination cannot simply be used as an algorithm for function symbol elimination by just coding functions as predicates. A different approach is suggested by the well-known operation of skolemization, which may be seen as an application of a second-order rewrite rule as described in chapter 5.

Let ϕ be the formula $\forall x \exists y P(x, y)$. This is typically skolemized to $P(x, f_y(x))$, where f_y is a new skolem function. In the context of automated theorem proving, where skolemization is employed, it is shown to be a satisfiability preserving transformation. However, as is explained in chapter 5, with the argument usually employed we could also show the stronger equivalence

$$\forall x \exists y \ P(x, y) \equiv \exists f_y \forall x \ P(x, f_y(x)).$$

We can generalize this by taking a formula $\phi(\bar{x}, y)$ instead of P, which contains y and the vector $\bar{x}$ as free variables, then

$$\forall \bar{x} \exists y \ \phi(\bar{x}, y) \equiv \exists f_y \forall \bar{x} \ \phi(\bar{x}, f_y(\bar{x})).$$

This rewrite rule can now be used for eliminating existentially quantified function symbols, which we shall call skolem functions, by using it backwards, replacing a second-order quantifier with a first-order quantifier. This is less simple as it seems at first sight, because we usually encounter more than one skolem function, which all have to be unskolemized in parallel.

1.4 Overview

The second chapter introduces necessary notation.

The third chapter assembles some known results that are relevant for automated quantifier elimination and shows their implications for its computational complexity. The quoted theorems involve characterization results for first-order logic. Most of the results concern the sheer computability of different reduction problems, but we also derive a complexity result for the time-complexity of reducing second-order quantified propositional formulas. We also prove by elementary means that there cannot be a complete reduction algorithm on undecidable fragments of second-order logic, a theorem elsewhere proven in more complicated ways or using sophisticated machinery.

In the fourth chapter, we present the SCAN algorithm for predicate elimination. A number of optimizations are introduced that improve its performance, its termination properties and the compactness of its output.

The fifth chapter is concerned with the UNSKOLEMIZE algorithm for function elimination. We show its correctness, sketch a completeness proof for a certain class of formulas and show that it can probably lead to an algorithm with only quadratic time complexity, as opposed to the existing algorithm by McCune, which has polynomial time complexity.

Finally, the sixth chapter describes **Scan**, the implementation of the algorithms SCAN and UNSKOLEMIZE and how it is founded on the theorem prover OTTER. It comprises a quick introduction and a user manual.

Chapter 2

Notation

This chapter will gather some basic notions and notational conventions.

2.1 First- and second-order logic

We assume the standard notions such as *term*, *clause*, *formula* etc. of first-order logic to be known, as presented for example in [HET84]. We use a restricted version of second-order logic, which is the subset of second-order logic consisting of formulas not containing predicates as arguments of other predicates. Quantifiers over predicates range over all predicates over a given domain. With Σ_1^1, we denote the set of all second-order formulas Φ which have the following form:

$$\Phi = \exists P_1, \ldots, P_n \phi$$

where ϕ is a formula of first-order logic. Analogously, Π_1^1 denotes the set of second-order formulas Φ of the form

$$\Phi = \forall P_1, \ldots, P_n \phi.$$

Models are written as pairs $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$, where $\mathbf{D}$ denotes a domain and $\mathbf{I}$ an interpretation. Assignments are written as $\mathbf{A}$. We denote the interpretation of a formula ϕ (or a term) by $\phi^{\mathbf{I}}$, with an additional assignment by $\phi^{\mathbf{I},\mathbf{A}}$.

We use lower cases x, y or z for variable names, P, Q or R for predicates, and f, g, h for functions. Constants are designated with lower cases of the beginning of the alphabet: a, b, c, ... We will designate formulas of first-order logic with the Greek lower cases θ, ϕ, ψ, χ.

We abbreviate vectors of of quantifiers by writing $\exists \bar{x} \phi$ for a formula beginning with a prefix of existential quantifiers, i.e., a formula of the form

$\exists x_1 r \ldots \exists x_n \phi$. Similarly, $\forall \bar{y} \phi$ abbreviates a formula starting with a prefix of universal quantifiers, and $\forall\exists \bar{P} \phi$ a formula with a mixed prefix of both universal and existential quantifiers.

We will need a simple complexity measure on formulas that counts the maximum nesting depth of quantifiers:

Definition 2.1.1 (Quantifier rank of a formula) The *quantifier rank* $qr(\phi)$ of a first-order formula ϕ is recursively defined as follows:

- if ϕ is atomic, then $qr(\phi) = 0$.
- if $\phi = \neg\psi$, then $qr(\phi) = qr(\psi)$.
- if $\phi = \psi_1 \circ \psi_2$, for a binary connective $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, then $qr(\phi) = max(qr(\psi_1), qr(\psi_2))$.
- if $\phi = \forall\psi$ or $\phi = \exists\psi$, then $qr(\phi) = qr(\psi) + 1$.

2.2 Miscellaneous

We use substitutions and related notions.

Definition 2.2.1 (Substitution) A *substitution* is a finite set of variable/term pairs $\sigma = \{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n\}$, where $x_i\sigma = t_i$, $1 \leq i \leq n$. The domain of a substitution σ is the set $Dom(\sigma) = \{x \mid x \neq x\sigma\}$.

As usual, substitutions are designated with Greek lower cases σ, μ, ν. We write $t\sigma$ or $\sigma(t)$ to denote the result of applying the substitution σ to t.

For accessing subterms in terms, we define *Occurrences*.

Definition 2.2.2 (Occurrences) An *occurrence* is a word over $\mathbb{N}$. Let ϵ denote the empty word. We first define the *set of occurrences* of terms t and variables v, $Occ(t)$ and $Occ(v)$, then the *set of occurrences* of a formula ϕ.

1. for all variables $v \in \mathcal{V}$, $Occ(v) = \emptyset$.
2. the empty word ϵ is in $Occ(t)$
3. $i.\pi$ is in $Occ(t)$ if $t = f(t_1, \ldots, t_n)$ and $\pi \in Occ(t_i)$.

The *set of occurrences* of a formula ϕ, $Occ(\phi)$ is defined as follows:

1. the empty word ϵ is in $Occ(\phi)$

2. $i.\pi$ is in $Occ(\phi)$ if ϕ is the atomic formula $P(t_1, \ldots, t_n)$, $i \leq n$ and $\pi \in Occ(t_i)$.

3. $i.\pi$ is in $Occ(\phi)$ if $\phi = \psi_1 \wedge \ldots \wedge \psi_n$ or $\phi = \psi_1 \vee \ldots \vee \psi_n$, $i \leq n$ and $\pi \in Occ(\psi_i)$

4. $1.\pi$ $(2.\pi)$ is in $Occ(\phi)$ if $\phi = \forall x\, \psi$ or $\phi = \exists x\, \psi$ or $\phi = \neg\psi$ or $\phi = \psi \Rightarrow \theta$ or $\phi = \psi \Leftrightarrow \theta$ and $\pi \in Occ(\psi)$ $(\pi \in Occ(\theta))$.

Given a term, variable or a formula and a position, we can get a term, variable or a subformula at that position: if $\pi \in Occ(\psi)$ then $\psi|_\epsilon = \psi$ and $\psi|_{i.\pi} = \psi_i|_\pi$ where $\psi = \psi_1 \circ \ldots \circ \psi_n$, $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, and $1 \leq i \leq n$, or $\psi|_{1.\pi} = \phi|_\pi$ if $\psi = \neg\phi$ or $\psi = \forall x\, \phi$ or $\psi = \exists x\, \phi$.

We obtain the set of all occurrences of a term s in a formula ϕ or a term t with $Occ(\phi, s)$ and $Occ(t, s)$.

We write $\phi[\pi/\psi]$ for the formula obtained from ϕ by replacing the formula at the position π by ψ, analogously $t[\pi/s]$ for terms s, t.

Additionally we define the *length* of an occurrence π to be the length of the word π over $\mathbb{N}$.

Chapter 3

General complexity of second-order quantifier elimination

Before we step into the details of practical quantifier elimination in the next chapter, it is worth noting some of the most basic theoretical limitations we have to face. One natural source of difficulties is the expressiveness of second-order logic, making it impossible to find a first-order equivalent for every formula of second-order logic in principle. We will exemplify this by the non-first-order notion of finiteness. Some results about the overall expressiveness of second-order logic will further illustrate its distance and relation to first-order logic.

However, as the next chapter will show, it is well possible to design practically usable algorithms for eliminating second-order quantifiers. Some simple observations will relate their termination properties to the decidability of the fragments of second-order logic they are operating on. In particular, it can be proven by elementary means that there is no algorithm computing first-order- equivalents of second order formulas if they exists and reporting failure otherwise.

3.1 Expressiveness of second-order logic

Second-order logic often allows for natural formulations of concepts that are awkward or impossible to express in first-order logic. For example, take Leibniz' definition of equality which stipulates that two objects be identical if they share the same properties. This can be written as

$$(*)\ \forall x, y\ (\forall P\ (P(x) \Leftrightarrow P(y)) \Rightarrow x = y).$$

When formalizing equality in first-order logic, this quantification over predicates is commonly simulated by an infinite set of formulas (see [Hod84]):

1. $\forall x(x = x)$
2. For every first-order formula ϕ the sentence

$$\forall \bar{z}, x, y(x = y \Rightarrow (\phi \Rightarrow \phi[x/y]))$$

The variable vector $\bar{z}$ contains all the free variables in ϕ, which are assumed to be different from x and y. The formulas in the second set express Leibniz' idea the other way around: if two elements are identical, they can be substituted for each other.

Despite its size, this version only approximates the intended semantics, as it still allows non-standard models of identity that the second-order formulation excludes, cf. [Hod84]. However, it is true in all models with standard identity and logically implies all sentences that are true in such models, which is good enough for many purposes, e.g. automated theorem proving.

Other concepts are not definable in first-order logic at all. A seemingly simple one is finiteness: There is no set of first-order sentences that is verified precisely by those models that have a finite domain. This follows from the compactness property of first-order logic, see [vBD84]. For a second-order characterization, one may use Dedekind's criterion that every injective mapping from a finite set to itself if already surjective. Injectivity and surjectivity are readily formalized in first-order logic, and then a formula of the form

$$\forall f(f \text{ is injective} \Rightarrow f \text{ is surjective})$$

guarantees the desired finiteness.

This expressiveness is reflected in the theoretical complexity of second-order logic. It lacks several model-theoretic properties of first-order logic. We mention two that have been chosen by Lindström ([Lin69]) for characterizing first-order logic (see below): compactness and upward Löwenheim-Skolem (for this and more, see [vBD84], for definitions of these properties [Hod84]).

Deduction also becomes much harder. Even in the fragment of Σ_1^1 formulas, universal validity is not an arithmetical notion. Therefore, the set of universally valid Σ_1^1 formulas is not recursively enumerable. Van Benthem and Doets show this in [vBD84], where they also measure the complexity of formulas with arbitrary prefixes. As a consequence, universal validity of second-order formulas can not be axiomatized, as any complete axiomatization together with an effective set of inference rules would again permit a simple enumeration procedure for the valid formulas.

3.2 Characterizing first-order logic

The question naturally arises how we can tell whether we are in the realm of first-order logic. Van Benthem and Doets ([vBD84]) mention essentially three main theorems that have been used in the past to distinguish first-order from second-order properties.

These theorems characterize first-order logic semantically. They do not directly provide methods for automatically computing first-order equivalents for second-order formulas. Also, they employ model theoretic machinery which is beyond the scope of this work. However, as there can be no algorithm that automatically decides whether a second-order formula has a first-order equivalent for *all* second-order formulas (see next section), they are indispensable for deciding the remaining cases. Furthermore, they allow us to identify whole classes of second-orderformulas that have first-order equivalents, rather than only single formulas.

To begin with, the aforementioned famous theorem by Lindström answers our question for extensions of first-order logic:

Theorem 3.2.1 (Lindström) *Let $\mathcal{L}$ be any extension with the two properties:*

(a) (Downward Löwenheim-Skolem) If a sentence ϕ of $\mathcal{L}$ has an infinite model then it has a model with at most countably many elements.

(b) Either (Upward Löwenheim-Skolem) if a sentence of $\mathcal{L}$ has an infinite model, then it has one with uncountably many elements; or (Compactness) if Δ is a theory in $\mathcal{L}$ such that every finite set of sentences from Δ has a model then Δ has a model.

Then every sentence of $\mathcal{L}$ has exactly the same models as some first-order sentence.

So, instead of any logic with the above properties, we can take first-order logic. However, this does not give a direct answer to the question whether a given property Φ of models is first-order definable. To apply Lindström's theorem in this case, we would have to construct a logic in which ϕ is represented in the language and which satisfies Lindström's requirements for logics, as sketched in [vBD84]. Then we could check whether this logic has the above properties.

A theorem by Keisler ([Kei71]) gives a more directly applicable characterization of first-order definable properties. It relies on special models, called ultra products (cf. [BS71]). In addition, we need the notion of elementary equivalence: Two models M_1, M_2 are called *elementary equivalent*, written $M_1 \equiv M_2$, if they satisfy the same set of first-order sentences.

Theorem 3.2.2 (Keisler) *A property Φ can be defined by a single first-order sentence iff both Φ and its complement are closed under the formation of isomorphs and ultra products.*

However, the ultra product construction is quite abstract. Another characterization of first-order logic by Fraissé ([Fra55]) is of a more combinatorial nature. To state his theorem, we specialize the notion of elementary equivalence of two models. Two models M_1, M_2 are called *n-elementary equivalent*, written $M_1 \equiv_n M_2$ if they satisfy the same set of first-order sentences with quantifier rank smaller or equal to n. We call a property Φ of models *closed under n-elementary equivalence* if for two n-elementary equivalent models M_1 and M_2 $M_1 \in \Phi$ implies $M_2 \in \Phi$.

Theorem 3.2.3 (Fraissé) *A property Φ can be defined by a single first-order sentence iff it is closed under n-elementary equivalence for some natural number n.*

In contrast to the purely semantical characterizations of Keisler and Lindström, the notion of n-elementary equivalence gives a syntactic criterion for the possible complexity of formulas which subdivides the formulas of first-order logic into equivalence classes. This sometimes allows for proofs that conspicuously exhibit structural properties of models that first-order logic is unable to express, cf. [HET84].

3.3 Computational complexity of reduction

Using the above theorems always requires special proof ideas. They do not provide a uniform, let alone mechanizeable method for deciding whether a given second-order sentence ϕ has a first-order equivalent. The following theorem of Van Benthem and Doets implies that such a method does not exist ([vBD84]):

Theorem 3.3.1 (van Benthem, Doets) *The set of Π^1_1 sentences that have a first-order equivalent is not recursively enumerable.*

It follows directly that there cannot be an algorithm that decides whether a Π^1_1 sentence can be reduced to a first-order one. For, given such an algorithm, we could recursively enumerate the set of Π^1_1 sentences and filter those that are not reducible, which results in an enumeration contradicting the theorem. So we have

Proposition 3.3.2 *There is no algorithm which decides reducibility for Π^1_1 sentences, and therefore no such algorithm for all second-order sentences.*

Another observation allows us to establish corresponding results for Σ^1_1 sentences.

Lemma 3.3.1 *Let ϕ be a second-order formula. Then ϕ is reducible iff $\neg\phi$ is reducible.*

Proof. Let ψ be a first-order equivalent for ϕ, that is $\psi \equiv \phi$. Then $\neg\psi \equiv \neg\phi$, $\neg\psi$ first-order, so $\neg\phi$ is reducible. $\square$

Proposition 3.3.3 *There is no algorithm which decides reducibility for Σ^1_1 sentences.*

Proof. We show that such an algorithm could be used to decide reducibility of Π^1_1 sentences, which contradicts proposition 3.3.2.

Let $\phi = \forall\bar{P}\phi' \in \Pi^1_1$, ϕ' a first-order sentence, be given. Then

$$\neg\phi = \neg\forall\bar{P}\ \phi' \equiv \exists\bar{P}\ \neg\phi' =: \varphi \in \Sigma^1_1.$$

By the above lemma 3.3.1, ϕ is reducible iff $\neg\phi$ is reducible. So if we could decide reducibility of any $\phi \in \Pi^1_1$, so we can for $\varphi \in \Pi^1_1$, contradicting proposition 3.3.2. $\square$

Proposition 3.3.4 *The set of Σ^1_1 that have a first-order equivalent is not recursively enumerable.*

Proof. As argued above, an enumeration algorithm could be transformed into an algorithm that decides reducibility. □

Our main concern, however, is not to decide reducibility, but to compute equivalents. More precisely, given a fragment Ω of second-order logic, we would like to have an algorithm $\mathcal{RED}$ which always terminates on the input $\phi \in \Omega$ and outputs

$$\mathcal{RED}(\phi) = \begin{cases} \psi & \text{with } \psi \Leftrightarrow \phi \text{ and } \psi \text{ first-order, if } \phi \text{ is reducible} \\ \textbf{failure} & \text{otherwise} \end{cases}$$

We call such an algorithm *complete on* Ω. Such an algorithm can obviously be used to decide reducibility, so it cannot exist if Ω is full second-order logic. We may relax the termination property by allowing the algorithm to run into an infinite loop on non-reducible formulas instead of terminating and reporting failure, obtaining a reduction algorithm $\mathcal{PRED}$ with

$$\mathcal{PRED}(\phi) = \begin{cases} \psi & \text{with } \psi \Leftrightarrow \phi \text{ and } \psi \text{ first-order, if } \phi \text{ is reducible} \\ \text{undefined} & \text{otherwise} \end{cases}$$

We call $\mathcal{PRED}$ *semi-complete on* Ω. Though this algorithm is less powerful than $\mathcal{RED}$, we could still enumerate the set of reducible formulas $\phi \in \Omega$ as follows:

For every natural number n, we enumerate the first n formulas $\phi_i \in \Omega$ and let the algorithm work for a limited time, say n seconds, on each formula. If it stops with the output ψ, we add ψ to our list. If a formula $\phi \in \Omega$ is reducible, the algorithm will eventually add its equivalent as n increases.

We sum this up in

Proposition 3.3.5 *Let Ω be a fragment of second-order logic, $\mathcal{PRED}$ a semi-complete reduction algorithm Ω. Then the set of reducible formulas $\phi \in \Omega$ is recursively enumerable.*

Using the above results about non-enumerable fragments of second-order logic, we can deduce

Proposition 3.3.6 *There is no semi-complete reduction algorithm for the fragments Π^1_1 and Σ^1_1 of second-order logic.*

Proof. By proposition 3.3.5, the existence of this algorithm would imply that the sets of reducible Π_1^1 and Σ_1^1 formulas would be recursively enumerable. But this contradicts proposition 3.3.1 and 3.3.3, respectively. □

For full second-order logic, the consequence is

Corollary 3.3.7 *There is no semi-complete reduction algorithm for second-order logic, i.e., an algorithm which stops on all reducible second-order sentences and outputs an equivalent first-order formula.*

Another simple enumeration argument can also show how decidability and reducibility of a fragment of second-order logic are connected in one direction.

Proposition 3.3.8 *Let Ω be a recursively enumerable fragment of second-order logic. If Ω is decidable, then the set of formulas in Ω that have a first-order equivalent in Ω is recursively enumerable.*

Proof. Note that we can decide equivalence of two formulas ϕ, ψ in Ω if we can decide tautologies, as ϕ and ψ are equivalent if and only if $\phi \Leftrightarrow \psi$ is a tautology. Now we proceed as above: For every natural number n, we enumerate the first n second-order formulas in Ω. For each of these formulas ϕ, we again enumerate the first n first-order formulas in Ω and check them for equivalence with ϕ by running our decision algorithm for a limited time of n seconds. If an first-order equivalent of ϕ is found, ϕ is printed out.

Now if a formula ϕ has an equivalent ψ in Ω with number m in the enumeration order, the equivalence can be decided in a limited time of n seconds. The above algorithm will find ψ after max(n,m) seconds. □

Note that the algorithm used in the proof can easily be modified to print out the reducible formulas together with their first-order equivalent as tuples. Then, for a given formula, we can reduce it to first-order logic by simply waiting for it to be printed out. If all formulas in Ω are reducible, we only have to wait for a finite time. Thus, for such fragments, we have a reduction algorithm. We will show in 3.4.3 that all propositional formulas with existentially quantified predicates can be reduced to propositional formulas without quantifiers. Therefore, for the set of these formulas, we have a - hopelessly ineffective - reduction algorithm. We will give a simple alternative reduction algorithm for this special case in the proof for 3.4.3.

Now for the converse. How can a reduction algorithm help to decide whether a formula is a tautology? To make the following results directly applicable to the SCAN algorithm to be developed in chapter 4, we look at the equivalent problem of deciding whether a formula is satisfiable.

We begin with a simple observation:

Lemma 3.3.2 *Let ϕ be a formula of second-order logic, P a vector of predicate symbols. Then ϕ is satisfiable iff $\exists \bar{P}\phi$ is satisfiable.*

Proof. We prove this for the case that $\bar{P}$ is a single variable. The lemma for longer vectors follows by iterated application of this case.

"$\Rightarrow$": If ϕ is satisfiable, then there is a model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ of ϕ and an assignment $\mathbf{A}$ with $\phi^{\mathbf{I},\mathbf{A}} = \mathbf{true}$. Assume that $P^{\mathbf{I}} = \mathbf{P}$. Then $\mathbf{I} = \mathbf{I}[P/\mathbf{P}]$, and consequently $\phi^{\mathbf{I},\mathbf{A}} = \phi^{\mathbf{I}[P/\mathbf{P}],\mathbf{A}} = \mathbf{true}$. So there exists a predicate $\mathbf{P}$ over the domain such that $\phi^{\mathbf{I}[P/\mathbf{P}],\mathbf{A}} = \mathbf{true}$. Then, by definition of the semantics of second-order logic, $\mathbf{M}$ satisfies $\exists P\phi$.

"$\Leftarrow$": If $\exists P\phi$ is satisfiable,there is a model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ of ϕ and an assignment $\mathbf{A}$ with $\phi^{\mathbf{I},\mathbf{A}} = \mathbf{true}$. So there exists a predicate $\mathbf{P}$ over the domain such that $\phi^{\mathbf{I}[P/\mathbf{P}],\mathbf{A}} = \mathbf{true}$. So the model $\langle \mathbf{D}, \mathbf{I}[P/\mathbf{P}] \rangle$ satisfies ϕ with the assignment $\mathbf{A}$. □

In the following, we will extensively use the fact that the truth value of the formula $\exists P\phi$ is independent of the interpretation of P. Intuitively, to decide satisfiability of a sentence ϕ, we first existentially quantify all the predicates it contains, which preserves satisfiability. We then eliminate all these predicates and get an equivalent formula whose satisfiability is easy to decide. A step in this direction is

Lemma 3.3.3 *Let ϕ be a sentence of second-order logic which contains only predicates. Let $\bar{P}$ be a vector of these predicates. Then ϕ is satisfiable iff $\exists \bar{P}\phi$ is a tautology.*

Proof. "$\Rightarrow$": If ϕ is satisfiable, then so is $\exists \bar{P}\phi$, by lemma 3.3.2. But the truth value $\exists \bar{P}\phi$ is independent of the interpretations of variables and predicates it contains, so if one model satisfies ϕ, then every model satisfies it.

"$\Leftarrow$": If $\exists \bar{P}\phi$ is a tautology, it is satisfiable, so by lemma 3.3.2 ϕ is satisfiable, too. □

To make the following more readable, we define a normalization algorithm $\mathcal{NORM}$ that will be used to eliminate redundant predicates that may be introduced during reductions. Let $fp(\phi)$ be the set of free predicates in ϕ. For arbitrary second-order formulas ψ and ϕ, define $\mathcal{NORM}(\phi, \psi)$ to be the second-order formula that results from replacing every atom $P(\bar{t})$ in ψ with $P \notin fp(\phi)$ by $\top$. The property of $\mathcal{NORM}$ crucial for our purpose is:

Lemma 3.3.4 *If $\phi \equiv \psi$, then $\phi \equiv \mathcal{NORM}(\phi, \psi)$.*

Proof. (sketched) Let P be a predicate symbol in ψ that is not in ϕ. Then the truth value of ϕ does not depend on the interpretation of P. But then the truth value of ψ cannot depend on the interpretation of P, either, as ψ and ϕ are equivalent. So we can interpret P as constantly true, for example, and replace P by $\top$. $\square$

One more point of notation before the next theorem: If Ω is a fragment of first-order logic, let $\exists\Omega$ be the smallest fragment of second-order logic such that for every $\phi \in \Omega$, $\exists\bar{P}\phi \in \exists\Omega$.

Theorem 3.3.9 *Let Ω be a set of first-order logic sentences without function symbols. Then a reduction algorithm that is complete on $\exists\Omega$ can be used as a decision algorithm for Ω.*

Proof. Define $\mathcal{EVAL}$ to be an algorithm that takes first-order formulas ϕ as input whose only atomic formulas are $\top$ and $\bot$ and outputs

$$\mathcal{EVAL}(\phi) = \begin{cases} \top & \text{if } \phi \text{ is a tautology} \\ \bot & \text{if } \phi \text{ is a contradiction} \end{cases}$$

Such formulas are either tautologies or contradiction, so $\mathcal{EVAL}$ is well-defined and total. Obviously, $\phi \equiv \mathcal{EVAL}(\phi)$.
In addition, define $\mathcal{CLOSE}$ to be an algorithm which computes $\mathcal{CLOSE}(\phi) = \exists\bar{P}\phi$ where $\bar{P}$ is a variable vector consisting of the free variables in ϕ.

Now assume that $\mathcal{RED}$ is a reduction algorithm that is complete on $\exists\Omega$. Let ϕ be an arbitrary first-order formula with $\mathcal{CLOSE}(\phi) = \exists\bar{P}\phi$. As $\exists\bar{P}\phi$ is second-order closed, it is either a tautology or a contradiction, so it has a first-order equivalent and $\mathcal{RED}(\exists\bar{P}\phi)$ returns a first-order formula. Let $\mathcal{RED}(\phi) = \psi$. Observe that ϕ contains no free predicates, so $\mathcal{NORM}(\exists\bar{P}\phi, \psi)$ returns a sentence that can only contain $\top$ and $\bot$ as atomic formulas. Therefore, $\mathcal{EVAL}$ can be applied to it.

We claim that

$$\phi \in \Omega \text{ is satisfiable iff } \mathcal{NORMRED}(\phi) = \mathcal{EVAL}(\mathcal{NORM}(\exists\bar{P}\phi, \psi)) = \top.$$

We have $\exists\bar{P}\phi \equiv \psi = \mathcal{RED}(\exists\bar{P}\phi)$ by the specification of $\mathcal{RED}$. By lemma 3.3.4, $\psi \equiv \mathcal{NORM}(\exists\bar{P}\phi, \psi)$. Together with the definition of $\mathcal{EVAL}$, we have $\exists\bar{P}\phi \equiv \mathcal{NORMRED}(\phi) = \mathcal{EVAL}(\mathcal{NORM}(\exists\bar{P}\phi, \psi))$. Especially, $\exists\bar{P}\phi$ is a tautology iff $\mathcal{NORMRED}(\phi)$ is a tautology iff $\mathcal{NORMRED}(\phi) = \top$.

Now by lemma 3.3.3, ϕ is satisfiable iff $\exists\bar{P}\phi$ is a tautology iff $\mathcal{NORMRED}(\phi) = \top$, by the above.

So $\mathcal{NORMRED}$ would be an algorithm for deciding satisfiability in Ω.
□

This immediately implies

Corollary 3.3.10 *Let Ω be an undecidable fragment of first-order logic. Then there can be no reduction algorithm that is complete on $\exists\Omega$.*

Proof. By theorem 3.3.9, a reduction algorithm that is complete on $\exists\Omega$ can decide Ω, which is a contradiction. □

We mention one instantiation of this theorem. Let Ω contain all first-order formulas. As first-order logic is undecidable (see, for example [HET84]), there can be no reduction algorithm that is complete on $\exists\Omega$, and as $\Omega \subset \Sigma^1_1$, there can be no such algorithm for Σ^1_1.

By examining the proof of theorem 3.3.9, we can obtain a complexity result for reduction algorithms. Let $\|\phi\|$ denote the length of a formula in some representation.

Corollary 3.3.11 *Let Ω be a decidable set of first-order sentences without function symbols, and RED a reduction algorithm complete on $\exists\Omega$. Let $\mathcal{RED}(\phi)$ run for at most $t_{\mathcal{RED}}(\|\phi\|)$ deterministic time on every second-order closed formula ϕ. If $\|\mathcal{RED}(\phi)\| \leq sp_{\mathcal{RED}}(\|\phi\|)$ for all second-order closed formulas ϕ, where $sp_{\mathcal{RED}}$ is a polynomial, satisfiability of sentences in Ω can be decided in $\mathcal{O}(t_{\mathcal{RED}}(p(\|\phi\|)) + q(\|\phi\|))$ time, where q and p are polynomials.*

Proof. We show that the algorithm

$$\mathcal{NORMRED} = \mathcal{EVAL}(\mathcal{NORM}(\mathcal{CLOSE}(\phi), \mathcal{RED}(\mathcal{CLOSE}(\phi))))$$

used in the proof of theorem 3.3.9 meets this complexity requirement. We calculate the time consumption for each computation step and sum up at the end.

We assume $t_{\mathcal{RED}}$, $sp_{\mathcal{RED}}$, and each of the time and space bounds given below w.l.o.g. to be monotonous and of course positive polynomials.

Let ϕ be an input formula of length n. It is easy to design $\mathcal{CLOSE}$ in such a way that it produces after polynomial time $t_{\mathcal{CLOSE}}(n)$ the output $\exists \bar{P}\phi$ which has a length of at most $sp_{\mathcal{CLOSE}}(n)$, $sp_{\mathcal{CLOSE}}$ a polynomial in any reasonable representation of formulas.

Let $\mathcal{RED}(\exists \bar{P}\phi) = \psi$, a result computed after $t_{\mathcal{RED}}(sp_{\mathcal{CLOSE}}(n))$. By assumption, and by monotonicity of $sp_{\mathcal{RED}}$,

$$\|\psi\| \leq sp_{\mathcal{RED}}\|\exists \bar{P}\phi\|) \leq sp_{\mathcal{RED}}(sp_{\mathcal{CLOSE}}(n)).$$

$\mathcal{NORM}(\exists \bar{P}\phi, \psi)$ can also compute its result φ in polynomial time when input the two formulas, say $t_{\mathcal{NORM}}(\|\exists \bar{P}\phi\| + \|\psi\|)$. φ will usually be the size of ψ, still we may give it polynomial space $sp_{\mathcal{NORM}}(\|\exists \bar{P}\phi\| + \|\psi\|)$. Now, by monotonicity and the above,

$$\begin{aligned} t_{\mathcal{NORM}}(\|\exists \bar{P}\phi\| + \|\psi\|) &\leq t_{\mathcal{NORM}}(sp_{\mathcal{CLOSE}}(n) + sp_{\mathcal{RED}}(sp_{\mathcal{CLOSE}}(n))) \\ &=: uptb_{\mathcal{NORM}}(n) \end{aligned}$$

with $uptb_{\mathcal{NORM}}$ a polynomial (as a composition of polynomials) giving an upper time bound for $upb_{\mathcal{NORM}}$.

Similarly the output length $\|\varphi\|$ of $\mathcal{NORM}$ is bounded by

$$\begin{aligned} \|\varphi\| &\leq sp_{\mathcal{NORM}}(\|\exists \bar{P}\phi\| + \|\psi\|) \\ &\leq sp_{\mathcal{NORM}}(sp_{\mathcal{CLOSE}}(n) + sp_{\mathcal{RED}}(sp_{\mathcal{CLOSE}}(n))) \\ &=: upsb_{\mathcal{NORM}}(n) \end{aligned}$$

by monotonicity and the above inequalities for $\|\exists \bar{P}\phi\|$ and $\|\psi\|$.

The job of $\mathcal{EVAL}$ can be done in linear time, we bound it by the polynomial $t_{\mathcal{EVAL}}$. On φ, it runs for $t_{\mathcal{EVAL}}(\|\varphi\|) \leq t_{\mathcal{EVAL}}(upsb_{\mathcal{NORM}}(n))$.

As total running time we have

$$\begin{aligned} & t_{\mathcal{NORMRED}}(n) \\ = \;& t_{\mathcal{CLOSE}}(\|\phi\|) + t_{\mathcal{RED}}(\|\exists \bar{P}\phi\|) + t_{\mathcal{NORM}}(\|\exists \bar{P}\phi, \psi\|) + t_{\mathcal{EVAL}}(\|\varphi\|) \\ \leq \;& t_{\mathcal{CLOSE}}(n) + t_{\mathcal{RED}}(sp_{\mathcal{CLOSE}}(n)) \\ & + uptb_{\mathcal{NORM}}(n) + t_{\mathcal{EVAL}}(upsb_{\mathcal{NORM}}(n)) \\ = \;& t_{\mathcal{RED}}(sp_{\mathcal{CLOSE}}(n)) + t_{\mathcal{CLOSE}}(n) \\ & + uptb_{\mathcal{NORM}}(n) + t_{\mathcal{EVAL}}(upsb_{\mathcal{NORM}}(n)) \\ = \;& t_{\mathcal{RED}}(p(n)) + q(n) \end{aligned}$$

for $p \stackrel{\text{def}}{=} sp_{\mathcal{CLOSE}}$ and $q \stackrel{\text{def}}{=} t_{\mathcal{CLOSE}} + uptb_{\mathcal{NORM}} + t_{\mathcal{EVAL}} \circ upsb_{\mathcal{NORM}}$. p and q are sums of compositions of polynomials and therefore polynomials themselves. This proves our claim. □

The seemingly strange condition on the complexity of the output of $\mathcal{RED}$ is necessary for the polynomial time transformation of a satisfiability problem into a reduction problem. This in turn allows for the following consideration. If $t_{\mathcal{RED}}$,p and q are polynomials, then so is $t_{\mathcal{RED}} \circ p + q$. That is, if formulas in $\exists\Omega$ can be reduced in polynomial time, satisfiability in Ω can be decided in polynomial time. But, taking Ω to be the set of propositional formulas, a polynomial time reduction algorithm would imply a polynomial time decision algorithm for satisfiability of propositional formulas. Such an algorithm has not been found until today. Its existence would prove $\mathcal{P} = \mathcal{NP}$. This clearly shows the depth and complexity of problems involved in the development of reduction algorithms.

3.4 Reducible fragments of second-order logic

In this section, we list some reducible fragments of second-order logic that have a simple syntactic form. For one of them, the fragment of quantified propositional formulas in Σ_1^1, we give a simple reduction algorithm.

Theorem 3.4.1 *Let ϕ be a quantifier free formula of first-order logic. Then*

1. *all formulas of the form $\exists\bar{P}\exists\bar{y}\ \phi(\bar{P})$ are reducible to a single first-order sentence.*

2. *all formulas of the form $\exists\bar{P}\forall\bar{y}\ \phi(\bar{P})$ are reducible to a set of possibly infinite disjunctions of first-order formulas.*

Proof. Both statements are proven in [vBD84]. □

To see the coverage of these fragments, it is useful to consider the following normal form, which is basically skolem normal form with skolem functions replaced by existentially quantified predicates:

Proposition 3.4.2 *For any formula $\phi \in \Sigma_1^1$ of second-order logic, an equivalent formula of the following form can effectively be found:*

$$\exists\bar{P}\forall\bar{x}\exists\bar{y}\ \psi(\bar{P},\bar{x},\bar{y})$$

where ψ is quantifier free. If all predicates in ϕ have zero arity (i.e., they are propositional variables), ψ can be chosen as a formula of propositional logic.

Proof. See [vB83]. $\square$

Together with the following lemma, this will allow any easy proof of the reducibility of Σ_1^1 formulas with only 0-place predicates.

Lemma 3.4.1 *Let $\phi(P)$ be a formula of second-order logic, and P a 0-place predicate. Then*

$$\exists P \phi(P) \equiv \phi(\top) \vee \phi(\bot)$$

where $\phi(\top)$ ($\phi(\bot)$) is the formula that results from replacing all occurrences of P in ϕ by $\top$ ($\bot$).

Proof. By structural induction. $\square$

Theorem 3.4.3 *Any formula $\phi \in \Sigma_1^1$ in which all occurring predicates are 0-place can effectively be reduced to a formula of propositional logic.*

Proof. We assume w.l.o.g. that ϕ is in the normal form of proposition 3.4.2, i.e., ϕ is given as

$$\exists P_1, \ldots, P_n \; \forall \bar{x} \exists \bar{y} \; \psi(\bar{P}, \bar{x}, \bar{y})$$

with ψ a quantifier free formula of propositional logic. Then there can be no free first-order variables in ψ, so we eliminate the first order quantifiers, resulting in

$$\exists P_1, \ldots, P_n \psi(\bar{P}, \bar{x}, \bar{y}).$$

Now we successively apply lemma 3.4.1: we replace the subformula $\exists P \;\; \psi(\bar{P}, \bar{x}, \bar{y})$, by the equivalent formula without P obtained by lemma 3.4.1. The same is done for the other quantified predicates, until we have a propositional formula without quantifiers that is equivalent to ϕ. $\square$

Chapter 4

SCAN: Elimination of existentially quantified predicates

In this chapter, we will formulate a special quantifier elimination problem and present the SCAN algorithm as a possible solution to this problem. Several extensions of the algorithm will improve its termination properties.

4.1 The basic SCAN algorithm

The SCAN algorithm tackles the following problem: given a second-order formula $\Phi(\bar{P}) = \exists \bar{P}\phi$, ϕ a first-order formula, find a (possibly infinite) set ψ of first-order formulas such that

$$\exists \bar{P}\phi \equiv \psi.$$

As is shown in [Ack35a], [Ack35b], this problem is solvable. Indeed, SCAN is able to solve it, as the algorithm always generates a set ψ with the above properties. However, SCAN does not always find a finite ψ when there is one. This would make SCAN a semi-complete reduction algorithm for the fragment Σ_1^1 of second-order logic as defined in section 3.3. By proposition 3.3.6, such an algorithm cannot exist.

4.1.1 A basic version

The main idea of SCAN is quite simple: SCAN generates a set of first-order consequences of a formula that is strong enough to imply the formula itself. This approach can be motivated by a simple observation. Assume we are given a formula $\Phi(\bar{P})$ and want to find an equivalent formula set ψ with $\Phi(\bar{P}) \equiv \psi$ as above. If ψ exists, then it can be found in the set of all first-order consequences of $\Phi(\bar{P})$, that is, the set $\{\psi \mid \psi \models \psi$ and ψ is a first-order formula$\}$ This set in turn of course implies $\exists \bar{P}$, for it contains ψ, and $\psi \models \Phi(\bar{P})$. So we may try to solve the problem by computing the set of first-order consequences of $\Phi(\bar{P})$ and hope that they are equivalent to this formula. However, as this set is usually infinite and full of redundancies, we should be content with a, hopefully finite, representation with few redundancies.

4.1.1.1 Preliminaries

Techniques for drawing consequences from clause sets without too much redundancies and removing redundancies from the set have been developed in the area of first-order theorem proving, and to some extent, they can be carried over to the problem of second-order quantifier elimination. In particular, the well-known resolution rule, invented by Robinson ([Rob65]), can be modified to yield the so-called C-resolution used in SCAN to draw consequences from clauses. The C-resolution inference rule (C for constraint) has the following form:

Definition 4.1.1 (C-resolution) *Let* $C = P(s_1, \ldots, s_n)\} \vee C'$ *and* $D = P(t_1, \ldots, t_n)\} \vee D'$ *be two clauses with disjoint variable sets. Then the rule*

$$\frac{P(s_1, \ldots, s_n) \vee C' \quad \neg P(t_1, \ldots, t_n) \vee D'}{C' \vee D' \vee s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n}$$

is called C-resolution rule*.*

We denote the *C-resolvent* by $CRes(C, P(s_1, \ldots, s_n), D, P(t_1, \ldots, t_n))$. Note that C-resolution can always be performed on complementary literals, whereas ordinary resolution requires the literals to be unifiable.

Classical factorization is modified to C-factorization analogously:

Definition 4.1.2 (C-factorization)
Let $C = P(s_1, \ldots, s_n) \vee P(t_1, \ldots, t_n) \vee C'$*. Then the rule*

$$\frac{P(s_1, \ldots, s_n) \vee P(t_1, \ldots, t_n) \vee C'}{P(s_1, \ldots, s_n) \vee C' \vee s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n}$$

is called C-factorization rule*.*

A *C-factor* of C is denoted by $CFac(C, P(s_1, \ldots, s_n), P(t_1, \ldots, t_n))$.

Some more definitions will abbreviate standard applications of these inference rules to clause sets. They extend the rules from predicates to clauses to sets on each argument.

Dealing with quantifier elimination, we will often be given a set or a vector of predicate variables $\bar{P}$. Again, we will avoid notational overhead by treating vectors as sets and vice versa, e.g. we will write "$P' \in \bar{P}$" instead of "P' is a component of the vector $\bar{P}$".

For a clause C, we let $CFac(C)$ denote the set of all factors obtained by one application of factorization on predicates in $\bar{P}$. Formally,

$$CFac(C, \bar{P}) \stackrel{\text{def}}{=} \{CFac(C, P'(\bar{t}), P'(\bar{s})) \mid C = P'(\bar{t}) \vee P'(\bar{s}) \vee R \text{ for some } P' \in \bar{P}\}.$$

The straightforward extension to clause sets M is

$$CFac(M) \stackrel{\text{def}}{=} \bigcup_{C \in M} CFac(C).$$

We need the closure of a clause set under factorization with respect to a vector of predicates $\bar{P}$. To this end, define

$$CFac^1(C, \bar{P}) \stackrel{\text{def}}{=} CFac(C, \bar{P}) \text{ and } CFac^{i+1}(C, \bar{P}) \stackrel{\text{def}}{=} CFac(CFac^i(C, \bar{P})).$$

Then

$$CFac^\infty(C, \bar{P}) \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}} CFac^i(C, \bar{P})$$

computes all recursive factors of a clause. Finally, for a clause set M,

$$CFac^\infty(M, \bar{P}) \stackrel{\text{def}}{=} \bigcup_{C \in M} CFac^\infty(C, \bar{P})$$

is the closure of M under factorization.

Similar notions are defined for C-resolution. For two clauses C and D, let $CRes(C, D)$ denote the set of all resolvents computed by a single application of the C-resolution rule on C and D. Extended to clause sets M,

$$CRes(M) \stackrel{\text{def}}{=} \bigcup_{C, D \in M} CRes(C, D).$$

Canonically, $CRes^1(M) \stackrel{\text{def}}{=} CRes(M)$, $CRes^{i+1} \stackrel{\text{def}}{=} CRes(CRes^i(M))$. Slightly different from the C-Factorization case, $CRes^1_\Sigma(M) \stackrel{\text{def}}{=} CRes(M)$,

$CRes^{i+1} \stackrel{\text{def}}{=} CRes(CRes^i(M)) \cup CRes^i(M)$. Now the closure is formed by $CRes^\infty(M) \stackrel{\text{def}}{=} \{C \mid C \in CRes^i_\Sigma(M) \text{ for some } i \in \mathbb{N}\}$. Then we set

$$CRes^\infty(M) \stackrel{\text{def}}{=} \bigcup_{C \in M} CRes^i(C).$$

An important operation will resolve one literal L in a clause $C = L \vee C'$ with all complementary literals in a clause set M. We define $CRes(C, L, M) \stackrel{\text{def}}{=} \{CRes(C, L, D, \neg L) \mid D = \neg L \vee D'\}$, $CRes^1_\Sigma(C, L, M) \stackrel{\text{def}}{=} CRes(C, L, M)$, $CRes^{i+1}_\Sigma(C, L, M) \stackrel{\text{def}}{=} CRes(C, L, CRes^i_\Sigma(C, L, M)) \cup CRes^i_\Sigma(C, L, M)$, and then

$$CRes^\infty(C, L, M) \quad \stackrel{\text{def}}{=} \quad \{C \mid C \in CRes^i_\Sigma(C, L, M) \text{ for some } i \in \mathbb{N}\}.$$

At last, we formally define the closure $CInf^\infty$ of a clause set under the C-inference rules C-factorization and C-resolution on predicate symbols from a vector. Note that we exclude self-resolution of clauses. Let $CInf^\infty(M, \bar{P})$ be the smallest set S such that $C_1[P(\bar{t}), C_2[\neg P(\bar{s})] \in S$, $C_1 \neq C_2$ implies $CRes(C_1, P(\bar{t}), C_2, \neg P(\bar{s})) \in S$, and $C[P(\bar{t}), P(\bar{s})] \in S$ implies $CFac(C, P(\bar{t}), P(\bar{s})) \in S$.

4.1.1.2 The algorithm

Now we are prepared for a defining and explaining a first version of the SCAN algorithm. It takes as an input a second-order formula $\exists \bar{P}\phi$ and outputs, if it terminates, a first-order or second-order formula ψ with $\psi \equiv \exists \bar{P}\phi$.

In the following, given some variable vector $\bar{P}$, we say that a clause C is a *$\bar{P}$-clause* if it contains a predicate $P' \in \bar{P}$. See figure 4.1 for the algorithm. The deletion of a clause C with a literal L from a clause set $M = C \cup R$ that has been resolved with the rest of the clause set R is called *purity deletion*, the literal itself and the clause C are called *pure*. Note that, to be pure, a literal is not required to be resolved with literals in the clause it is contained in. In other words, self-resolution of clauses in SCAN is not necessary, though correct.

The algorithm can be subdivided into three phases:

1. Normalization: ϕ is converted into a clause set $M = CNF(\phi)$, possibly by introducing skolem functions
2. Elimination: M is transformed into a new clause set M' not containing the existentially quantified predicates

Algorithm 4.1.3 (Basic SCAN)

formula SCAN(**formula** $\exists\bar{P}\phi$)
set<**clause**> M;
Let $M = CNF(\phi)$;
While (There is some $\bar{P}$-clause $C \in M$){
// Nondeterministically generate all C-factors and C-resolvents of a particular literal
  If (There are two distinct clauses $C[P'(\bar{t})], D[P'(\bar{s})]$ such that
   $CRes(C, P'(\bar{t}), D, \neg P'(\bar{s})) \notin M$)
  then Let $M = M \cup CRes(C, P'(\bar{t}), D, \neg P'(\bar{s}))$;
  If (There is a clause $C[P'(\bar{t}), P'(\bar{s})]$ such that
   $CFac(C, P'(\bar{t}), P'(\bar{s})) \notin M$)
  then Let $M = M \cup CFac(C, P'(\bar{t}), P'(\bar{s}))$;
  // If all resolvents with a literal have been generated, delete clause
  If $(C[P'(\bar{t})] \in M, M = C \cup R, (CRes^{\infty}(C, P'(\bar{t}), R) \cup CFac^{\infty}(C, \bar{P})) \subset M)$
  then Let $M = R$; }
// Eliminate skolem functions
Let $\psi = Usk(M)$;
Return(ψ);

Figure 4.1: The basic SCAN algorithm

3. Unskolemization: M' is transformed into a formula ψ without skolem functions

Each phase deserves some explanation, especially about how the normalization and unskolemization steps preserve equivalence.

Normalization The normalization operation is standard in many first-order theorem provers. There, it is commonly used as an satisfiability preserving operation, i.e., ϕ is satisfiable iff $CNF(\phi)$ is satisfiable. However, many algorithms for computing clause normal form do something stronger, namely, they compute $CNF(\phi)$ such that $\phi \equiv \exists\bar{f}\forall\bar{x}\ CNF(\phi)$, where $\bar{f}$ is the vector consisting of all the skolem functions introduced for generating the normal form, and $\bar{x}$ consists of all free variables in the clause set. In chapter 5, this interpretation is justified and examined more closely. We assume that $CNF(\cdot)$ satisfies the above equivalence, which implies

$$\exists\bar{P}\phi \equiv \exists\bar{P}\exists\bar{f}\forall\bar{x}\ CNF(\phi) = \exists\bar{P}\exists\bar{f}\forall\bar{x}\ M.$$

Elimination The elimination process does the main work of the algorithm. Note that by the CNF transformation, the clause set now contains the existentially quantified predicates of ϕ without explicit quantification, so that C-resolution and C-factorization can be applied. This may be seen as a kind of skolemization of predicates. In essence, the elimination step computes the closure of the clause set under C-resolution and C-factorization and afterwards deletes all clauses not containing any of the existentially quantified predicates. If the closure is finite, then the second step terminates, and we have

$$\exists \bar{P} \exists \bar{f} \forall \bar{x}\ M \equiv \exists \bar{P} \exists \bar{f} \forall \bar{x}\ M' \equiv \exists \bar{f} \forall \bar{x}\ M'.$$

The second equivalence holds because M' does not contain any clauses with predicates in $\bar{P}$.

Unskolemization The last phase now gets a clause set M' with $\exists \bar{f} \forall \bar{x}\ M' \equiv \exists \bar{P} \phi$, $\bar{f}$ containing all skolem functions. To get a regular first-order formula without skolem functions, we have to turn them back into existential quantifiers. E.g., let f be a skolem function and assume that the output of the elimination phase is $M = \{G(f(x), x)\}$. We interpret this clause as the formula $\exists f \forall x\ G(f(x), x)$, which can be seen as a the result of skolemizing $\forall x \exists z\ G(z, x)$. We call this process of reversing skolemization *unskolemization*. Chapter 5 deals with this problem.

We will use $\textsc{Scan}(\Phi)$ to denote the result of SCAN applied to a second-order formula Φ. In addition, we will need to talk about the elimination phase of SCAN alone, which gets as input a clause set (resulting from the CNF transformation of Φ) and outputs a clause set, without subsequent unskolemization. The result of this phase is $\textsc{Scan}(M)$, where M is a clause set. We take this set to be defined, even if SCAN does not terminate. In this case, $\textsc{Scan}(M)$ is the set that SCAN enumerates. Note that this set does not contain any clause with an existentially quantified predicate, as all such clauses are pure and therefore deleted.

An example illustrates how the algorithm works. Let

$$\Phi = \exists P \phi = \exists P \exists z_1, z_2\ (Q(z_1) \Rightarrow P(z_1)) \wedge (P(z_2) \Rightarrow Q(z_2)).$$

We translate the implication connectives and turn the existential quantifiers into skolem constants a and b to obtain the clause set

$$M \stackrel{\text{def}}{=} CNF(\phi) = \{C_1 \stackrel{\text{def}}{=} \neg Q(a) \vee P(a), C_2 \stackrel{\text{def}}{=} \neg P(b) \vee Q(b)\}.$$

One C-resolution of C_1 with C_2 yields

$$C_3 \stackrel{\text{def}}{=} CRes(C_1, P(a), C_2, \neg P(b)) = Q(a) \vee Q(b) \vee a \neq b$$

C_1 is now pure and can be removed from M, leaving $M = \{C_2, C_3\}$. There are no further resolution possibilities for the literal $\neg P(b) \in C_2$ with literals in M, so it is pure, and we can delete C_2 from M,too. Now $M = \{C_3\}$.

Finally, we unskolemize M. Let

$$\psi = Usk(\exists a, b\ Q(a)) = \exists z_1, z_2\ Q(z_1) \vee Q(z_2) \vee z_1 \neq z_2,$$

where z_1, z_2 are first-order variables. To sum it up, we have the following chain of equivalences:

	$\exists P\phi$	
$\equiv$	$\exists P\exists a, b\ M$	by properties of CNF
$\equiv$	$\exists P\exists a, b\ C_3$	by correctness of SCAN
$\equiv$	$\exists a, b\ C_3$	as P has no occurrence in C_3
$\equiv$	$\exists z_1, z_2\ Q(z_1) \vee Q(z_2) \vee z_1 \neq z_2$	as $\exists a, b\ C_3 \equiv Usk(\exists a, b\ C_3)$

4.1.2 Properties of the basic algorithm

The SCAN algorithm is proven correct and, with a generalized notion of soundness explained below, also sound in [GO92]. Correctness of SCAN guarantees that every output formula of SCAN is equivalent to the input formula, in short SCAN$(\exists\bar{P}\phi) \equiv \exists\bar{P}\phi$.

Completeness of SCAN would imply that the algorithm would be capable of computing a first-order equivalent for all input formulas. This is simply impossible because such an equivalent does not exist for every formula in the considered input class, c.f. section 3.1. A relaxed condition is to require an equivalent formula for those cases where it exists. But this is an requirement that would make SCAN a semi-complete reduction algorithm as defined in 3.3 for all formulas of the form $\exists\bar{P}\phi$. However, these formulas make up the fragment Σ^1_1 of second-order logic, and proposition 3.3.6 states that a semi-complete reduction algorithm cannot exist for this fragment.

We can escape these negative results by taking as the output of SCAN the set M of clauses that it enumerates, ignoring whether SCAN terminates or not. Recall that we denoted this set by SCAN(M). In [GO92] the authors deduce from results of Ackermann ([Ack35a],[Ack35b]) that the set of clauses in the closure that do not contain any of the existentially quantified variables is equivalent to the clause normal form of the input formula. Therefore, SCAN is complete in the sense that, on any input clause set, it enumerates a clause set equivalent to the initial clause set which does not contain predicate variables.

More formally, let $\bar{P}$ be the vector of existentially quantified predicates in a clause set M. Then

$$\exists \bar{P} \forall \bar{x}\ M \equiv \exists \bar{P} \forall \bar{x}\ \textsc{Scan}(M) \equiv \forall \bar{x}\ \textsc{Scan}(M).$$

Keeping in mind the theoretical limitations discussed above, this is in some sense optimal.

Note, however, that the set $\textsc{Scan}(M)$ does not represent a first-order formula as long as it contains skolem functions, which it usually does. Therefore, to obtain a first-order formula, we also have to eliminate those skolem functions. So it depends heavily on the final unskolemization step, whether SCAN finally outputs a first-order formula. Not surprisingly, there can be no unskolemization algorithm that transforms any SCAN output clause set into am equivalent first-order formula, or even an equivalent set of first-order formulas. This would again imply that there exists a first-order equivalent for every SCAN input formula $\Phi \in \Sigma_1^1$, which is not true, as discussed in section 3.1.

We sum this up in

Theorem 4.1.4 (Gabbay, Ohlbach) *The* SCAN *algorithm is correct and complete in the following sense: Let M be a clause set, interpreted as a formula $\exists \bar{P} \exists \bar{f} \forall \bar{x}\ M$. Then there exists a (possibly infinite) clause set M' such that $\exists \bar{P} \exists \bar{f} \forall \bar{x}\ M \equiv \exists \bar{P} \exists \bar{f} \forall \bar{x}\ M'$, and* SCAN *enumerates such a clause set, i.e.,*

$$\exists \bar{P} \exists \bar{f} \forall \bar{x}\ M \equiv \exists \bar{f} \forall \bar{x}\ \textsc{Scan}(M).$$

The core of the proof is the correctness proof of the following purity deletion rule.

Lemma 4.1.1 (Gabbay, Ohlbach) *Let M be a clause set, $C \in M$ a clause and $L \in C$ a literal with $L = P$ or $L = \neg P$ for some predicate symbol P that is interpreted as existentially quantified. If M contains all C-factors of C and all C-resolvents of C with some other clause in M on L, then C is called* pure*, and removing C from M preserves equivalence. Formally,*

$$\exists P\ M \equiv \exists P\ (M \setminus C) \cup CRes(C, L, M) \cup CFac^{\infty}(\{C\}, \{P\})$$

The purity deletion criterion, which demands the presence of all C-factors and C-resolvents on a selected literal will play an important role in the next section, where it will be used for proving the correctness of the extended SCAN algorithm.

4.2 Extensions of the basic algorithm

Using the SCAN algorithm, we have to deal with problems well known from automated theorem proving: exponentially growing clause sets, generation of redundant clauses, non-termination of the procedure and many more. Fortunately, some techniques that have been developed in the field of automated theorem proving for handling those problems can be carried over to our problem of quantifier elimination. As theorem provers are usually based on a calculus, there are two straightforward ways to improve them:

- by adding new inference rules
- by changing the inference strategy

We will extend SCAN mainly by adding new inference rules and do not explore inference strategies. For discussing the new rules, we will even fix an inference strategy, which is similar to the one used in the implementation of SCAN. However, the new rules cannot be arbitrarily combined with the purity deletion rule, which will impose restrictions on possible inference strategies.

We separated the presentation of the extended SCAN algorithm and the sketch of its correctness proof (section 4.2.3), as the proof is technically awkward and does not provide much intuitive insight.

4.2.1 Simplification rules

All simplification rules presented in this section transform clause sets into equivalent clause sets which are smaller. In general, this makes a clause set more readable, and, as a computational advantage, also saves space, which are both desirable effects. However, our primary intention is to use simplification rules for making SCAN terminate on a larger class of second-order formulas than it does in its basic version by avoiding loops. This is crucial, as SCAN may loop on even very simple input formulas:

Example 4.2.1 *Consider the following clause set:*

1. $\neg P(x) \vee P(f(x)) \vee R(f(x))$
2. $P(a)$
3. $R(f(a)))$

Assume we choose the literal $\neg P(x)$ in clause 1 as the literal to purify. It has no C-factors and so we start by resolving it with clause 2, resulting in clause 4: $P(f(x)) \vee R(f(x)) \vee x \neq a$, which we replace by the equivalent $P(f(a)) \vee R(f(a))$ (Intuitively correct as it seems, this step must be justified, which is explained and done below). We choose this clause as the new resolution partner and obtain clause 5: $P(f(f(x)) \vee R(f(f(x))) \vee x \neq a \vee R(f(a))$, again simplifying this to $P(f(f(a))) \vee R(f(f(a))) \vee R(f(a))$. If we continue this way, always resolving with the last resolvent, we get the $P(f^n(a)) \vee \bigcup_{i=1}^{n} R(f^i(a))$ as clause number $3+n$, and get caught in an infinite loop. Of course we might avoid this loop by starting with another clause as the pure clause. But on more complex clause sets, things might not be so obvious. In fact, there can be no 'perfect' inference strategy for SCAN that avoids all loops simply for the reason that no second-order quantifier algorithm is complete and inevitably loops on some input formulas. In this special case, we can alternatively apply a simplification rule: Note that clause 3 $R(f(a))$ subsumes the first resolvent. We will show in section 4.2.3 that, in this case, we do not have to generate the C-resolvents of a resolvent that is subsumed by another, so we may stop and delete clause 1, which is now pure in a wider sense. It is not pure in the sense of the definition, as not all resolvents of clause 1 with clause 2 are present, though they are implied, as they all would contain the literal $R(f(a))$ of clause 3 and therefore be subsumed.

In the above example, it is clear that deletion of subsumed clauses preserves equivalence of a clause set. It is not at all obvious, however, that the deletion of clause 1 is equivalence preserving, because it is not pure in the sense of lemma 4.1.1. This is possible because the C-resolvents of the subsuming clause (none in this case) will subsume the C-resolvents of the subsumed clause, which is proven in section 4.2.3. So the C-resolvents of the subsumed clause are "virtually" present and we may still delete the pure clause - provided that the C-resolvents of the subsuming clause have been generated. Therefore, given an equivalence-preserving transformation rule for clause sets, we face the question how it interplays with the purity deletion rule, that is, which C-resolvents have still to be generated after the transformation.

The example also shows the hallmark of loops: recursive clauses, that is, clauses containing the literal selected for purification and its negation. It is easy to see that only the purification of recursive clauses leads to loops. Consequently, we will try to eliminate them.

We will now describe equivalence preserving transformation rules for clauses and clause sets and place them in a new extended SCAN algorithm, interplaying with the purity deletion rule.

4.2.1.1 Equivalence preserving transformation rules on clauses

C-resolution and c-factoring produce a lot of inequalities which can easily be eliminated by the following proposition.

Proposition 4.2.2 *For any first-order clause ϕ and term t, where t does not contain x,*

$$\forall \bar{y}\ \phi_{\{x \mapsto t\}} \equiv \forall x \forall \bar{y}\ \phi \vee x \neq t$$

in all standard models of equality, where y is a variables vector containing all variables in ϕ and t.

For an arbitrary term t, we call a constraint $x \neq t$ non-recursive if $t = x$ or t does not contain x. We use the above proposition to define an operation that eliminates all non-recursive constraints in a clause.

Definition 4.2.3 (Constraint elimination) *Let C be a clause. Then we define $CtEl(C)$ to be a clause that results from eliminating all non-recursive constraints from C by repeatedly applying proposition 4.2.2.*

From proposition 4.2.2 it is obvious that

$$C \equiv CtEl(C).$$

This constraint simplification makes clauses more readable and facilitates subsumption on clauses. For example, $P(a)$ does not subsume $C \stackrel{\text{def}}{=} P(x) \vee x \neq a$, while it trivially does subsume $P(a) = CtEl(C)$.

Another simplification rule is *condensing*, known from the field of theorem proving, a combination of standard factoring and subsumption that shortens clauses. We again assume a linear order on the set of clauses to obtain uniqueness in the following definition.

Definition 4.2.4 (Condensing) *Let C be a clause, F be a factor of C obtained by an arbitrary number of binary factorizations with the following properties:*

- *$F \trianglelefteq C$*
- *there is no factor of C with less literals which subsumes C*

- *in the linear order on clauses, F is the smallest of all factors which have the above two properties.*

We define

$$Cond(C) \stackrel{\text{def}}{=} F.$$

By correctness of factorization, it is clear that

$$C \models F = Cond(C),$$

and by the properties of subsumption we have

$$F \models Cond(C),$$

combining to $C \equiv Cond(C)$.

Note that we do not restrict factorization to quantified predicates. Syntactically, condensing can be done by instantiating a clause and deleting all those literals that become redundant by a subsumption. For example, $Cond(A(x) \vee A(b) \vee R) = A(b) \vee R$.

4.2.1.2 Equivalence-preserving transformation rules on clause sets

If M is a clause set and C a clause such that $M \models C$, then of course $M \cup C \equiv M$. In other words, we may delete clauses that are implied by the rest of the clause set. As two special cases, we may delete subsumed clauses, which are implied by a single other clause, and tautologies, which are implied by the empty clause set. These special cases have the pleasant property that they are decidable on first-order clause sets, while implication is only semi-decidable. Therefore, in any real-world implementation of SCAN, we can only approximate the full deletion rule by trying to prove clauses in a limited time or with limited space. Otherwise, we might reintroduce the infinite loops that we wanted to avoid.

A combination of resolution and subsumption leads to the unit-deletion rule.

Definition 4.2.5 *Let M be a clause set, C a clause. Let U be the set of all unit clauses in M matching some literal in C:*

$$\begin{aligned} U \quad = \quad & \{D \in M \mid D = L_D \textit{ for some literal } L, \\ & \textit{and } \nu L_D = \neg L_C \textit{ for some literal } L_C \in C \textit{ and some matcher } \nu\} \end{aligned}$$

Let S the subset of all literals in C that are matched by some literal in U. We define

$$UnitDel(C, M) = C \setminus S.$$

Proposition 4.2.6 *For a clause set M and a clause C, we have*

$$M \cup C \equiv M \cup UnitDel(C, M).$$

Proof. (Sketch) Let $D \in M$ be a unit $D = L_D$ such that $\nu L_D = \neg L_C$ for some literal $L_C \in C = L_C \vee R$ and some matcher ν. We resolve (standard resolution) on L_D and L_C using the matcher as the most general unifier, and obtain the resolvent R, which back subsumes C, because the matcher does not instantiate variables in C. Deletion of subsumed clauses preserves equivalence, and by repeated resolutions with all units in U, we arrive at the equivalent clause set $M \cup UnitDel(C, M)$. □

The next simplification is very powerful, as it can eliminate a quantified predicate from a clause set completely in one step. It can be seen as an application of the purity deletion rule with a certain inference strategy on clause sets with a special structure. Therefore, it is labeled *extended purity rule*. To define and prove it, it is useful to define the notion of *polarity* of a clause. Let P be a predicate. We say that a clause C containing occurrences of P has *positive polarity* in P if it only contains positive occurrences of P, and *negative polarity* if it only contains negative occurrences of P. C has *mixed polarity* in P if P occurs both positively and negatively, and *neutral polarity* in P if P does not occur in C at all. Define

$$Pol(P, C)) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } P \text{ occurs in } C \text{, and } C \text{ has positive polarity in } P \\ 0 & \text{if } P \text{ has neutral or mixed polarity in } C \\ -1 & \text{if } P \text{ occurs in } C \text{, and } C \text{ has negative polarity in } P \end{cases}$$

Two clauses C, D have *opposite polarity* in P if $Pol(P, C) \cdot Pol(P, D) = 1$. For the definition, recall that $\overline{proj}_P(M)$ denotes the set of all clauses in M not containing the predicate symbol P.

Definition 4.2.7 *Let M be a clause set and P a predicate symbol. We say that P is* polarity-pure *or* p-pure *in M if it does not contain two clauses that have opposite polarity in P. We define*

$$ExtPurify(P, M) = \begin{cases} \overline{proj}_M(P) & \textit{if } P \textit{ is p-pure in } M \\ M & \textit{otherwise} \end{cases}$$

Proposition 4.2.8 *For any clause set N and predicate symbol P,*

$$\exists P \forall \bar{x}\ N \equiv \exists P \forall \bar{x}\ ExtPurify(P, N)$$

Proof. This is obvious in case that P is not p-pure, as nothing changes. Assume that P is p-pure. It is clear that

$$\forall \bar{x}\ N \equiv \forall \bar{x}\ \overline{proj}_N(P),$$

so that by monotonicity of the formula $\exists P \forall \bar{x} \phi$ we conclude

$$\exists P \forall \bar{x}\ N \equiv \exists P \forall \bar{x}\ ExtPurify(P, N).$$

For the other direction, we assume that N contains only clauses C with $Pol(P, C) = 1$ or $Pol(P, C) = 0$. The case $Pol(P, C) = -1$ or $Pol(P, C) = 0$ is alike. Now assume we are given a model $\langle \mathbf{D}, \mathbf{I} \rangle$ for $\overline{proj}_N(P)$. We have to show that in this model there is an interpretation of P such that $\mathbf{M} \models \forall \bar{x} N$. We choose $P^{\mathbf{I},\mathbf{A}} = \mathbf{D}$, that is, P is constantly interpreted as true. It is obvious that this interpretation meets the requirement, which completes the proof. □

With these simplification rules at hand, we may define an improved version of SCAN.

4.2.2 The extended SCAN algorithm

The extended SCAN algorithm XSCAN uses all simplification rules presented in the last section. It also removes some non-determinism of the basic version by employing a simple inference strategy:clauses are purified successively. That is, a clause and a literal within this clause is selected, all C-factors of this clause and C-resolutions with other clauses are generated, and the clause is deleted before C-resolutions with or C-factors of other clauses are generated.

We already pointed out that simplification rules relieve us from building C-resolutions with certain clauses. We will attach marks to clauses that must be resolved against the selected clause by the function $mark()$, and test them for marks with the boolean function $marked()$. Marks are removed with $unmark()$, and we assume them to disappear if the clause they mark is deleted. These functions are extended to sets by member-wise application. See figure 4.2 for the algorithm. The procedure **Process** contains all simplification rules but the extended purity rule. It appends the simplified clause to the clause set given as parameter, if it is not redundant. If the

Algorithm 4.2.9 (Extended SCAN)

formula XSCAN(**formula** $\exists \bar{P}\phi$)
set<**clause**> M;
Let $M = CNF(\phi)$;
// While there are clauses with quantified predicates
For (All $P \in \bar{P}$)
  $M = ExtPurify(P, M)$;
While (there is a $\bar{P}$-clause $PureCl$){
  Let $M = M \setminus PureCl$;
  Select a literal $PureLit \in PureCl$ built from a quantified predicate P;
  Let $M = M \cup CFac^{\infty}(C,P)$;
  // mark all resolution parters
  For (All clauses $D \in M$ with $\neg PureLit \in D$)
    $mark(D)$;
  While (There is a clause D with $marked(D) = \mathbf{true}$){
      For (All resolvents $Res = CRes(PureCl, PureLit, D, L')$) **Do** {
        // check Res for redundancy, simplify and mark
        // see below for procedure **Process** (Res, M, PureCl)
        Process$(Res, M, PureCl)$; }
    // Purify if predicate is extended pure
    $M = ExtPurify(P, M)$; }
};
// Eliminate skolem functions
Let $\psi = Usk(M)$;
Return(ψ);

Figure 4.2: The extended SCAN algorithm

clause $PureCl$ is back subsumed by one of its C-resolutions, all clauses in M are unmarked, and the purification of $PureCl$ thus stopped.

Algorithm 4.2.10 (Process)

> **Process**(**clause** Res, **set<clause>** M, **clause** $PureCl$)
> // Simplify resolvent
> $Res = UnitDel(Res, M)$;
> $Res = Cond(Res)$;
> // Append Res to M if it is not redundant
> **If** (Res is not a tautology and not $M \models Res$)
> $\quad M = M \cup Res$;
> // Collect clauses that are (back) subsumed by Res
> **Let** $B = \{C \in M \mid Res, C \trianglelefteq\}$;
> **Let** $M = M \setminus B$;
> **If** ($Res \trianglelefteq PureCl$)
> $\quad unmark(M)$;
> **Return**;

4.2.3 Properties of the extended version

Due to lack of time, I could not give a proof of the correctness of XSCAN. However, I try to sketch how such a proof might look like, and what pitfalls one has to avoid. For simplicity, we will use a modified version of the main loop of XSCAN, which has slightly different marking and uses labeled clauses.

4.2.4 Modeling XSCAN inference

We model the XSCAN algorithm as follows. We will investigate the situation that a pure clause *pure* is given and successively resolved with a clause set M, and that simplification rules are applied. Now we look at the algorithm in figure 4.3: Note that we directly add newly generated clauses to the clause set, and then apply simplifications to the clause set, instead of adding them when they could not be proven redundant. This can model even further extensions of the XSCAN algorithm. We denote the marking function by α for notational convenience. All clauses that have not been resolved upon carry a mark, written $\alpha(C) = 1$ for marked clauses. We assume for simplicity that *all* unresolved clauses carry marks, as opposed to the situation in XSCAN, which marks only possible resolution partners with literals complementary to the pure literal. $\alpha(M)$ denotes the set of all clauses in M that are marked, $\overline{\alpha}(M)$ the complement. Note that all of

1. Mark all clauses in M.
2. While there are marked clauses:

   Non-deterministically, either
   - Pick a marked clause C and add all its C-resolvents R with $pure$ to M, unmark C, mark R
   - or apply a simplification rule to M

Figure 4.3: A model for XSCAN

the following could be transferred to the case of a literal marking function, which would allow for finer control.

As one parent in each resolution step always is $pure$, we write $CRes(C)$ for $CRes(pure, C)$, similarly for other resolution functions. Then by $X(M)$, given our marking function α, we denote the set $\overline{\alpha}(M) \cup CRes(\alpha(M))$. $X(M)$ denotes the set that XSCAN would compute if there were no further simplifications. We write X_α if we want to emphasize that X depends on a marking function.

We assume that all clauses that have been generated by a C-resolution step are labeled with their parent clause (which was resolved with $pure$), which distinguishes clauses with the same set of literals by their parents. That is, strictly speaking, we operate on tuples, or on proof trees, and have to lift, for example, subsumption to this structure. However, we hope that the following is clear without these formal details.

4.2.5 A loop invariant

We want to show that after each transformation of the set M into a set M', we have

$$X_\alpha(M) \equiv X_{\alpha'}(M'),$$

with a new marking function α'. If the algorithm terminates with some M and α, $\alpha(M)$ must be empty, and we would have $X(M) = \overline{\alpha}(M)$, which by a chain of equivalences leads to the closure of the original clause set M_0, since, for M_0, $X(M_0) = CRes(M)$, because all clauses in M_0 are marked. It then remains to be shown how this simulates XSCAN. The main point is that XSCAN cannot distinguish between two clauses that have the same set of literals but are derived from different parents, whereas we can do this using the labels. Hence, a bijective mapping from the set of generated labeled clauses to the set of clauses does not necessarily exist. However, we can use the subsumption elimination rule to remove copies of clauses with different labels, which modulo some technical details makes the set of labeled "homomorphic" to the set of clauses in XSCAN.

4.2.6 Studying subsumption simplification

The difficult part is of course to show that simplification observes the above equivalence. To begin with, it is easy to prove that

Proposition 4.2.11 *$M \equiv M'$ implies $CRes(M) \equiv CRes(M')$.*

We can apply this equivalence to our problem in some cases. For example,if we only transform the set of marked clauses equivalently: With $M = \overline{\alpha}(M) \cup \alpha(M)$ and $M' = \overline{\alpha}(M') \cup \alpha(M')$, and assuming that $\overline{\alpha}(M) = \overline{\alpha'}(M')$ and $\alpha(M) \equiv \overline{\alpha'}(M')$, we have essentially substituted with an equivalent, and it is clear that $X_\alpha(M) \equiv X_{\alpha'}(M')$.

However, if we only know $M \equiv M'$, we have to think of a different approach. In the following, we will examine the equivalence preserving transformation that removes subsumed clauses from a set, and sketch how this can be extended to the case of removing implied clauses.

So assume that we have two clauses $C_1 \in \overline{\alpha}(M)$, $C_2 \in \alpha(M)$, $C_1 \trianglelefteq C_2$. We transform M into $M' = \overline{\alpha}(M) \cup \alpha(M) \setminus C$ by removing C_2, and want to show (the marking function α is the same for both sets)

$$X(M) \equiv X(M').$$

A crucial lemma is

Lemma 4.2.1 *If $C_1 \trianglelefteq C_2$, then $CRes(C_1) \trianglelefteq CRes(C_2)$.*

Proof. Special case of 4.2.2. □

Hence, if we guarantee that $X(M')$ contains (or implies) all C-resolvents of C_1, then all C-resolvents of C_2 are also implied. However, if we delete C_2, we might loose some of the C-resolvents of C_1, for example, if C_2 has been derived from C_1. To study this problem more closely, we will first visualize a state of the deduction process as a directed graph as in figure 4.4. Nodes of the graph represent clauses, and edges (C_1, C_2) represent the inference relation $C_1 = CRes(C_2)$. The additional arrows labeled A and B represent subsumptions. The figure depicts the following situation:

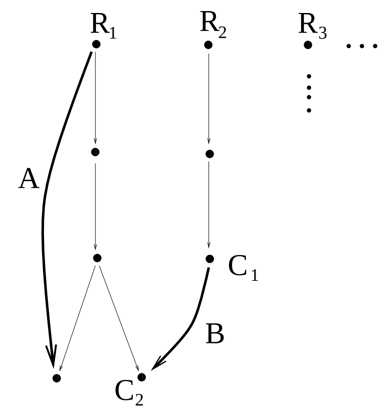

Figure 4.4: Visualization of a deduction state

We assume that we have for some time added resolvents and removed marks, but not done any simplifications. Note that no two equal clauses have been generated, which would have triggered subsumption simplification. Then the graph we defined is a set of trees, and all marked clauses are the leaves of the trees, since all inner nodes have descendants and therefore lost their marks. The roots of the trees represent the members R_i of the initial clause set $M_0 \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} R_i$, and its closure $CRes(M_0)$ can be evidently be computed by computing their closures separately:

$$CRes(M_0) = \bigcup_{i=1}^{n} CRes(R_i).$$

Denoting the ancestors and descendant of a clause C by $Anc(C)$ and $Dc(C)$, respectively, the figure should provide the intuition for $CRes(R_i) = X(Dc(R_i))$, and therefore with $M = \bigcup_{i=1}^{n} R_i$, $CRes(M_0) = X(M)$.

Now we can compare the subsumptions represented by the arrows A and B. The subsumption that arrow A visualizes literally cuts the branch on which our argumentation sits: the subsumed clause is marked, and therefore maybe necessary for generating resolvents, especially of the subsuming clause.

The case of C_1 and C_2 in the figure is different. The C-resolvents of both clauses can be computed independently, and the lemma 4.2.1 above can be applied. Let

$$R_r = \bigcup_{i=3}^{n} R_i,$$

and note that, for any sets M, N, $CRes(M \cup N) = CRes(M) \cup CRes(N)$. With the latter, and since C_2 is marked, we obtain from the definition of X that

$$X(R_2) = X(Dc(R_2) \setminus C_2) \cup CRes(C_2). \tag{4.1}$$

The lemma yields, together with $CRes(C_1) \subseteq CRes(R_1) \subseteq X(Dc(R_1))$,

$$X(Dc(R_1)) \trianglelefteq CRes(C_2) \tag{4.2}$$

which we can use in

$$\begin{aligned} X(M) &= CRes(M_0) \\ &= CRes(R_1) \cup CRes(R_2) \cup CRes(R_r) \\ &= X(Dc(R_1)) \cup X(Dc(R_2)) \cup X(Dc(R_r)) \\ &= X(Dc(R_1)) \cup X(Dc(R_2) \setminus C_2) \cup CRes(C_2) \cup X(Dc(R_r)) \\ &\equiv X(Dc(R_1)) \cup X(Dc(R_2) \setminus C_2) \cup X(Dc(R_r)) \quad \text{by 4.2} \\ &= X(M') \quad \text{by definition of } M' \end{aligned}$$

This proves our claim. Note that, as long as we have trees, we can define roots as above, and the argument is no longer limited to the assertion that all clauses in the input clause set still exists, which may not be true when using, for example, back subsumption.

The argument may be generalized. It is not necessary that C_1, C_2 have different root nodes. It suffices that C_2 is not a descendant of C_1. This criterion can be generalized to the case of extended subsumption, where $S \models C_2$ for some set of clauses S. Then we require that C_2 is not among the descendants of any clause in S. Of course, for a similar argument, we need a parallel to lemma 4.2.1. This is given by

Lemma 4.2.2 *If $M \models C$, then $CRes(M) \trianglelefteq CRes(C)$.*

Proof. (Sketch) We prove this using the resolution calculus. A well-known property of this calculus is that if $M \models C$, then we can deduce some C' subsuming C. We prove by induction that

$$CRes^i(pure, M) \vdash CRes^i(pure, C'),$$

that is, the i-th layer of C-resolvents of C can be deduced from the i-th layer of C-resolvents. The closure $CRes(M)$ then implies the closure $CRes(C')$, which by lemma 4.2.1 then subsumes $CRes(C)$. □

However, the subsuming clauses are now "supporting" the subsumed clause in the sense that their closure must be computed in order to compute the closure of the subsumed clause (strictly speaking, this may not be necessary, but it is necessary for the line of argumentation given above to work). Further simplification rules must care not to delete supporting clauses without replacing them with another supporting clause. As long as all simplification rules respect the support structure, the above argumentation works. The critical case is when a clause is used to subsume a clause set by which it is supported. If we picture the support dependencies in a clause graph with added arrows for support relations, then one critical case are cycles, which occur when a clause can be used to simplify the set it is supported by.

With this method, we can only prove the correctness of XSCAN in its current form if we can prove that these cycles do not destroy it. In the case of simple subsumption, this presents no problem, while it seems more complicated in the case of extended subsumption. Yet, we have only treated the problem on a very abstract level, leaving aside a lot of information. For example, on the semantical side, we only use first-order equivalence arguments, though we have existentially quantified predicates. Additionally, on the syntactical side, we can examine where the existentially quantified predicates occur, and what role they play, for example, in a proof used by the extended subsumption rule. Hence, we believe it is justified to conjecture that XSCAN is correct in its current form.

One further note on the proof method. We think that a proof based on the above method will be technically complicated. There could be a lot of support from the theory of connection graph resolution (e.g. [Kow75]), which may also yield additional simplification rules. By its compact representation of potentially infinite resolution branches, it may provide a suitable theoretical means for reasoning about our problems. However, I see no way of entirely getting rid of syntactic arguments as long as the purity deletion criterion has a syntactic form.

4.2.7 Future work

Loops are a natural main issue in research on SCAN. One direction of research might focus on patterns in clause sets generated by loops. With fixpoint methods as used in [NS95], these infinite sets can then reduced to finite ones.

Another direction of research should explore how loops can be avoided by clever combination of inference rules and inference strategies. A point of interest is whether simplification rules merely make the clause set smaller, or really avoid loops. For example, a subsuming clause in many cases can have infinitely many resolvents when the subsumed clauses has, and nothing maybe won. And of course loops may also be terminated by introducing redundancy instead of removing it.

SCAN may prove a useful tool for studying second-order logic with quantifiers ranging over *all* functions from a deductive point of view. Furthermore, the close resemblance of the C-resolution rule and ordinary resolution gives rise to many speculations. For example, one might speculate that any complete calculus for some logic, especially modal logics, might be extended to a complete quantifier elimination algorithm for this logic (in the sense in that SCAN is complete, that is, equivalent clause sets are enumerated). This offers a different view on completeness proofs for first-order logic and other logics. Maybe there are "lowering" techniques similar to the well-known lifting technique. Also, a study of the relation between both resolution rules may gradually abstract from their syntactical form, and hence lead to a deeper and more semantical understanding of their properties.

Although most of these points are already of interest in first-order logic, they now have a much wider applicability.

Chapter 5

Unskolemize: Elimination of existentially quantified functions

In the last chapter, we presented Scan as an algorithm for eliminating existentially quantified predicate symbols from second-order formulas. Scan works on clauses and therefore eliminates first-order existential quantifiers by skolemization. The introduced skolem functions may still occur in Scan's output, so their elimination poses a new quantifier elimination problem, which we will discuss in this chapter and tackle with the algorithm Unskolemize.

We will show that skolemization may be understood as a transformation of first-order existential quantifiers into second-order existential quantifiers. By reversing this transformation, second-order quantification over functions can be reduced to first-order quantification over individuals. The reversal of skolemization is a much less general approach to function elimination than the Scan approach is to predicate elimination. However, contrary to Scan, the Unskolemize algorithm, which is based on this approach, always terminates. Our algorithm improves an existing algorithm by McCune ([McC88]) in both scope and time complexity.

5.1 The problem of skolem functions in Scan

Consider the simple Scan output

$$P(x, f(x))$$

that contains the skolem function f before f is eliminated. We already remarked in chapter 4 that this must be interpreted as the second-order formula

$$\exists f \forall x \; P(x, f(x)).$$

Therefore, SCAN has rather transformed a predicate elimination problem into a function elimination problem than solved a reduction problem. Consequently, we should feed back the new problem of eliminating f to SCAN by coding f as a predicate F with functional properties. That is, we exploit

$$\begin{array}{rc} & \exists f \forall x \; P(x, f(x)) \\ \equiv & \exists f[\forall x, y \; (P(x, y) \vee y \neq f(x))] \\ \equiv & \exists F[\forall x, y \; (P(x, y) \vee \neg F(x, y)) \wedge \forall x \exists z F(x, z)] \end{array}$$

with $F(x, y)$ true iff $f(x) = y$. For F to be a function, we should have required that *exactly one* x for each x with $F(x, y)$ exists, but for the syntactical issue we are up to, this is not relevant. When we input this formula to SCAN, it is put into clause normal form and skolemized yielding two clauses:

C_1 $P(x, y) \vee \neg F(x, y)$

C_2 $F(x, f(x))$

with a new skolem function f. SCAN resolves C_1 with C_2 on F, which puts f back into the place where it annoyed us:

$$C_3 : P(x, y) \vee y \neq f(x) \equiv P(x, f(x)).$$

It is clear that this approach fails in general, too. We now sketch an alternative approach, which is described in more detail in the subsequent sections.

5.2 Reversing skolemization as function elimination

Skolemization may be seen as a transformation of a first-order formula into a second-order one, by applying the following equivalence (see, for example, [vB83]) from right to left:

$$\forall \bar{x} \exists y \Phi[\bar{x}, y] \equiv \exists f \forall \bar{x} \Phi[\bar{x}, y \mapsto f(\bar{x})] \tag{5.1}$$

where $\bar{x}, y$ are free in Φ, and f does not occur in $\Phi[\bar{x}, y]$.

For skolemization of a first-order formula, all existential quantifiers are eliminated by repeated application of this rewrite rule. If we skolemize a formula in prenex form, then the argument sets of the introduced skolem functions are linearly ordered by set inclusion, which can easily be seen.

Now we apply 5.1 in the other direction to eliminate f, as in the example above. Therefore, following McCune ([McC88]), we also call the elimination of existentially quantified functions *unskolemization*. We will now examine the conditions that allow us to unskolemize a formula with several skolem functions.

We assume the formula to be in the form of SCAN output, that is, it is a clause set $M = \{C_1, C_2, \ldots, C_n\}$, interpreted as the formula

$$\exists \bar{f} \forall \bar{x}\ C_1 \wedge C_2 \ldots \wedge C_n$$

where the C_i are clauses, and $\bar{f}, \bar{x}$ contain all skolem functions and free variables in M.

Note that a clause set M to be unskolemized must be in the form $\mathbf{M'}[y \mapsto f(\bar{x})]$ for some clause set M', that is, terms and subterms in M with the top symbol f must have the same argument lists, and they must only consist of variables. We say that f is required to be *consistent* in M. We say that M is consistent if all skolem functions in M are consistent.

In addition, we noted that in a skolemized prenex formula, all argument sets of skolem functions are linearly ordered by set inclusion. If a clause set satisfies this condition, we say that it is *globally linear*, or short *linear*.

Putting this together, we can unskolemize a clause set if it is both linear and consistent. In the following, we try to establish these conditions in a clause set by equivalence preserving transformations. The transformations we employ are

- variable abstraction and
- variable renaming in clauses.

Variable renaming is a well-known equivalence preserving operation on clauses:

Definition 5.2.1 (Renaming) *A substitution σ is a* (variable) renaming *for a clause C if the restriction of σ on the variables of C is injective, that is*

$$\forall x, y \in Vars(C),\ x \neq y \textit{ implies } \sigma(x) \neq \sigma(y).$$

σ is a renaming for a set M of clauses if it is a renaming for each clause.

Variable abstraction is the replacement of terms by variables similar to the constraint elimination introduced in chapter 4. E.g., we use equivalences like

$$\forall x\ P(f(a)) \vee Q(g(x,a)) \equiv \forall x,y\ P(f(y)) \vee Q(g(x,y)) \vee y \neq a.$$

Example 5.2.2 *To illustrate the whole unskolemization process, consider the clauses*

C_1: $P(a, f(a)) \vee Q(g(a,y))$

C_2: $R(f(b)) \vee Q(g(b,w))$

with the skolem functions f and g. By variable abstraction, we obtain

C_1: $P(x, f(x)) \vee Q(g(x,y)) \vee x \neq a$

C_2: $R(f(v)) \vee Q(g(v,w)) \vee v \neq b.$

and by renaming v to x and w to y, we have

C_1: $P(x, f(x)) \vee Q(g(x,y)) \vee x \neq a$

C_2: $R(f(x)) \vee Q(g(x,y)) \vee x \neq b.$

The conditions of consistency and linearity are met, so we can rewrite the formula corresponding to the clause set in two steps using 5.1, by introducing the first-order individual variables z_g for g and z_f for f.

$$\begin{array}{rcl} & \exists f\ (\exists g \forall x,y\ P(x,f(x)) \vee Q(g(x,y)) \vee x \neq a & \wedge \\ & R(f(x)) \vee Q(g(x,y)) \vee x \neq b) & \\ \equiv & \exists f\ \forall x\ (\forall y \exists z_g\ P(x,f(x)) \vee Q(z_g) \vee x \neq a & \wedge \\ & R(f(x)) \vee Q(z_g) \vee x \neq b) & \\ \equiv & \forall x \exists z_f\ (\forall y \exists z_g\ P(x,z_f) \vee Q(z_g) \vee x \neq a & \wedge \\ & R(z_f) \vee Q(z_g) \vee x \neq b) & \end{array}$$

Note that we have to start the quantifier prefix conversion with the function having the most arguments if we want to apply 5.1 successively.

By $SkExpr(M), SkExpr(C), SkExpr(t)$, we denote the set of skolem expressions occurring in a clause set M, clause C or term t. $\|t\|$ for a term $t = f(\bar{s})$ denotes, a little sloppy, the arity of the top symbol f of t. We now give exact algorithms for performing the steps sketched above.

5.2.1 Normalizing argument lists by variable abstraction

Proposition 5.2.3 *Let C be a clause, $z \notin Vars(C)$ a fresh variable not occurring in C. Let $\Pi = \{\pi_1, \ldots, \pi_n\}$ be a set of occurrences of the term t in C, that is, $C|_{\pi_i} = t$. Then*

$$\forall \bar{x}\ C \equiv \forall \bar{x}, z\ C[\Pi/z] \vee z \neq t$$

where $\bar{x}$ contains the free variables in C.

For example, we have

$$P(a, a) \equiv \forall x\ P(x, a) \vee x \neq a.$$

Using this equivalence repeatedly on all arguments of skolem functions, we can assure that all skolem expressions only have variable arguments. Note that the above proposition does not force us to replace a term t everywhere in the clause when we abstract it, as shown in the example. This will be exploited in the next section.

We call expressions with only variable arguments *flat*. The algorithm **Abstract** flattens all skolem functions in a clause. For brevity, by $f[t]$ we denote a function f with a non-variable term t as one of its arguments, and by $C[f]$ a clause C containing the function symbol f. $C[t \mapsto z_t]$ denotes the clause that results from replacing t in C by z_t.

Algorithm 5.2.4

> **set**<**clause**> **Abstract**(**set**<**clause**> M)
> **While** (There is a clause $C[f] \in M$ containing a
>   non-linear skolem expression $f[t]$){
>   **Let** $C' = C[t \mapsto z_t] \vee z_t \neq t$; // z_t does not occur in C
>   **Let** $M = (M \cup C') \setminus C$; }
> **Return**(M);

5.2.2 Establishing argument uniqueness

Given a clause set M with only flat skolem functions, we can use proposition 5.2.3 again to ensure that each variable argument of each skolem function occurs only once in the argument list. Then we say that M satisfies the condition of uniqueness. For example, we replace

$$P(x, f(x, x)) \vee Q(g(x, y, y))$$

by

$$P(x, f(x,z)) \vee Q(g(x,v,y)) \vee z \neq x \vee v \neq y.$$

For the algorithm UNSKOLEMIZE, we choose the abstractions establishing argument uniqueness in such a way that the consistency and linearity of a clause is preserved, i.e., skolem expressions headed by the same skolem function symbol are already equal, and the argument sets of all skolem expressions are linearly ordered by set inclusion.

Unfortunately, sometimes it is impossible to preserve these properties when uniqueness of arguments is required. For example, consider the unit clause

$$P(f(x,x), g(x,y)),$$

where f, g are skolem functions. It is obvious that there cannot be a unique abstraction that established argument uniqueness and preserves local linearity. However, if the argument lists are ordered by multiset inclusion, then the following (straightforward) algorithm **Unique** preserves this order. It assumes that all skolem expressions in the input clause set are flat.

Algorithm 5.2.5

set<clause> Unique(**set<clause>** M)
// While there is a skolem expression with duplicate arguments in M
While ($\exists C \in M, \pi_f \in Occ(C)$ *such that* $\mathrm{C}|_{\pi_\mathrm{f}} = \mathrm{f}(\bar{\mathrm{x}}) \in \mathrm{SkExpr(C)}$
and $|Occ(f(\bar{x}), w)| \ \geq 2$ *for some variable* w){
// Replace one (any) occurrence of w in every skolem expression
// in C that contains w more than once
// Collect these expressions in Dup
Let $Dup = \{\pi_i \mid \ |Occ(C|_{\pi_i}, w)| \geq 2\}$
For ($i = 1$ **To** $|Dup|$)
Let $s = C|_{\pi_i}$;
Choose $\rho_i \in Occ(s, w)$;
// Replace all occurrences with a fresh variable $u \notin Vars(c)$
Let $C' = C[\rho_i, \ldots, \rho_{|Dup|}/u]$;
Let $C' = C' \vee u \neq w$;
Let $M = M \setminus C \cup \{C'\};$}
Return(M);

It is easy to see that **Unique** preserves equivalence of a clause set:

Lemma 5.2.1 *For a clause set M with skolem expressions occurring only flat,*

$$\exists \bar{f} \forall \bar{x} M \equiv \exists \bar{f} \forall \bar{x} \ \mathbf{Unique}(M)$$

Proof. In the algorithm, we have $C \equiv C'$ by 5.2.3, so in M clauses are only substituted by equivalent ones, which preserves equivalence. □

The algorithm **Unique** makes arguments unique preserves strong linearity in a clause, which we will define by

Definition 5.2.6 (Strong local linearity) *Let C be a clause. Let $f(\bar{s})$ and $g(\bar{t})$ be two skolem expressions occurring C. If for any such two expressions, $\|f\| \leq \|g\|$, then*

$$|Occ(f(\bar{s}), s_i)| \ \leq \ |Occ(g(\bar{t}), s_i)|.$$

That is, the argument sets of f and g are linearly ordered by set inclusion, and each argument of f occurs in g at least as often as in f. In other words, they are linearly ordered by multiset inclusion.

We say that a clause set M is strongly locally linear if all its clauses are.

Proposition 5.2.7 *Let M be a clause set with all skolem expressions flat. If M is strongly locally linear, then* **Unique**(M) *is.*

Proof. The proof needs no special ideas, but requires some formal care. We have to show that **Unique**(M) is strongly locally linear. We examine the step that constructs C' from C and show that C' as in the definition of **Unique** is strongly locally linear if C is.

To begin with, we define $red(s, w, u)$ as the result of replacing one occurrence of the variable w in the flat skolem expression f by the variable u, if w occurs more that once in the argument list of s, and s otherwise. We will use red to analyze the abstraction step in the **Unique** algorithm.It is easy to see that

$$(1)\ Args(s) \subseteq Args(red(s, w, u))$$

for any s, w, u with $u \notin Args(s)$. Furthermore, for two flat skolem expressions s_f, s_g with $Args(s_f) \subseteq Args(s_g)$, and any variable $u \notin Args(s_f) \cup Args(s_g)$ we have

$$(2)\ \forall v\ |Occ(s_f, v)| \ \leq \ |Occ(s_g, v)| \text{ implies } \forall w\ red(s_f, w, u) \subseteq red(s_g, w, u),$$

which can easily be shown using (1) and examining the cases $w \in Args(f)$ and $w \notin Args(f)$.

Now we look at the abstraction step in **Unique**. Let w and u denote the same variables as in the algorithm. By assumption, C is strongly locally linear. Consider two occurrences $\pi_f, \pi_g \in Occ(C')$ of skolem expressions with $C'|_{\pi_f} = s'_f, C'|_{\pi_g} = s'_g$. By the definition of C', it is easy to see that $\pi_f, \pi_g \in Occ(C)$. Let $s_f \stackrel{\text{def}}{=} C|_{\pi_f}, s_g \stackrel{\text{def}}{=} C|_{\pi_g}$. Again by the definition of C' and red, we have

$$(3)\ s'_f = red(s_f, w, u) \text{ and } s'_g = red(s_g, w, u).$$

Then, we also have $\|s_f\| = \|s'_f\|$ and $\|s_g\| = \|s'_g\|$ because the functions symbols are unchanged by red, which implies $\|s_f\| \leq \|s_g\|$, and therefore, exploiting that C is strongly locally linear,

$$s_f \subseteq s_g \text{ and } |Occ(s_f, v)| \leq |Occ(s_g, v)| \text{ for any variable } v,$$

so s_f and s_g satisfy the preconditions of (2). With (3), we can substitute $red(s_f, w, u)$ by s'_f and $red(s_g, w, u)$ by s'_g in (3), resulting in

$$s'_f \subseteq s'_g \text{ and } |Occ(s'_f, v)| \leq |Occ(s'_g, v)| \text{ for any variable } v.$$

which was to be proven. □

5.2.3 Establishing consistency

The next algorithm **Consistency** establishes consistency of skolem expressions in a clause set M with only flat skolem expressions by computing an appropriate substitution σ. We call σ a *globally consistent substitution*, or short, *consistent substitution*. If a consistent substitution for M exists, **Consistency** finds one. This substitution has several important properties:

- If there is some μ such that $\mu(M)$ is globally consistent, then $\sigma(M)$ is globally consistent.
- If there is some consistent μ that is also a renaming of M, then **Consistency** finds some such σ. We call μ a *consistent renaming*.
- If M satisfies uniqueness of arguments, and there is a consistent renaming μ of M such that $\mu(M)$ satisfies uniqueness of arguments, then $\sigma(M)$ does.

We need

Definition 5.2.8 (Local and global consistency) *A skolem function f is consistent in a clause C if all skolem expressions headed by f in C are* *equal. It is consistent in a clause set M if all skolem expressions headed by f in M are equal.*

A clause C is consistent, *if all skolem functions are consistent in it. A clause set M is* locally consistent *if all its clauses are consistent. A clause set M is* globally consistent *if all skolem functions are consistent in it.*

If a clause is not consistent, then there is no renaming that can make it consistent, which can easily be seen, so the algorithm assumes a locally consistent input clause set M. The idea behind the algorithm is to collect variables in sets that must necessarily be mapped on the same variable by any consistent renaming, and then construct a substitution from these sets. Extending our previous definition, we define $SkExpr(M, f)$ to be the set of all skolem expressions in a clause set M headed by the skolem function symbol f. We will need to "flatten" a set of sets:

$$Flat(\{M_1, \ldots, M_n\}) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} M_i.$$

Now for the algorithm. Its input consists of a clause set M and an empty substitution σ by reference, and it alters σ such that $\sigma(M)$ is consistent. It returns $\sigma(M)$ and the substitution by reference. See figure 5.1 for the algorithm. We give an example.

Example 5.2.10 *For* **Consistency***, only the skolem functions in a clause matter, which is why we consider only sets of skolem functions representing clauses in the following. Consider the skolem function sets*

$$\begin{array}{ll} C_1: & \{f(x_1), g(x_1, x_2)\} \\ C_2: & \{g(y_2, y_3), h(y_1, y_2, y_3)\} \\ C_3: & \{f(z_1), h(z_1, x_2, z_3)\} \end{array}$$

Consistency *computes the set $\mathcal{E}^0$*

$$\left\{ \begin{array}{l} \{\{x_1, z_1\}_{f_1}\}, \{\{x_1, y_2\}_{g_1}\}, \{\{x_2, y_3\}_{g_2}\} \\ \{\{y_1, z_1\}_{h_1}\}, \{\{y_2, z_2\}_{h_2}\}, \{\{y_3, z_3\}_{h_3}\} \end{array} \right\}$$

where sets are indexed with the function and the argument position of the function whose consistency condition they represent. Now **Consistency**

Algorithm 5.2.9

set<**clause**> **Consistency**(**set**<**clause**> M, **substitution** $\&\sigma$)
Let $\mathcal{E}^0 = \emptyset$;
Foreach (Skolem function f)
  // Collect the i-th arguments of each expression
  For $i = 1$ **To** $\|f\|$ **Do** {
    Let $A = \{x \mid \exists s \in SkExpr(M, f) \textit{ with } x = s|_i\}$;
    Let $\mathcal{E}^0 = \mathcal{E}^0 \cup \{\{A\}\};\}$
// Now join sets in $\mathcal{E}$ as long as their flattenings are not all disjoint
Let $j = 0$;
While $(\exists E_1, E_2 \in \mathcal{E}^j \textit{ with } Flat(E_1) \cap Flat(E_2) \neq \emptyset)\{$
    Let $\mathcal{E}^{j+1} = \mathcal{E}^j \setminus \{E_1, E_2\} \cup \{E_1 \cup E_2\}$;
    Let $j = j + 1;\}$
// Now build substitution σ.
// Assume $u_i \notin M$ to be a fresh variable
// for $1 \leq i \leq |\mathcal{E}^j|$, and let
// $\mathcal{E}^j = \{\mathcal{E}^j_1, \ldots, \mathcal{E}^j_{|\mathcal{E}^j|}\}$
Let $\sigma = \emptyset$;
For $i = 1$ **To** $|\mathcal{E}^j|$
    Let $\sigma = \sigma \cup \bigcup_{v \in Flat(\mathcal{E}^j_i)} \{v \mapsto u_i\}$;
Let $M' = \sigma M$;
Return(M');

Figure 5.1: The algorithm **Consistency**

successively puts sets with common variables in one set:

$$\left\{\begin{array}{l}\{\{x_1,z_1\}_{f_1}\},\{x_1,y_2\}_{g_1}\},\{\{x_2,y_3\}_{g_2}\}\\ \{\{y_1,z_1\}_{h_1}\},\{\{y_2,z_2\}_{h_2}\},\{\{y_3,z_3\}_{h_3}\}\end{array}\right\}\ \textit{since}\ g_1\cap f_1=\{x_1\}$$
$$\left\{\begin{array}{l}\{\{x_1,z_1\}_{f_1},\{x_1,y_2\}_{g_1},\{y_2,z_2\}_{h_2}\}\}\}\\ \{\{x_2,y_3\}_{g_2},\{y_1,z_1\}_{h_1}\},\{\{y_3,z_3\}_{h_3}\}\end{array}\right\}\ \textit{since}\ h_2\cap g_1=\{y_2\}$$
$$\left\{\begin{array}{l}\{\{x_1,z_1\}_{f_1},\{x_1,y_2\}_{g_1},\{y_2,z_2\}_{h_2}\}\}\\ \{\{x_2,y_3\}_{g_2},\{y_3,z_3\}_{h_3}\},\{\{y_1,z_1\}_{h_1}\}\end{array}\right\}\ \textit{since}\ h_3\cap g_2=\{y_3\}.$$

Now all flattenings of "variable set"-sets are disjoint. The variables in these sets are now assigned new variables, yielding the substitution (abbreviated)

$$\sigma\{\{x_1,z_1,y_2,z_2\}\mapsto u_1,\{y_3,z_3,x_2\}\mapsto u_2,\{y_1,z_1\}\mapsto u_3\}.$$

Applying σ to the function set, we get

$$\begin{array}{ll}\sigma(C_1): & \{f(u_1),g(u_1,u_2)\}\\ \sigma(C_2): & \{g(u_1,u_2),h(u_3,u_1,u_2)\}\\ \sigma(C_1): & \{f(u_1),h(u_3,u_1,u_2)\}.\end{array}$$

σ obviously is a consistent renaming.

Now for the properties of **Consistency**. It is clear that it always terminates, as $\mathcal{E}^0$ is finite and $|\mathcal{E}^{j+1}|\le|\mathcal{E}^j|$. We show that the algorithm is correct and complete, starting with correctness.

Proposition 5.2.11
For any locally consistent clause set M, **Consistency***(M) is a globally consistent clause set.*

Proof. We have to check that σ is a substitution, that is, no variable is "mapped" on two different variables. This is so because $Flat(E_1)\cap Flat(E_2)=\emptyset$ for all $E_1,E_2\in\mathcal{E}^j$ at the end of the algorithm.

σ also makes M globally consistent. If, for a skolem function f, $|SkExpr(M,f)|=1$, then it already is consistent, and *any* substitution preserves its consistency.

If $|SkExpr(M,f)|\ge 2$, then consider any two argument variables v_1,v_2 in the same argument position of two different skolem expressions $f(\bar{x_1}),f(\bar{x_2})$, both headed by f. Then there is some $E\in\mathcal{E}^0$ such that $v_1,v_2\in Flat(E)$, by definition of $\mathcal{E}^0$. But then, by definition of the $\mathcal{E}^j$, there is some $E_j\in\mathcal{E}^j$ such that $E\in E^j$, so both variables are assigned the same value at the end of the algorithm. This is easily proven by induction over j. □

We specified several completeness conditions for **Consistency** at the beginning of the section and now prove them. They all follow from a minimality condition that the algorithm satisfies and which says that **Consistency** identifies variables only if they have to be identified by any consistent substitution.

Lemma 5.2.2 *Let σ be the substitution computed by* **Consistency**. *Let $\sigma(v) = \sigma(w)$ for some two variables in M. If there is a consistent substitution μ for M, then $\mu(v) = \mu(w)$.*

Proof. We show by induction over k that every consistent substitution μ must map all variables of the set $Flat(E^k)$, $E^k \in \mathcal{E}^k$ on the same variable. For $k = 0$, this is clear by definition of $\mathcal{E}^0$ and the definition of consistency.

Now assume this is true for some $k \leq n$, where n is the value of the variable j at the end of the algorithm. As in the algorithm, let $E_1, E_2 \in \mathcal{E}^k$, $Flat(E_1) \cap Flat(E_2) \neq \emptyset$ be given, which exist because otherwise the algorithm would have terminated before. We have to prove the induction hypothesis for all $E \in \mathcal{E}^{k+1}$. Consider some $E \in \mathcal{E}^{k+1}$. If $E \in \mathcal{E}^k$, then our claim is true because of the induction hypothesis. If $E \notin \mathcal{E}^k$, then $E = E_1 \cup E_2$, so there is some variable $v \in Flat(E_1) \cap Flat(E_2)$. Now, by the induction hypothesis, for all variables in $Flat(E_i)$, $i \in \{1, 2\}$, are mapped on the same variable, that is, for all $w \in Flat(E_i)$, $i \in \{1, 2\}$, $\mu(v) = \mu(w)$, so we have $\mu(v) = \mu(w)$ also for all $w \in Flat(E_1) \cup Flat(E_2)$. This completes the proof. □

Now the completeness conditions follow easily.

Proposition 5.2.12 *Let σ be the substitution computed by* **Consistency** *for a clause set M. If there is some consistent renaming μ of M, then σ is a consistent renaming. If there is a consistent renaming μ of M that preserves uniqueness of arguments, then σ does also.*

Proof. We prove by contradiction. Assume that σ is not a consistent renaming, though there exists one. Then for some clause C and for some variables $x, y \in Vars(C), x \neq y, \sigma(x) = \sigma(y)$. By 5.2.2, then $\mu(x) = \mu(y)$, which contradicts the assumption that μ is a renaming. Preservation of argument uniqueness is proven analogously. □

5.2.4 Establishing linearity

Finally, we must find a renaming a consistent clause set M which puts all argument sets of skolem expressions in M into a linear order. We then say that M is *globally linear* and define two related useful notions:

Definition 5.2.13 (Local and global linearity) *We say that a clause is* linear*, if the argument list of its skolem expressions are linearly ordered by set inclusion, that is, for some skolem expressions s_1, s_2 in C,*

$$Args(s_1) \subseteq Args(s_2) \text{ or } Args(s_2) \subseteq Args(s_1).$$

We say that a clause set M is locally linear if all its clauses are linear. We say that M is globally linear if the argument list of all its skolem expressions are linearly ordered by set inclusion.

It is easy to see that a non-linear clause cannot be made linear by any renaming, so we require local linearity (LL) of M for the next algorithm **Linearity**, which we abbreviate by (LL). Also, we require M to satisfy uniqueness of argument lists (U), consistency (C) and minimal consistency (MC) as formulated in 5.2.2.

To start with, we give a criterion that is necessary and sufficient for the existence of a globally linear renaming of a consistent clause set, that is, a renaming ρ such that $\rho(M)$ is globally linear. For two sets M_1, M_2, we write $M_1 \subseteq\!\!\supseteq M_2$ if $M_1 \subseteq M_2$ or $M_2 \subseteq M_1$.

Proposition 5.2.14 (Extendible linearity) *If a clause set M fulfills (C), (U), (LL) and (MC), there exists a consistent globally linear renaming ρ of M if and only if for any skolem expressions s_1, s_2, s_3, we have*

$$Args(s_1), Args(s_2) \subseteq Args(s_3) \text{ implies } Args(s_1) \subseteq\!\!\supseteq Args(s_2),$$

that is, set inclusion orders $\{Args(s_1), Args(s_2), Args(s_3)\}$ linearly.

We first prove necessity of this criterion (EL). For sufficiency, we afterwards prove an intermediate lemma, and then give an algorithm that computes ρ as desired if (EL) is fulfilled.

Lemma 5.2.3 (Necessity of (EL)) *If there is some renaming ρ making M globally linear, then (EL) holds for M.*

Proof. We show by contraposition that (EL) must hold since otherwise a substitution making M globally linear could not be a renaming.

Now assume that $\rho(M)$ is globally linear, and that s_1, s_2, s_3 exist such that $Args(s_1), Args(s_2) \subseteq Args(s_3)$, but $Args(s_1) \mathrel{\subseteq\!\!\!\!\supseteq} Args(s_2)$ is violated, so that there exists some variable $v \in Args(s_1)$ with $v \notin Args(s_2)$. W.l.o.g., we assume that $Args(s_1) \subseteq Args(s_2)$.

We begin by showing that

$$(*)\ Args(\rho(s_1)) \subseteq Args(\rho(s_2)).$$

Since ρ is a renaming, it is a bijection on the argument lists of the s_i, which implies $|Args(s_i)| = |\rho(Args(s_i))|$ for $i \in \{1,2,3\}$. Now if $|Args(s_1)| = |Args(s_2)|$, then

$$(**)\ |Args(\rho(s_1))| = |Args(\rho(s_2))|.$$

$\rho(M)$ is globally linear, therefore $Args(\rho(s_1)) \mathrel{\subseteq\!\!\!\!\supseteq} Args(\rho(s_2))$. With $(**)$, this even implies $Args(\rho(s_1)) = Args(\rho(s_2))$.

If $|Args(s_1)| < |Args(s_2)|$, then

$$(**)\ |Args(\rho(s_1))| < |Args(\rho(s_2))|.$$

But then $Args(\rho(s_2)) \subseteq Args(\rho(s_1))$ is of course impossible, which with $Args(\rho(s_1)) \mathrel{\subseteq\!\!\!\!\supseteq} Args(\rho(s_2))$ implies $(*)$.

Now, since $(*)$ holds, with v as above, there is some $x \in Args(s_2), x \neq v$ such that $\rho(v) = \rho(x)$. But $Args(s_1), Args(\ s_2) \subseteq Args(s_3)$, so $x, v \in Args(s_3)$, that is ρ identifies two variables in the same expression, hence it cannot be a renaming as was assumed. $\square$

The next lemma allows us to abstract from the clause structure and to operate on variable sets only.

Lemma 5.2.4 *If (EL) and (MC) hold for M, then for any two skolem expressions s_1, s_2 with $Args(s_1) \cap Args(s_2) \neq \emptyset$ there exists a skolem expression s_3 in M such that $Args(s_1) \cap Args(s_2) = Args(s_3)$.*

Proof. (Sketch)

To prove this, we need the notion of *dominance*, which is defined relatively to a clause set M which we take as constant in the following. A skolem function f directly dominates another function g if they both occur in the same clause, and $Args(g) \subseteq Args(f)$. Note that dominance is reflexive. Domination is the transitive closure of direct domination. We write $f \succeq g$ if f dominates g. This notion is well-defined because of local linearity and consistency. For two skolem expressions $s_1 = f(\bar{x})$, $s_2 = g(\bar{y})$, it is easy to see that with (MC) and (LL),

$$f \succeq g \text{ implies } Args(s_2) \supseteq Args(s_1).$$

Now for the proof. We distiguish three cases. If f and g are both dominated by some h, then by (EL) and the above, we have $Args(f) \subseteq Args(g)$, and the claim holds.

Now assume there is no such common dominating function. Then consider the sets D_f and D_g which contain the skolem expressions in M headed by functions dominated by f and g. By (EL), both sets are already linearly ordered by set inclusion on their arguments lists. Now $D_f \cap D_g$ is empty or not. If it is empty, define $S = \emptyset$. If it is not empty, take C to be $D_f \cap D_g$, and $s_3 = h(\bar{z})$ to be the maximum function h with respect to set inclusion on argument sets, which is again well-defined by (EL) and consistency. We show that s_3 is the searched function, if it exists, or that otherwise $Args(s_1) \cap Args(s_2) = \emptyset$. In other words, we show that $Args(s_1) \cap Args(s_2) = S$. $Args(s_1) \cap Args(s_2) \supseteq S$ is easy. For $S = \emptyset$, this is trival, and if $S \neq \emptyset$, then this follows because both f and g dominate h. For the other direction $Args(s_1) \cap Args(s_2) \supseteq S$, we assume that there is some $v \in \bar{x} \cap \bar{y}$ such that $v \notin \bar{z}$. We then change v in all clauses which contain functions that are dominated by some function that also dominates f to a new variable u not in M. This does not destroy consistency, and v is not changed in g. Essentially, this is so because in all clauses with functions dominated by g and f, v may change, but the common dominated functions do not contain v, and functions that dominate f or g only occur in different clauses by assumption (this step requires a little elaboration).

But then we found a consistent renaming with less variable identifications than before, which contradicts (MC). □

Now we define the algorithm **Linearity**, which operates only on terms.

Example 5.2.16 *As in the example for the* **Consistency** *algorithm, we only consider the skolem function sets contained in clauses instead of the whole clauses. Let the following clause set be an output of* **Consistency***:*

$$\begin{aligned}
&C_1 : \{f(v_1), g(v_1, v_2)\}\\
&C_2 : \{f(v_1), h(v_1, y_1, y_2)\}\\
&C_3 : \{f(v_1), g(v_1, v_2), k(v_1, v_2, z)\}
\end{aligned}$$

Linearity *represents this by*

$$\mathcal{A} = \{\{v_1\}, \{v_1, v_2\}, \{v_1, y_1, y_2\}, \{v_1, v_2, z\}\}.$$

Then the σ_j are computed:

$$\begin{aligned}
\sigma_1 &= \{v_1 \mapsto u_1\}\\
\sigma_2 &= \sigma_1 \cup \{v_2 \mapsto u_2\}\\
\sigma_3 &= \sigma_2 \cup \{y_1 \mapsto u_3, y_2 \mapsto u_2\}\\
\sigma_4 &= \sigma_3 \cup \{z \mapsto u_3\}.
\end{aligned}$$

Algorithm 5.2.15

```
set<clause> Linearity( set<clause> M)
// Collect all argument sets of skolem expressions in a set 𝒜
Let S = SkExpr(M);
Let 𝒜 = ⋃_{s∈S}{Args(s)};
Let σ_0 = ∅;
// Build up substitution σ
// Assume that S = {V_1, ..., V_n} with |V_j| ≤ |V_{j+1}|
// for 1 ≤ j ≤ n − 1
For j = 1 To n Do {
  // Let {x_1, ..., x_l} denote the variables in V_j
  // with σ(x_i) = x_i for 1 ≤ i ≤ l }
  // and let the u_i be fresh variables not in 𝒜
  Let σ_j = σ_{j−1}
  For i = 1 To l
    Let σ_j = σ_j ∪ {x_i ↦ u_{|V|−i+1}}; }
Return(σ_n(M));
```

Figure 5.2: The algorithm **Linearity**

σ is a renaming, and, applied to the skolem functions, we have linearity:

$$\begin{array}{l}\sigma(C_1) : \{f(u_1), g(u_1, u_2)\} \\ \sigma(C_2) : \{f(u_1), h(u_1, u_3, u_2)\} \\ \sigma(C_3) : \{f(u_1), g(u_1, u_2), k(u_1, u_2, u_3)\}\end{array}$$

We show that **Linearity** indeed computes a linear renaming if (EL) and (MC) hold for M, which is the case if M is the output of **Consistency**.

Lemma 5.2.5 *σ_n as computed by* **Linearity**(M) *is a linear renaming if (MC) and (EL) hold for M.*

Proof.
It is easily shown by induction on j that $\sigma(V_j) = \{u_1, \ldots, u_{|V_j|}\}$,which gives linearity of σ.

To show that σ is a renaming, first note that be the definition of $\mathcal{A}$ and by lemma 5.2.4, which holds because (MC) holds, $\mathcal{A}$ is closed under intersection. We show that for any two distinct variables x, y occurring in some $V_j \in \mathcal{A}$, we have $\sigma_i(x) \neq \sigma_i(y)$. Let V_{i_x}, V_{i_y} denote the sets that were processed when x and y were assigned one of the u_i. Now either $V_{i_x} = V_{i_y}$ or not.

If $V_{i_x} = V_{i_y}$, then x and y were assigned one of the u_i at the same time, and different variables by definition of the algorithm.

Now assume $V_{i_x} \neq V_{i_y}$.

If V_{i_x} is not a subset of V_j, then $I \overset{\text{def}}{=} V_j \cap V_x$ is a proper subset of V_j. As $\mathcal{A}$ is closed under intersection, $I \in A$. But $x \in V_j$ by assumption, $x \in V_{i_x}$ by definition, so that $x \in V_j \cap V_{i_x} = I$ with $|I| \leq |V_{i_x}|$ because I is a proper subset of V_j. But then x already occurred in a smaller set an must have been labeled before V_{i_x} was processed, which contradicts the choice of V_{i_x} .

Thus, it remains that $V_{i_x} \subset V_j$, and $V_y \subset V_j$ by the same arguments, hence by (EL), $V_x \subseteq\!\supseteq V_y$.

W.l.o.g., $V_{i_x} \subseteq V_{i_y}$, so that x was assigned u_k with $k \leq |V_{i_x}|$. Besides, all variables $v \in V_{i_x}$ were already assigned new variables when V_{i_y} was processed, and $y \notin V_{i_x}$ by definition of V_{i_y}, so y was assigned some u_l with $l > |V_{i_x}|$. $\square$

This proof completes the proof of proposition 5.2.14.

5.2.5 The basic Unskolemize algorithm

Now we integrate the algorithms of the previous sections in one algorithm Unskolemize. We prove it correct and make some observations about its completeness.

The algorithm UNSKOLEMIZE, shown in figure 5.3, takes a second-order formula $\Phi = \exists \bar{f} \forall \bar{x}\ M$ as input, where M is a conjunction of clauses, and tries to reduce it to an equivalent first-order one. If it succeeds, it returns a formula without second-order quantifiers, if it fails, a second-order formula equivalent to the input. In any case, UNSKOLEMIZE$(\Phi) \equiv \Phi$.

We assume that M, conceived as a set of clauses, cannot be partitioned into two clause sets M_1, M_2, such that the sets S_1, S_2 of skolem functions in M_1, M_2 are disjoint. If this were the case, we could unskolemize M_1 and M_2 clause sets separately, by the rules for quantifier distribution. We also

Algorithm 5.2.17

> **formula** UNSKOLEMIZE(**formula** $\exists \bar{f} \forall \bar{x} M$)
> **Let** $M = \mathbf{Separate}(M)$;
> **Let** $M_a = \mathbf{Abstract}(M)$;
> **Let** $found = \mathbf{false}$; // signals solution
> **Let** $\mathcal{A} = \{$all abstractions α that make M unique$\}$
> **While** $(\mathcal{A} \neq \emptyset$ *and* $\neg found)\{$
>   **Choose** $\alpha \in \mathcal{A}$;
>   **Let** $\mathcal{A} = \mathcal{A} \setminus \{\alpha\}$;
>   // Apply abstraction (informal notation)
>   **Let** $M_u = \alpha(M_a)$;
>   **Let** $\sigma = \emptyset$;
>   **Let** $M_c = \mathbf{Consistency}(M_u, \&\sigma)$;
>   **If** (σ is a global renaming and
> M_c satisfies local and extendable linearity conditions)
>   **then**$\{$ **Let** $M_l = \mathbf{Linearity}(M_c)$;
>   **Let** $\Phi = \mathbf{Reduce}(\exists \bar{f} \forall \bar{x} M_c)$;
>   **Let** $found = \mathbf{true}; \}$
> // Return input if unskolemization failed
> **If** $(\neg found)$ **Let** $\Phi = \exists \bar{f} \forall \bar{x} M$;
> **Return**(Φ);

Figure 5.3: The algorithm UNSKOLEMIZE

assume an algorithm **Reduce** which converts a consistent, globally linear clause set by equivalence 5.1 into an equivalent first-order formula. This algorithm consists of simply applying proposition 5.1 to the skolem symbols in order of decreasing arity, as we already demonstrated in the example

5.2.2. Finally, the algorithm **Separate** makes clauses in a clause set M pairwise variable disjoint.

Proposition 5.2.18 (Correctness of UNSKOLEMIZE) *For any formula $\exists \bar{f} \forall \bar{x}\ M$, where M is a disjunction of clauses,*

$$\text{UNSKOLEMIZE}(\exists \bar{f} \forall \bar{x}\ M) \equiv \exists \bar{f} \forall \bar{x}\ M.$$

Proof. This follows from the properties of the used algorithms and transformations. Note that the preconditions for **Linearity** are satisfied: M_u is unique, and any renaming must preserve this property, so (U) is satisfied. (EL) and (LL) are demanded by the algorithm, and (MC) is guaranteed by the properties of **Consistency**. □

Note that a successful unskolemization may depend on the way arguments are made unique by **Unique**. Consider the unit clauses

$$P(g(x,x), f(x,x))$$

$$Q(g(y,y), f(y,y), a).$$

Two unique equivalent versions are

$$P(g(x,u), f(x,u)) \vee x \neq u$$

$$Q(g(y,v), f(y,v), a) \vee v \neq y.$$

and

$$P(g(x,u), f(u,x)) \vee x \neq u$$

$$Q(g(y,v), f(y,v), a) \vee v \neq y.$$

The first version can obviously be renamed consistently, while the second one can not: any consistent substitution σ would have to satisfy $\sigma(x) = \sigma(y)$ and $\sigma(u) = \sigma(y)$, therefore $\sigma(x) = \sigma(u)$, so σ could not be a renaming. Hence, we may have to try all possibilities for making a clause set unique.

5.3 Extensions of UNSKOLEMIZE

We now list two cases for which UNSKOLEMIZE fails and propose ways of treating them in general.

5.3.1 Uniqueness revisited

We have already shown in 5.2.2 that sometimes uniqueness of arguments cannot be obtained without destroying local linearity. The example clause was

$$P(f(x,x),g(x,y)).$$

However, the following equivalence can help to make f unique without variable abstraction:

Proposition 5.3.1 *Let $\phi(y)$ be a $first-order$ formula with the free variable y, f a function not occurring in ϕ. Then*

$$\exists f\ \phi(y \mapsto f(x_1,\ldots,x_i,\ldots,x_j,\ldots,x_n)) \equiv \exists f\ \phi(y \mapsto f(x_1,\ldots,x_i,\ldots,x_n))$$

if x_i and x_j denote the same variable.

We could then replace the above clause by the equivalent (interpreted as second-order formula)

$$P(f(x),g(x,y)),$$

which is easily unskolemized by UNSKOLEMIZE. In general, we can extend UNSKOLEMIZE by a procedure that, for each skolem function, tries to make it consistent in the clause set, and then replaces it with a function without duplicate arguments.

5.3.2 Henkin quantifiers

Henkin quantifiers are a generalization of first-order quantifiers that may sometimes allow us to write second-order formulas we cannot unskolemize in a suggestive way. Consider the following clause:

$$P(x,y,f(x),g(y))$$

with the skolem functions f and g. As opposed to clauses obtained by skolemization, the argument sets of the skolem functions are not linearly ordered by set inclusion, and as they are in one clause, UNSKOLEMIZE fails. Leon Henkin ([Hen61]) suggested to write the corresponding second-order formula $\exists f,g\ \forall x,y\ P(x,y,f(x),g(y))$ as

$$\begin{pmatrix} \forall x \exists z_f \\ \forall y \exists z_g \end{pmatrix} P(x,y,z_f,z_g)$$

to indicate that z_f only depends on x, and z_g only on y. We generalize this notation for our purposes to

$$\begin{pmatrix} \forall \bar{x}_1 \exists y_1 \\ \vdots \\ \forall \bar{x}_n \exists y_n \end{pmatrix} \phi(\bar{x}_1, \ldots, \bar{x}_n, y_1, \ldots, y_n)$$

as an abbreviation for

$$\exists f_1, \ldots, f_n \; \forall \bar{x}_1, \ldots, \bar{x}_n \; \phi(y_1 \mapsto f_1(\bar{x}_1), \ldots, y_n \mapsto f_n(\bar{x}_n)),$$

where the f_i do not occur in ϕ. The following proposition states that all clause sets can be unskolemized using Henkin quantifiers.

Proposition 5.3.2 *Let M be a clause set with skolem functions, interpreted as the second order formula $\exists \bar{f} \forall \bar{x} \; M$. This formula can be equivalently be written with a Henkin quantifier in the above form.*

Proof. We only have to prove that we can transform M into the form of ϕ above, that is, all skolem functions are renamed consistently. By the proposition 5.2.3 used for variable abstraction, we can rewrite all skolem functions in such a way that any two occurrences of skolem functions have disjoint argument sets, even within clauses, and all variables are unique in each argument list. It is obvious to see that M has a consistent renaming in this form, which was to be proven. $\square$

5.4 The McCune algorithm

In [McC88], McCune describes an algorithm for eliminating skolem functions. It is only proven and designed to be satisfiability preserving, but can also be shown to be equivalence preserving. Though independently developed, it works in much the same way as UNSKOLEMIZE: First tries to establish global consistency, then global linearity. Variable abstraction is employed to achieve consistency in practically the same way as in UNSKOLEMIZE. Indeed, the consistent set M_c in the algorithm UNSKOLEMIZE is the same set that is the input to the linearization phase of the McCune algorithm.

However, the linearization phase of McCune's algorithm is much different. It has exponential time complexity and also fails to unskolemize some clause sets for which UNSKOLEMIZE succeeds. Given a consistent clause set, the algorithm attempts, for each permutation of all argument sets of skolem expressions, to unify all skolem expressions that are headed by functions of the same arity. Afterwards, it checks whether the computed unifier is a renaming. Of course, the number of permutations of each argument set grows exponentially with its size, and the number of combinations of the permutations of all arguments sets grows exponentially with the number of functions. Furthermore, this technique fails for simple clause sets. For example, consider the consistent clause set

$$P(f(x), g(x, y))$$

$$P(f(x), h(x, u, v))$$

All skolem expressions headed by functions with the same arity are already unified, but the clause set is not globally linear, hence it cannot be unskolemized. UNSKOLEMIZE finds the obvious solution by unifying y and u.

5.5 Some remarks on the scope and complexity of UNSKOLEMIZE

This section gathers some observations about the classes of formulas that UNSKOLEMIZE can unskolemize and its time complexity.

5.5.1 Scope

We should at least expect UNSKOLEMIZE to unskolemize all clause sets for which this is possible by renaming. However, we have already seen that this is not possible because **Unique** may destroy the local linearity of a clause, which makes it impossible to find a globally linear renaming. We conjecture that the algorithms **Consistency** and **Linearity** also work for a non-unique clause set, because the uniqueness condition has not been used anywhere. Due to limited time, we could not check this, so we only try to characterize the set of clauses for which UNSKOLEMIZE works in the current form, without the extension for establishing uniqueness as described in 5.3.

To begin with, we can simply give a criterion for clause sets which is sufficient for *failure* of UNSKOLEMIZE.

Proposition 5.5.1 *If a clause C is not strongly locally linear, then it cannot be made unique by variable abstraction without destroying local linearity.*

Proof. (Sketch)
Consider to skolem expressions s_1, s_2 with $Args(s_1) \subseteq Args(s_2)$. Let us assume x occurs more than once in s_1. If we apply variable abstraction to one of its occurrences, we introduce a fresh variable which must also be introduced in s_2, because otherwise we loose linearity, and of course it could not be reestablished by later abstractions. Hence it is clear that s_1 can only be made unique if we can introduce all new variables both in s_1 and s_2, which is only possible if each variable occurs in s_1 at most as often as in s_2. □

On the other hand, we have shown that **Unique** preserves strong local linearity, so that in this case, it does not destroy local linearity. We claim that a clause set M satisfying strong local linearity can be unskolemized using renaming only if **Unique**(M) can be unskolemized (note that the notion of strong local linearity has also been defined for clauses with non-variable skolem function arguments). Then we are able to prove

Proposition 5.5.2 *If M is a strongly locally linear, locally consistent clause set that can be made globally consistent and linear by renaming, then* UNSKOLEMIZE *finds such a renaming.*

Proof. We search for a substitution σ_{cl} which is a renaming on M such that $\sigma_{cl}(M)$ is globally linear and consistent. Note that, after all clauses have been made pairwise variable disjoint by **Separate** , such a global renaming exists if and only if a set of local renamings (for each clause) exists that also makes M locally and globally linear.

Now assume that M is strongly locally linear, and some σ_{cl} as above exists. Intuitively, is quite clear that such a substitution also exists for $M_a \stackrel{\text{def}}{=} \textbf{Abstract}(M)$, because the replacement of a term by a variable should only increases our chances to find it, as long as the term is replaced by the same variable in the *whole* clause (as opposed to the abstractions of occurrences **Unique** makes). Also, **Abstract** preserves strong local linearity. We do not prove these assertions.

Now we have that σ_{cl} exists for M_a, and that it is strongly locally linear. Our above claim was that some σ_{cl} exists also for $M_u \stackrel{\text{def}}{=} \textbf{Unique}(M_a)$, and we now have to show that it it is found by **Linearity** on $M_c \stackrel{\text{def}}{=} \textbf{Consistency}(M_u, \&\mu)$.

One way to see this is to observe that a consistent renaming unifies all skolem expressions headed by the same function symbol. When we regard the search for a consistent renaming as a unification problem, there is of course (under these simple conditions) a most general unifier, so any consistent renaming σ can be written as

$$\sigma = \sigma_c \circ \sigma_r,$$

with σ_c being the most general consistent renaming, and σ_r some substitution. Therefore, we can decompose σ_{cl}: $\sigma_{cl} = \sigma_c \circ \sigma_l$.

Using 5.2.2, it can also be shown that **Consistency** computes the most general renaming σ_c. It was to show that UNSKOLEMIZE finds σ_{cl} for M_u, provided it exists. If it exists, σ_c does exist, and by the completeness property stated in proposition 5.2.12, **Consistency** finds it. But then, by the completeness property 5.2.5, **Linearity** finds σ_l if and only if it exists for $M_c \stackrel{\text{def}}{=}$ **Consistency**(M_a), which completes the proof. □

5.5.2 Time complexity

Most algorithms used by UNSKOLEMIZE can be shown to have quadratic or even linear time complexity on the length of the input with any reasonable data structures and algorithms for sets and set operations. The only problem is the non-determinism we introduce when we establish uniqueness. This algorithm can indeed have exponential complexity. However, in the above section 5.5.1, we made the reasonable claim that we could get rid of this step, which would make UNSKOLEMIZE an algorithm with quadratic time complexity.

5.6 Future work

UNSKOLEMIZE can be seen as an inference strategy on a set of equivalence preserving rewrite rules. Therefore, we might study elimination of functions in the framework of rewrite systems: We could study inference strategies and there completeness, introduce new inference rules and examine how they combine with others etc. We could study rewrite rules used in optimized *skolemization* to see whether they are not only satisfiability, but also equivalence preserving. When the number of skolem functions and their arity is small, then it might even be possible to run through all combinations of rewritings of a clause set.

Alternatively, we could try to transfer the SCAN method of generating sufficiently many resolvents to the problem of function elimination. I propose to design a predicate elimination algorithm that works without skolem functions. An indication that such an algorithm is not too difficult to find might be the close resemblance between C-resolution and ordinary resolution. For first-order logic, calculi working without skolem functions have been designed, and perhaps they could be "lifted" for our second-order purpose. Then we could encode skolem functions as predicates and eliminate the introduced predicates, an approach not working with SCAN in its current form. This way, we could study function and predicate elimination in a uniform way. We could take advantage of methods from model theory, the theory of deductive systems and especially unification theory. Furthermore, as we have shown in chapter 4 that first-order theory proving and second-order quantifier elimination are related in a simple way, we could then study deduction problems in first-order logic from the viewpoint of second-order quantifier elimination, which might also be elucidating, and *thrilling*.

Chapter 6

Implementation of SCAN and UNSKOLEMIZE

This chapter contains a description of the program **Scan**, an implementation of the algorithms SCAN and UNSKOLEMIZE. The implementation is based on the code of the theorem prover OTTER, a wide-spread theorem prover for first-order logic, which has been developed by William McCune ([McC90]). From OTTER, **Scan** inherits the programming language C, basic data structures and a user interface with a comfortable, PROLOG-style parser for first-order formulas. The necessary extensions of the OTTER code also use C++ constructs, making **Scan** a C++ program on the whole.

The three main sections of this chapter deal with a description of the user interface, a general description of the algorithms used in the implementation and a detailed description of the implementation itself.

The algorithms together with the technical details of the implementation should provide a good specification of **Scan** and a guide through the code. To achieve this , we also provide some information about the OTTER data structures and their implementation, which has no official documentation.

The description of the user interface should give all of those who are eager to work with **Scan** on elimination problems a quick start.

But first we pay tribute to

6.1 The roots: OTTER

OTTER is short for Organized Techniques for Theorem-proving and Effective Research. The C program OTTER is basically an implementation of the resolution calculus and some of its extensions.

Apart from the standard resolution rule, OTTER offers more complicated inference rules like hyper-resolution and UR-resolution, paramodulation for equality reasoning and demodulation for rewriting techniques. Simplification rules like forward and back subsumption can help to remove redundancies. The inference rules are applied according to a simple set-of-support strategy, with weighing mechanisms for selecting a given clause.

To achieve a high performance, which has been an explicit design goal in the OTTER development ([McC90]), OTTER manages its clause set data via indexes. They are mainly used for quick retrieval of unification partners that are needed in the various occurring unification problems.

Though OTTER mainly operates on clauses, it can read both formulas and clauses via a comfortable parsing mechanism. This can parse formulas with user defined operators according to their precedence and notation (infix,prefix,postfix) which can be declared by the user. Thereby, a lot of redundant parentheses can be saved, resulting in good readability and convenient notation.

6.2 Scan features and how to use them

Scan contains implementations of all simplification rules described in 4. Their use is optional. **Scan** has two inference strategies, one eliminating predicates in arbitrary order, leaving the choice to SCAN, and one eliminating predicates successively. Results of the SCAN algorithm can be minimized to eliminate clauses that are implied by others.

The user interface of SCAN allows for declarations of notation type (prefix, infix, postfix) and precedence of operators to simplify input. Full information about the inference process can be printed out, or parts of it suppressed.

Scan gets as input a text file which specifies an elimination problem and options. Section 6.4.3 describes the content and the syntax of the **Scan** input file, and how the features described above can be used. Note that it is not possible to understand the precise effects of certain parameters, especially those influencing the inference process of **Scan**, without the algorithms in the next section.

6.3 Implementation of Scan

In this section, we will sketch how **Scan** works, which will facilitate the explanation of its user interface afterwards.

We will briefly describe the input to **Scan** and its main computation phases.

Scans input can be subdivided into two major parts:

- The logical part: A representation of a second-order formula, represented by a first-order formula and lists of quantified predicates and functions
- The control part: Optional control information for SCAN and UNSKOLEMIZE and optional control information for the user interface

Scan manages the logical information about an elimination problem in mainly three clause lists: **Sos**, **Usable** und **Demodulators**. **Sos** contains the clause form of the formula with the quantified predicates. On this list, SCAN operates by drawing inferences. **Usable** contains clauses that are only used for subsumption and unit-deletion of clauses in **Sos**. This list is not modified. **Demodulators** contains equalities that can be used for rewriting clauses in **Sos**. Whether it may be modified is determined by the user: equalities that are generated during the search can be automatically added to **Demodulators**.

The management of control information is not important at this level of description; in the following sketch of the main **Scan** algorithm, we will refer to this information as "options".

Some remarks on the algorithm:

- The variable $ElMode$ is part of the options and determines the order in which predicates are eliminated: Either one after the other, and according to their position in $PredList$, or in arbitrary order, leaving free choice of the next pure clause to SCAN.
- The procedure **PurifyLit** and **RemoveRedundancy** do roughly what one expects, their interns are described below. So is the procedure **PreProcessInput**, which basically normalizes the input with respect to some simplification rules.
- We write '&' in front of parameters to denote a call by reference which may change the parameter

See figure 6.1 for the algorithm.

Algorithm 6.3.1 (Scan)

```
list<clause> sos, usable;
list<predicate> PredList;
clause PureCl;
literal PureLit;
formula Result;
//  read information from input file
ReadInput(&Sos, &Usable, &PredList, &FnList, &Options)
//  process Sos and Usable
ProcessInput(&Sos, &Usable);
// Apply extended purification rule to Sos
XPurify(Sos, &PredList);
// Get first clause to be purified
SelectFirstPureLit(&PureCl, &PureLit, ElMode, &PredList, &found);
// as long as there is a literal to be purified ....
While (found){
      PurifyLit(PureCl, PureLit);
      XPurify(Sos, &PredList);
      SelectNextPureLit(&PureCl, &PureLit,
                        ElMode, &PredList, &found); }
// remove redundant clauses in the Sos
RemoveRedundancy(&Sos);
// eliminate skolem functions
Let Result = Usk(&Sos, FnList);
Output(Result);
```

Figure 6.1: The basic algorithm of **Scan**

Algorithm 6.3.2

RemoveRedundancies(**list<clause>** &**Sos**)
list<clause> *Passed*; // clauses that passed the test
clause *Goal*; // the clause to be proven
// Sort clauses according to their length, longest to the beginning
SortByLength(**Sos**)
While (!**Empty**(**Sos**)){
  Let *Goal* = **Car**(**Sos**);
  Let Sos = **Cdr**(**Sos**);
  If (*Goal* can be proven from **Sos** in a limited time)
    then Let *Passed* = **Append**(**Sos**, *Goal*); }
Let Sos = *Passed*;
Return;

Figure 6.2: The redundancy check of **Scan**

RemoveRedundancies The procedure **RemoveRedundancies** uses a simple but sometimes quite effective check for redundant clauses. It tries to prove each clause from the rest of the clause set, see figure 6.2. For the proof of a clause, OTTER is invoked for some time set by the user in the options, see 6.4.3.4. The details of this option are described in 6.5.3.

PurifyLit The procedure **PurifyLit** purifies a literal in a clause and checks generated C-resolvents for redundancies. Its redundancy check **PreProcessCl**, again explained below, is an extension of the OTTER procedure **pre_process**, see [McC90] and 6.5. Actually, it is not only a redundancy check but also sometimes infers new clauses. The procedure **PostProcessList** generates C-factors and extends the OTTER procedure **post_process_all**. See figure 6.3 for the algorithm.

Note that it is not necessary to check whether a factor of a resolvent *Res* back subsumes *PureCl*, because then *Res* also back subsumes *PureCl*, and this has been checked before.

PreProcessInput, PreProcess and PostProcess The procedure **PreProcess** takes a clause and a clause list as parameters. It adds the clause to the set if it passes several redundancy tests. Figure 6.4 describes the algorithm.

The procedure **PreProcessInput** takes clauses from the clause list given

Algorithm 6.3.3

PurifyLit(**clause** $PureCl$, **literal** $PureLit$)
list<clause> $Kept$;
// Flag for: PureCl has been back subsumed by one of its resolvents
boole $PureBackSubsumed$;
Let $Kept =$ **Nil**;
Resolve $PureCl$ with every clause in **Sos**
For (Every resolvent Res in **Sos**) **Do** {
  // check redundancy, stop if $PureCL$ is back subsumed
  // non-redundant clauses are added to $Kept$
  PreProcess($\&Res, \&Kept$);
  If (Res has been kept and back subsumes $PureCl$)
    then ExitLoop; }
// Now generate factors and append them to kept
PostProcess($\&Kept$);
Append($\&Sos, \&Kept$);
Return;

Figure 6.3: Purification of literals

as parameter and again adds them to the clause set as if they where inferred clauses. That is, the clauses are filtered by the procedure **PreProcess**.

In the following algorithms, all steps that are optional are marked by '*'.

Algorithm 6.3.4 (PreProcess)

```
    boole PreProcess( clause Res, list&lt;clause&gt; ClList)
    Push constraints of Res into its literals;
*   Demodulate Res;
*   Orient equalities;
*   // Delete literals in Res that can be matched with units in Sos
*   UnitDel(Sos, Res);
    Merge identical literals in Res;
*   Apply factor simplification to Res;
    If (Res is a tautology)
        then Return(false);
*   Sort literals;
*   If (Res is subsumed by some clause in Sos or Usable)
        then Exit();
*   If (Res can be proven from Sos)
      then Exit();
    If (a stop condition has been fulfilled)
    // Especially if time limit has been exceeded
      then Exit();
    // Now keep clause
    Append(Res, ClList);
    If (|Res|) = 1)
        then{ Check for unit conflict with other unit literals,
                add empty clause if proof has been found; }
        If (Res is the empty clause or a proof has been found previously)
        then Print the proof;
*   If possible, make Res a demodulator;
*   Return(true);
```

Figure 6.4: Preprocessing of generated clauses

The proof checking is another inheritance from OTTER. It helps to detect unsatisfiable clause sets more quickly.

Finally, the procedure **PostProcess**, see figure 6.5, finishes the process-

Algorithm 6.3.5 (PostProcess)

> **boole PostProcess**(**list<clause>** &$Kept$)
> **For** (all clauses C in $Kept$) **Do** {
>   **If** (C has been made a demodulator by **PreProcess**)
>   Back demodulate clauses in **Sos** and **Usable** ;
>   Discard clauses in **Sos** that are (back) subsumed by Res;
>   // Generate all C-factors
>   **For** (all C-factors $CFac$ of C) **Do** {
>   // Append C-factors if they pass retention test;
>   **PreProcess**(&$CFac$, &$Kept$); }}

Figure 6.5: Postprocessing of generated clauses

ing of a clause by generating all its C-factors, and doing back subsumption and back demodulation.

6.4 The user interface

The user interface of **Scan** has been taken over completely from OTTER, with only minor modifications. Similarly, we could have taken over the OTTER documentation of the user interface with some modifications. However, this would have resulted in quite a lot of patching and referencing, so we decided to write a new introduction. This might slow down frequent OTTER users, but also speed up those who are unacquainted with OTTER.

Scan can be used quickly and intuitively, though the documentation of all its features is quite big and its full syntax quite complex. Therefore, we will start with two small example that illustrate how to eliminate function and predicate quantifiers with **Scan**.

6.4.1 A quick example

Let the formula

$$\exists P \forall x \exists y \; (P(x) \vee R(x,y)) \wedge (Q(x,y) \wedge \neg P(a))$$

be given, which we want to reduce to first-order logic. A corresponding input file for **Scan** might look like this:

Example 6.4.1

```
% set \so{} variable lists
pred_list[P(_)]. % specify predicates to be eliminated

% set options
% flags
set(unskolemize). % activate unskolemization
% parameters
assign(minimize_at_end_time,1). % try to prove clauses in result redundant,
                                % one second for each proof attempt
% write result into log file name
logfile("doc1_example.log").

% elimination problem in formula form

formula_list(sos).
(all x exists y ( (P(x) | R(x,y)) & (Q(x,y) & -P(a)) )).
end_of_list.
```

We start **Scan** with

Scan < scan_example1.in > scan_example1.out.

At the end of the output file 'scan_example1.out' or in the log file, we find the result:

```
% result of Scan:
list(usable).
2 [] Q(x,$f1(x)).
4 [binary_c_res,1.1,3.1] R(a,$f1(a)).
end_of_list.

% result of unskolemization:
(all x0 exists e0 (Q(x0,e0) & (R(x0,e0) | x0 != a)))
% Clause set unskolemized.
```

The SCAN algorithm may not terminate. Like in theorem proving, one sometimes has to be patient to get a solution. The UNSKOLEMIZE algorithm always terminates, but may fail to remove the quantified functions.

The rest of the output file contains information about SCAN's inference process. The section 6.4.3.4 describes how to control the amount of information that is given.

6.4.2 Using Scan: The shell level

Scan takes as input an elimination problem, whose specification is described in the following sections . Then it applies SCAN or UNSKOLEMIZE or both algorithms to the input, and outputs information about the computation process of the selected algorithms. Finally, **Scan** outputs the results of the algorithms.

Input is read from the standard input, which normally is the keyboard, and written to standard output, which normally is the screen. In practice, the input will of course usually be coded in a file, and the output should be available in file form, too. In this situation, the standard input and output streams must be redirected. On a Unix system, an appropriate **Scan** invocation reads like this:

$$\text{Scan} < \text{inputfile} > \text{outputfile}.$$

The mere result, without any information about the computation process, can be output on a file with a special command, see the following 6.4.3.

6.4.3 The input file: Specifying elimination problems

This section describes in general terms how to use **Scan** input files for specifying elimination problems. It should be easily understood with the example 6.4.1 in mind. In describing the input file, we will follow OTTER terminology.

The input file contains

- commands for specifying control information, and
- lists of predicates, functions, or logical statements in clause or formula form specifying logical information.

The order in which information is given in the input file is usually not relevant.

Control information Roughly speaking, control information can either concern the user interface or the elimination algorithms and their output etc.

Control parameters for algorithms tell **Scan**

- which algorithm, SCAN or UNSKOLEMIZE to apply to the logical input
- which inference strategy to select
- which inference rules to use
- which resources like time and memory to limit
- how to index clauses for retrieval
- which information to output

The **Scan** control parameters for algorithms are listed and explained in 6.4.3.4.

Parsing control information tells the parser

- the declaration of operators
- whether to check for use of symbols with multiple arities
- what terms to recognize as variables in clauses

A.5 explains how to declare and use operators. The remaining parsing control parameters are described in 6.4.3.3.

Commands have the form $<\!\mathit{command\_name}\!>(<\!\mathit{par}_1\!>, \ldots, <\!\mathit{par}_n\!>)$. Like OTTER, **Scan** understands only a few commands:

- **set**, **clear** and **assign**, to specify options specify beginning and end of clause and formula lists
- **include**(<*file_name*>) to include another file.

Scan distinguishes two types of options, namely flags and parameters. Flags are boolean-valued options, that are activated by the command **set**(<*flag_name*>) and deactivated by **clear**(<*flag_name*>) Parameters are integer-valued options, that are assigned a value by

$$\mathbf{assign}(<\!\mathit{parameter\_name}\!>, <\!\mathit{value}\!>).$$

Logical information Lists contain the first-order part of an elimination problem, and contain either logical statements, which represent the first-order formula of a second-order formula, or function or predicate variables, which are then interpreted as quantified in the lists with logical statements.

The list **Sos** contains the first-order part of a formula. **Usable** contains clauses that are used for subsumption, but not for inferring new clauses, and **Demodulators** contains a list of demodulators. For the use of demodulators, we refer to the OTTER manual [McC90].

Formula and clause lists have the form

list(<*list_name*>).<*sequence_of_clauses*> **end_of_list**.
or **formula_list**(<*list_name*>).<*sequence_of_formulas*> **end_of_list**.

where

<*sequence_of_clauses*> $\rightarrow \varepsilon$ | <*clause*>. <*sequence_of_clauses*>
<*sequence_of_formulas*> $\rightarrow \varepsilon$ | <*formulas*>. <*sequence_of_formulas*>

Possible list names are **Sos**, **Usable** and **Demodulators**.

Clauses and formulas can be written down quite naturally. Formulas must always be closed. We will give an example below, and their exact syntax in 6.4.3.2.

Finally, predicate and function lists contain predicates and functions with dummy arguments, marked by '_':

skolem($[f(\_,\_), g(\_)]$). for skolem functions
pred_list($[P, Q(\_,\_,\_), R(\_)]$). for predicate variables

6.4.3.1 General syntax

An input file is a sequence of OTTER terms which must be terminated by a point '.'. Formulas, clauses and commands are specializations of this term type. OTTER terms may contain unary and binary operators, for which precedence and notation type (prefix,infix, postfix) can be declared as described in A.5. Comments reach from '%' to the end of the line.

Terms are called complex when they have subterms, and simple, when they have none.

6.4.3.2 Syntax of formulas and clauses

Formulas and clauses are built up as usual from atoms and a set of logical operators and can be written down quite naturally. Clauses are disjunctions of literals without explicit quantification of variables, while all variables in formulas have to be quantified. This raises the question how variables can be told from constants in clauses. OTTER solves this problem by assuming all arguments that start by u,v,w,x,y or z to be variables. As an example:

```
% a is a constant
list(sos). P(x,y) | Q(a) | R(f(b,y2)). end_of_list.
% a is a variable
formula_list(sos).
all a x y y2 (P(x,y) | Q(a) | R(f(b,y2))).
end_of_list.
```

The exact syntax of formulas and clauses can be found in the appendix. The text has been copied almost literally from the OTTER user manual.

6.4.3.3 Parsing control parameters

Option	Type	Default	Description
check_arity	Flag	set	If set, all symbols in the input file must have fixed arity
prolog_style_variables	Flag	clear	If set, terms without arguments in clauses are identified as a variable iff the start with A through Z (upper case) or with _.

6.4.3.4 Control parameters

Selecting algorithms **Scan** applies SCAN and UNSKOLEMIZE one after the other to the input formula sets. Both algorithms can separately be enabled and disabled.

Option	Type	Default	Description
scan	Flag	set	If set, apply SCAN to input
unskolemize	Flag	clear	If set, apply UNSKOLEMIZE to output of SCAN

Inference strategy The choice among inference strategies is not so big yet. As described in 6.3, the user can choose to eliminate the predicates one after another as they occur in the list of predicate variables, leave the choice to **Scan**.

Option	Type	Default	Description
succ_elimination	Flag	clear	If set, eliminate predicate variables one after another as they are listed. If clear, elimination order is arbitrary

Inference rules The **Scan** optional inference rules fall roughly into two categories: rules for deriving new clauses, rules for simplifying the clause set or single clauses in some way, and rules for normalizing clauses in some way. Apart from clauses resulting from demodulation, new clauses are generated by the C-factoring and binary C-resolution rule. When the C-resolution rule is disabled, the SCAN algorithm is disabled. Simplifications are done by forward and backward subsumption, unit deletion and factor simplification. Forward subsumption is available in a simple and an extended form, as described in 6.3. Optional normalizing is restricted to orienting equalities and sorting of literals.

Unlike in theorem provers, it is usually a good choice to activate all available inference rules of **Scan**, and they are activated by default, except for extended forward subsumption, as it is quite time consuming. However, they can be deactivated for special purposes, for example to achieve different but equivalent results. Figure 6.6 lists options concerning inference rules.

Option	Type	Default	Description
binary_c_res	Flag	Set	If set, use binary C-resolution
c_factor	Flag	Set	If set, apply C-factorization and factor simplification
unit_deletion	Flag	Set	If set, apply unit deletion
check_redundancy	Param	0	set time for redundancy check of a generated clause in seconds.
sort_literals	Flag	Clear	sort literals of generated clauses: first negative literals, then positive literals. May speed up subsumption.

Figure 6.6: **Scan** inference rules

Resources The amount of time and space that **Scan** may consume can be roughly limited. **Scan** stops more precisely than OTTER when resource limits are exceeded. See figure 6.7.

Option	Type	Default	Description
max_seconds	Param	-1	If value is -1, then there is no time limit for **Scan**, otherwise it stops after n seconds, where n is the integer value of max_seconds
max_mem	Param	-1	If value is -1, then there is no memory limit for **Scan**, otherwise it stops before more than n kilobytes have been allocated, where n is the integer value of max_mem

Figure 6.7: **Scan** resource control options

Output The main purpose of **Scan**'s output is to show its result, and to document every inference step, thus making its result verifiable. The output parameters are listed in figure 6.8.

Indexing Indexes allow for quicker forward and backward subsumption und unit deletion. SCAN has inherited two index types from OTTER: FPA indexing and discrimination tree indexing (See also 6.5). While back subsumption is always done with discrimination trees, the user has the choice in the case of forward subsumption.

FPA indexing is usually much slower than discrimination tree indexing, while discrimination tree indexing may require a lot more memory then FPA indexing. The index type has no influence on the result of SCAN, especially not on its termination.

For FPA indexing, the accuracy of retrieval results can be influenced by the indexing depth. A higher indexing depth needs more memory and more time for indexing, but retrieval can be much faster. Figure 6.9 lists indexing options.

Option	Type	Default	Description
very_verbose	Flag	clear	If set, output much information about inference process
print_given	Flag	clear	If set, print selected pure clause
print_kept	Flag	set	If set, print generated clauses that are actually added to a clause set
print_back_sub	Flag	set	If set, print clauses that are back subsumed. During input processing, back subsumed clauses are always printed (see 6.5)
print_new_demod	Flag	set	If this flag is set, clauses are printed as they are back demodulated. Back-demodulated clauses are always printed during input processing (see 6.5.
display_terms	Flag	clear	If this flag is set, all clauses and terms are printed in pure prefix form (Sec. A.2).
pretty_print	Flag	clear	If this flag is set, clauses are output in an indented form that is sometimes easier to read. The parameter **pretty_print_indent** (default 4) specifies the number of spaces for each indent level.
bird_print	Flag	clear	If this flag is set, terms constructed with the binary function a are output in combinatory logic notation (without the function symbol a, and left associated unless otherwise indicated). For example, the clause $a(a(a(S,x),y),z) = a(a(x,z),a(y,z))$ is output as $Sxyz = xz(yz)$.

Figure 6.8: SCAN output parameters

Option	Type	Default	Description
for_sub_fpa	Flag	clear	If set, use FPA indexing for forward subsumption, otherwise discrimination tree indexing
fpa_literals	parameter	8	Sets indexing depth from 0 to 100

Figure 6.9: Indexing options

6.5 Details of the implementation

6.5.1 Using OTTER for building Scan

Otter has been used in two ways for **Scan**:

- As a collection of data structures and techniques for theorem proving
- As a complete theorem prover that is used in subprocesses to prove clauses redundant

In fact, the code of **Scan** contains the full OTTER code, with some patches of course. The main part of this section will focus on the first point and show how parts of OTTER have been reused in **Scan**. This was also the more difficult part of the implementation. We will show this on an abstract level, regarding OTTER as a clean implementation of several data structures and algorithms based on these structures. This allows for a very simple presentation. However, OTTER has not been implemented that cleanly, and quite some low-level patches were necessary to use OTTER code for our purposes.

The **Scan** implementation can roughly be subdivided into three major parts:

- The user interface
- The implementation of SCAN
- The implementation of UNSKOLEMIZE

We have already mentioned that the user interface of **Scan** is nearly identical to that of OTTER. The minor modifications can be traced in the code. The implementation of the UNSKOLEMIZE algorithm uses its own data structures, especially an own term data structure. Communication with the OTTER parts is provided by conversion routines for both term types. Therefore, we will restrict ourselves to describing the implementation of SCAN using OTTER algorithms.

6.5.2 Using OTTER algorithms and data types for implementing SCAN

Scan uses many of OTTER's data types. In contrast to some of the reused OTTER algorithms, none of them had to be modified. For implementation of SCAN, the OTTER clause management and its inference process is central, so let us first have a look at them.

OTTER clause management OTTER's main data type for drawing inferences is the clause data type. Though there is a formula data type for representing input formulas, these formulas are converted into clause form before the inference process starts. OTTER's uses structure sharing on terms for efficient clause storage and term indexing for efficient clause retrieval in unification problems. For different purposes, several indexing techniques are available, namely FPA-indexing and discrimination tree indexing. [1]

OTTER's inference process The structure of OTTER's inference process is quite simple. We present it under the aspect of the OTTER clause management, which is important for understanding its reuse in **Scan**. Also, we reduce OTTER to a simple binary resolution system with factoring, which can in fact be done by setting options accordingly. This was the starting point of the original code analysis: We wanted to obtain **Scan** simply by to replacing ordinary resolution with C-resolution and factoring with C-factoring. Figure 6.10 describes OTTER's inference process.

The processing of newly generated clauses in OTTER, like in SCAN is divided into the pre- and postprocessing phase. Both will be described more precisely in the next paragraph. Roughly speaking, preprocessing, which is done by the **OtterPreProcess**-Procedure, includes demodulation,unit-deletion and checks whether the resolvent is forward subsumed by a clause

[1] Structure sharing of terms tries to save space by storing common subterms of several terms only once. Term indexing is a means for quick retrieval of terms in unification problems. By techniques that may also be seen as structure sharing, term sets are organized in such a way that it is possible to quickly retrieve, say, all the terms in the set unifying with or matching $f(x, g(y))$.

Algorithm 6.5.1

Read input;
Transform input into clause lists **Sos** and **Usable**;
Integrate and index all clauses
While (**Sos** is not empty and no refutation has been found)
        Pick the "lightest" clause given_clause from **Sos**
        While (There are resolution partners forgiven_clause)
            Get a resolution partner for given_clause from **Usable**
            using the term index
            and generate the resolvent *Res*;
            // Now try several redundancy tests on *Res*.
            // If it passes all of them, append it to **Sos**,
            // index and integrate it.
            OtterPreProcess(*Res*); // (Details below)
        // After all resolvents with given_clause
        // have been made, build factors and do back subsumption
        For (All generated resolvents *Res*) **Do** {
            OtterPostProcess(*Res*)} // (Details below)

Figure 6.10: OTTER's inference process

in **Sos**,**Usable** or **Passive**. Then the resolvent is integrated. The postprocessing (**OtterPostProcess**) eliminates clauses from **Sos** and **Usable** that are back subsumed by the new clause and generates its factors, which are in turn pre- and postprocessed. Back subsumed clauses are unindexed and disintegrated. The separation into pre- and postprocessing is in part a consequence of the indexing and structure sharing of terms: Since term integration is expensive, a clause is only integrated and indexed if it is surely kept and not changed anymore.

Modifying OTTER **to implement** SCAN As we've said, the original approach to reusing OTTER for a primitive implementation of SCAN was: Replace standard binary resolution with C-resolution and standard factorization with C-factorization. Or rather, extend OTTER with these new inference rules by copying and modifying the code for the similar old ones. However, the simplified set-of-support inference strategy that OTTER uses when the *given_clause* has been selected is not compatible with the SCAN algorithm, as the *given_clause* must be resolved with all clauses, not only those in **Usable**, especially not with itself. Therefore, the inference algorithm of SCAN has been implemented independently of the OTTER inference algorithm, but using the OTTER clause management. The OTTER processing of inferred clauses could be used with some modifications, on the basis of the original OTTER data structures for clause management. We will describe the most important of those modifications in the following.

It has been possible to modify OTTER so smoothly that the new pre- and postprocessing facilities are compatible with all other OTTER processing facilities. Therefore, the OTTER facilities can still be used by setting options that influence pre- and postprocessing. If they are not mentioned in this documentation, they did not seem very useful in the context of quantifier elimination, or only in very special cases or with much knowledge about OTTER. Especially, we do not describe the use of user-defined evaluable predicates, which allow for very refined control. Still, all these facilities can be used exactly in the same way as they are described in the OTTER manual if it turns out to be useful.

We begin with the modifications to OTTER's preprocessing. We leave out computation steps that are not relevant in this context. These are mainly the evaluation of user-defined predicates and the deletion of clauses whose user-defined weight has exceeded a certain limit or who contain identical nested skolem functions. A full account of the OTTER preprocessing steps can be found in the OTTER manual.

We look backwards and obtain **OtterPreProcess** from **Scan**'s **PreProcess** as described in 6.3 by simply deleting three steps:

- Constraint elimination
- Extended forward subsumption of clauses that can be proven from **Sos**
- The check for stop conditions

Constraint elimination and extended forward subsumption are of course required by the SCAN algorithm. The new check for stop conditions was not necessary, but reasonable to obey time limits more precisely. This is an obvious advantage anyway, but has especially been motivated by the timing of the extended forward subsumption. For details, see section 6.5.3.

During the preprocessing in both OTTER and SCAN, kept clauses are integrated and indexed. This is done after the kept clause has been appended to **Sos**.

OTTER's postprocessing is very similar to **Scan**'s. Only the standard factorization of OTTER has been replaced by the C-factorization necessary for **Scan**.

6.5.3 Using OTTER for proving clauses redundant

For the extended forward subsumption test and the minimization of the clause set at the end of the elimination process, **Scan** uses OTTER's theorem proving abilities. We describe the case of extended forward subsumption. Minimization is done in a similar, though not identical way.

Given a newly generated clause C, OTTER has to be invoked with the current **Sos** of **Scan** and the negation of C in its own **Sos**. Of course, this could be done by using the OTTER user interface, that is, by writing the input clause set to a file, starting OTTER on this input and extracting the result from the output file. However, this is quite a waste of time, as the clause set already has the internal representation that is used by OTTER. Especially, it is already indexed and integrated, that is, the OTTER inference process could at once start on a memory copy of this clause set (one copy has of course to be kept).

However, things are not that simple. Though this solution is possible, it requires a lot of changes in the OTTER sources. We made a compromise which turned out to be quite effective. It depends on the multitasking architecture of Unix systems.

When a new clause C has to be checked for redundancy, a new subprocess of the **Scan** process is generated which is an exact copy of its father process by the fork() system command. In particular, it inherits a copy of the **Sos** of **Scan**. Essentially, these clause sets are now emptied, copied and filled with the clauses from their copies again as if they were input clauses. Before, some OTTER options are set for activating options that are deactivated during quantifier elimination. Especially, OTTER's auto mode is activated, which analyses input clause sets and automatically selects inference rules which seem appropriate. Then OTTER is run for a limited time given by the parameter **check_redundancy_time**. The result of its search is communicated to its father process, and C is eliminated if OTTER has found a proof.

This method required a change in the OTTER preprocessing which has been mentioned in the previous section. The original OTTER checks its stop conditions, in particular its computation time limit, after all resolvents of the *given_clause* have been postprocessed. Recall that postprocessing of these resolvents means that all their factors are generated and again pre- and postprocessed. This can take a lot of time, during which the time limit is not checked and may well be exceeded. This inaccuracy is of course multiplied when **Scan** invokes OTTER every time a new clause is generated. To keep the behavior of **Scan** a little more predictable, now OTTER's modified preprocessing checks the stop conditions. During postprocessing, all generated factors are preprocessed, so that the time limit is now obeyed there.

6.6 Modules and source files

As OTTER is not cleanly modularized, and due to some necessary patches, clean modularization of **Scan** would only have been possible with structural changes of the source files much bigger than the current. However, modulo some patches, the whole implementation can roughly be subdivided into the OTTER, SCAN and UNSKOLEMIZE. Source files are divided into pure OTTER files, patched OTTER files, OTTER file patch files (!), files for the implementation of SCAN and files for the implementation of UNSKOLEMIZE. The file type can easily be identified:

- Files with the prefix 'scan', like 'scan_Proving.c' are part of the implementation of SCAN
- Files with the prefix 'us', like 'us_RelationVnPos' are part of the implementation of UNSKOLEMIZE

- Files with no prefix (without underscore in the name), like 'main.c' belong to OTTER. They are patched if a corresponding patch file exists, and pure otherwise
- Files with the prefix 'Patch' contain patches for the OTTER file named in the suffix, like 'Patch_misc.c' which patches 'misc.c' of OTTER

Lines of OTTER files that have been modified or added are marked by the label 'SCAN_PATCH' or, in the case of complicated patches, preceded by this label.

6.7 Installation

The installation of **Scan** is described in the INSTALL file that comes with the sources.

Appendix A

Syntax of formulas and clauses in Scan

A.1 Names for Variables, Constants, Functions, and Predicates

Three kinds of character string, collectively referred to as *names*, can be used for variables, constants, function symbols, and predicate symbols:

- An *ordinary name* is a string of alphanumerics, $, and _.
- A *special name* is a string of characters in the set *+-/\^<>=`~:?@&!;# (and sometimes |).
- A *quoted name* is any string enclosed in two quotation marks of the same type, either " or '. We have no trick for including a quotation mark of the same type in a quoted name.

(The reason for separating ordinary and special names has to do with infix, prefix, and postfix operators; see Sec. A.5 in the Otter manual.) Although out of place here, for completeness we list the meanings of the remaining printable characters.

- . (period) — terminates input expressions.
- % — starts a comment (which ends with the end of the line).
- ,()[]{} (and sometimes |) — are punctuation and grouping symbols.

Variables. Determining whether a simple term is a constant or a variable depends on the context of the term. If it occurs in a clause, the symbol determines the type: the default rule is that a simple term is a variable if it starts with `u`, `v`, `w`, `x`, `y`, or `z`. If the flag `prolog_style_variables` is set, a simple term is a variable if and only if it starts with an upper-case letter or with `_`. (Therefore, variables in clauses must be ordinary names.) A simple term in a formula is a variable if and only if it is bound by a quantifier.

Reserved and Built-in Names. Names that start with `$` are reserved for special purposes, including evaluable functions and predicates, answer literals and terms (see OTTER manual), and some internal system names. The name `=` and any name that starts with `eq`, `EQ`, or `Eq`, when used as a binary predicate symbol, is recognized as an equality predicate by the demodulation and paramodulation processes. And some names, when they occur in clauses or formulas, are recognized as logic symbols.

Overloaded Symbols. The user can use a name for more than one purpose, for example as a constant and as a 5-ary predicate symbol. When the flag `check_arity` is set (the default), the user is warned about such uses. Some built-in names are also overloaded; for example, `|` is used both for disjunction and as Prolog-style list punctuation, and although `-` is built in as logical negation, it is generally used for both unary and binary minus as well.

A.2 Terms and Atoms

Recall that, when interpreted, terms are evaluated as objects in some domain, and atoms are evaluated as truth values. Constants and variables are terms. An n-ary function symbol applied to n terms is also a term. An n-ary predicate symbol applied to n terms is an atom. A nullary predicate symbol (also referred to as a propositional variable) is also an atom.

The pure way of writing complex terms and atoms is with *standard application*: the function or predicate symbol, opening parenthesis, arguments separated by commas, then closing parenthesis, for example, `f(a,b,c)` and `=(f(x,e),x)`. If all subterms of a term are written with standard application, the term is in *pure prefix form*. Whitespace (spaces, tabs, newlines, and comments) can appear in standard application terms anywhere *except* between a function or predicate symbol and its opening parenthesis. If the flag `display_terms` is set, OTTER will output terms in pure prefix form.

Infix Equality. Some binary functors can be written in infix form; the most important is `=`. In addition, a negated equality, `-(a=b)` can be abbreviated `a!=b`.

List Notation. Prolog-style list notation can be used to write terms that represent lists. Table A.1 in the OTTER manual gives some example terms in list notation and the corresponding pure prefix form. Of course, lists can

`[]`	`$nil`
`[x|y]`	`$cons(x,y)`
`[x,y]`	`$cons(x,$cons(y,$nil))`
`[a,b,c,d]`	`$cons(a,$cons(b,$cons(c,$cons(d,$nil))))`
`[a,b,c|x]`	`$cons(a,$cons(b,$cons(c,x)))`

Table A.1: List Notation

contain complex terms, including other lists.

A.3 Literals and Clauses

A literal is either an atom or the negation of an atom. A clause is a disjunction of literals. The built-in symbols for negation and disjunction are `-` and `|`, respectively. Although clauses can be written in pure prefix form, with `-` as a unary symbol and `|` as a binary symbol, they are rarely written that way. Instead, they are almost always written in infix form, without parentheses. For example, the following is a clause in both forms.

Pure prefix:	`|(-(a),|(=(b1,b2),-(=(c1,c2))))`
Infix (abbreviated):	`-a | b1=b2 | c1!=c2`

Otter accepts both forms. (Clauses are parsed by the general term-parsing mechanism presented in Sec. A.5).

A.4 Formulas

Table A.2 lists the built-in logic symbols for constructing formulas.

Formulas in Pure Prefix Form. Although the practice is rarely done, formulas can be written in pure prefix form. Quantification is the only tricky part: there is a special variable-arity functor, `$Quantified`, for quantified formulas. For example, $\forall xy \exists z (P(x,y,z) | Q(x,z))$ is represented by

negation	`-`
disjunction	`|`
conjunction	`&`
implication	`->`
equivalence	`<->`
existential quantification	`exists`
universal quantification	`all`

Table A.2: Logic Symbols

```
$Quantified(all,x,y,exists,z,|(P(x,y,z),Q(x,z))).
```

If the flag `display_terms` is set, the formulas (and everything else) will be displayed in pure prefix form.

Abbreviated Formulas. Formulas are usually abbreviated in a natural way. The associativity and precedence rules for abbreviating formulas and the mechanism for parsing formulas are presented in Sec. A.5. Here are some examples.

standard usage	OTTER syntax (abbreviated)
$\forall x P(x)$	`all x P(x)`
$\forall xy \exists z(P(x,y,z) \vee Q(x,z))$	`all x y exists z (P(x,y,z) | Q(x,z))`
$\forall x(P(x) \wedge Q(x) \wedge R(x) \rightarrow S(x))$	`all x (P(x) & Q(x) & R(x) -> S(x))`

Note that if a formula has a string of identical quantifiers, all but the first can be dropped. For example, `all x all y all z p(x,y,z)` can be shortened to `all x y z p(x,y,z)`. In expressions involving the associative operations `&` and `|`, extra parentheses can be dropped. Moreover, a default precedence on the logic symbols allows us to drop more parentheses: `<->` has the same precedence as `->`, and the rest in decreasing order are `->`, `|`, `&`, `-`. Greater precedence means closer to the root of the term (i.e., larger scope). For example, `p | -q & r -> -s | t` represents `(p | (-(q) & r)) -> (-(s) | t)`, or in pure prefix form, `->(|(p,&(-(q),r)),|(-(s),t))`.

When in doubt about how a particular string will be parsed, one can simply add additional parentheses and/or test the string by having OTTER read it and then display it in pure prefix form. The following input file can be used to test the preceding example.

```
assign(stats_level, 0).
```

```
set(display_terms).
formula_list(usable).
p| -q&r-> -s|t.        % This formula has minimum whitespace.
end_of_list.
```

In general, whitespace is required around `all` and `exists` and to the left of `-`; otherwise, whitespace around the logic symbols can be removed. See Sec. A.5 for the rules.

A.5 Infix, Prefix, and Postfix Expressions

Many Prolog systems (for example Quintus and Sicstus) have a feature that allows users to declare that particular function or predicate symbols are infix, prefix, or postfix and to specify a precedence and associativity so that parentheses can sometimes be dropped. OTTER has a similar feature. In fact, the clause and formula parsing routines use the feature. Users who use only the predeclared logic operators for clauses and formulas and the predeclared infix equality `=` can skip the rest of this section.

Prolog users who are familiar with the declaration mechanism should note the following differences between the Quintus/Sicstus mechanism and OTTER's.

- The predeclared operators are different. See Table A.3.
- OTTER does not treat comma as an operator; in particular, `a,b,c` cannot be a term, as in `a,b,c -> d,e,f`.
- OTTER treats the quantifiers `all` and `exists` as special cases, because they don't seem to fit neatly into the standard Prolog mechanism.
- OTTER requires whitespace in some cases where the Prologs do not.

Functors to be treated in this special way are given a type and a precedence. Either OTTER predeclares the functor's properties, or the user gives OTTER a command of one of the forms

```
op(precedence, type, functor).
op(precedence, type, list-of-functors).
```

The `precedence` is an integer i, $0 < i < 1000$, and `type` is one of the following: `xfx`, `xfy`, `yfx` (infix), `fx`, `fy` (prefix), `xf`, `yf` (postfix). See Table A.3 for the commands corresponding to the predeclared functors.

```
op(800, xfx, ->).    op(700, xfx, @<).
op(800, xfx, <->).   op(700, xfx, @>).
op(790, xfy, |).     op(700, xfx, @<=).
op(780, xfy, &).     op(700, xfx, @>=).

op(700, xfx, =).     op(500, xfy, +).
op(700, xfx, !=).    op(500, xfx, -).

op(700, xfx, <).     op(500, fx, +).
op(700, xfx, >).     op(500, fx, -).
op(700, xfx, <=).
op(700, xfx, >=).    op(400, xfy, *).
op(700, xfx, ==).    op(400, xfx, /).
op(700, xfx, =/=).   op(300, xfx, mod).
```

Table A.3: Predeclared Functors

Given an expression that looks like it might be associated in a number of ways, the relative precedence of the operators determines, in part, how it is associated. A functor with higher precedence is more dominant (closer to the root of the term), and one with lower precedence binds more tightly. For example, the functors `->`, `|`, `&`, and `-` have decreasing precedence; therefore the expression `p & - q | r -> s` is understood as `((p & (-q)) | r) -> s`.

In each of the types, `f` represents the functor, and `x` and `y`, which represent the expressions to which the functor applies, specify how terms are associated. Given an expression involving functors of the *same* precedence, the types of the functors determines, in part, the association. See Table A.4. The following are examples of associativity:

`xfx`	infix (binary)	don't associate
`xfy`	infix (binary)	associate right
`yfx`	infix (binary)	associate left
`fx`	prefix (unary)	don't associate
`fy`	prefix (unary)	associate
`xf`	postfix (unary)	don't associate
`yx`	postfix (unary)	associate

Table A.4: Functor Types

- If `+` has type `xfy`, then `a+b+c+d` is understood as `a+(b+(c+d))`.

- If `->` has type `xfx`, then `a->b->c` is not well formed.
- If `-` has type `fy`, then `- - -p` is understood as `-(-(-(p)))`. (The spaces are necessary; otherwise, `---` will be parsed as single name.)
- If `-` has type `fx`, then `- - -p` is not well formed.

Caution: The associativity specifications in the infix functor declarations say nothing about the logical associativity of the operation, e.g., whether `(a+b)+c` is the same object as as `a+(b+c)`. The specifications are only about parsing ambiguous expressions. In most cases, when an operator is `xfy` or `yfx`, it is also logically associative, but the logical associativity is handled separately; it is built-in in the case of the logic symbols `|` and `&` in OTTER clauses and formulas, and it must be axiomatized in other cases.

Details of the Functor Declarations. (This paragraph can be skipped by most users.) The precedence of functors extends to the precedence of expressions in the following way. The precedence of an atomic, parenthesized, or standard application expression is 0. Respective examples are `p`, `(x+y)`, and `p(a+b,c,d)`. The precedence of a (well-formed) nonparenthesized nonatomic expression is the same as the precedence of the root functor. For example, `a&b` has the precedence of `&`, and `a&b|c` has the precedence of the greater functor. In the type specifications, `x` represents an expression of lower precedence than the functor, and `y` represents an expression with precedence less than or equal to the functor. Consider `a+b+c`, where `+` has type `xfy`; if association is to the left, then the second occurrence of `+` does *not* fit the type, because `a+b`, which corresponds to `x`, does not have a lower precedence than `+`; if association is to the right, then all is well. If we extend the example, under the declarations `op(700, xfx, =)` and `op(500, xfy, +)`, the expression `a+b+c=d+e` must be understood as `(a+ (b+c))= (d+e)`.

A.6 Whitespace in Expressions

The reason for separating ordinary names from special names (Sec. A.1) is so that some whitespace (spaces, tabs, newline, and comments) can be removed. We can write `a+b+c` (instead of having to write `a + b + c`), because "`a+b+c`" cannot be a name, that is, it must be parsed into five names.
Caution. There is a deficiency in OTTER's parser having to do with whitespace between a name and opening parenthesis. The rule to use is: *Insert*

some white space if and only if it is not *a standard application.* For example, the two pieces of white space in `(a+ (b+c))= (d+e)` are required, and no white space is allowed after `f` or `g` in `f(x,g(x))`.

A.7 Bugs, etc., in Input and Output of Expressions

- The symbol `|` is either Prolog-style list punctuation or part of a special name. With the built-in declaration of `|` as infix, the term `[a|b]` is ambiguous, with possible interpretations t_1 =`$cons(a,b)` and t_2 =`$cons(|(a,b),$nil)`. OTTER recognizes `[a|b]` as t_1. The term t_2 can be written `[(a|b)]`. The bug is that t_2 will be output without the parentheses. This is the only case I know in which OTTER cannot correctly read a term it has written.

- A term consisting of a unary `+` or `-` applied to a nonnegative integer is always translated to a constant.

- Parsing large terms without parentheses, say `a1+a2+a3+...+a1000`, can be very slow if the operator is left associative (`yfx`). If you intend to parse such terms, make the operator right associative (`xyf`).

- Quoted strings cannot contain a quotation mark of the same type.

- The flag `check_arity` sometimes issues warnings when it should not.

- Braces (`{}`) can be used to group input expressions, but OTTER always uses ordinary parentheses on output.

A.8 Examples of Operator Declarations

Group Theory. Suppose we like to see group theory expressions in the form $(ab^{-1}c^{-1-1})^{-1}$, in which right association is assumed. We can approximate this for OTTER with `(a*b^ *c^ ^)^`. (We have to make the group operator explicit; `-1` is not a legal OTTER name; the whitespace shown is required.) The declarations `op(400, xfy, *)` and `op(350, yf, ^)` suffice. Other examples of expressions (with minimum whitespace) using these declarations are `(x*y)*z=x*y*z` and `(y*x)^ =x^ *y^`.

OTTER **Options.** Options are normally input as in the following examples.

```
set(prolog_style_variables).
clear(print_kept).
assign(max_given, 300).
```

If, however, we make the declarations (the precedences are irrelevant)

```
op(100, fx, set).
op(100, fx, clear).
op(100, xfx, assign).
```

then we may write

```
set prolog_style_variables.
clear print_kept.
max_given assign 300.
```

List of Figures

Bibliography

[Ack35a] Wilhelm Ackermann. Untersuchung über das Eliminationsproblem der mathematischen Logik. *Mathematische Annalen*, 110:390–413, 1935.

[Ack35b] Wilhelm Ackermann. Zum Eliminationsproblem der mathematischen Logik. *Mathematische Annalen*, 111:61–63, 1935.

[BS71] John Bell and Alan Slomson. *Models and Ultraproducts.* North Holland, 1971.

[Fra55] R. Fraissé. Sur quelques classifications des relations, basée sur des isomorphismes restreints. *Ann. Sci. Ecole Sup.*, 71:363–388, 1955.

[GO92] D. Gabbay and H.J. Ohlbach. Quantifier elimination in second-order predicate logic. In *Proceedings of the KR'92, Principle of Knowledge Representation and Reasoning*, pages 425–436. Morgan Kaufmann (1992), 1992.

[Göd30] Kurt Gödel. *Über die Vollständigkeit des Logikkalküls.* PhD thesis, University of Vienna, 1930.

[Hen61] L. Henkin. Some remarks on infinitely long formulas. In *Infinitistic Methods*, pages 167–183. Pergamon Press, Oxford, 1961.

[HET84] J. Flum H. Ebbinghaus and W. Thomas. *Mathematical Logic.* Springer, 1984.

[Hod84] Wilfried Hodges. Elementary predicate logic. In Gabbay Dov M and Franz Guenthner, editors, *Handbook of Philosophical Logic, Vol. I, Elements of Classical Logic, Synthese Library Vol. 164*, pages 2–131. D. Reidel Publishing Company, Dordrecht, 1984.

[Kei71] H.J. Keisler. *Model theory for infinitary logic.* North Holland, 1971.

[Kow75] Robert Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, 22(4):572–595, October 1975.

[Lin69] P. Lindström. On extensions of elementary logic. *Theoria*, 35, 1969.

[McC88] William W. McCune. Unskolemizing clause sets. *Information Processing Letters*, 29:257–263, 1988.

[McC90] William McCune. OTTER 2.0. In Mark Stickel, editor, *Proc. of* 10^{th} *Internation Conference on Automated Deduction, LNAI 449*, pages 663–664. Springer Verlag, 1990.

[NS95] Andreas Nonnengart and Andrzej Szalas. A fixpoint approach to second-order quantifier elimination with applications to correspondence theory. Technical report, Saarbrücken, 1995.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Sza92] Andrzej Szałas. On correspondence between modal and classical logic: Automated approach. Technical Report MPI–I–92–209, March 1992.

[vB83] Johan van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, Naples, 1983.

[vB84] Johan van Benthem. Correspondence theory. In Gabbay Dov M and Franz Guenthner, editors, *Handbook of Philosophical Logic, Vol. II, Extensions of Classical Logic, Synthese Library Vol. 165*, pages 167–248. D. Reidel Publishing Company, Dordrecht, 1984.

[vBD84] Johan van Benthem and Kees Doets. Higher-order logic. In Dov Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic, Vol. I, Elements of Classical Logic, Synthese Library Vol. 164*, pages 275–329. D. Reidel Publishing Company, Dordrecht, 1984.