

Daniel Sheridan

# **A Preprocessor for Resolution Proof Procedures**

BA Computer Science

King's College, 1999



# Proforma

**Title:** Preprocessing for Resolution Proof Procedures

**Examination:** Part II BA Computer Science

**Year:** 1999

**Word count:** 10 000

**Project originator:** Lawrence C Paulson

**Project supervisor:** Lawrence C Paulson

**Original aims:** To produce a preprocessor which takes as input a statement in general first-order logic and converts it to a clause form representation suitable for direct input to a theorem prover. For an improvement in efficiency, the software should implement the renaming scheme discussed in [NRW98] in addition to the conventional method taught in Part Ib CST [Pau97].

The software will be implemented in some dialect of Prolog. The project evaluation will compare the two methods with respect to a variety of different example problems, and discuss the efficiency tradeoff between time spent in the preprocessor and time spent in the theorem prover.

**Work completed:** A piece of software, *Clausify*, was written in Prolog implementing both the conventional and renaming methods of conversion. It was tested with a number of input problems to compare the effectiveness of the two methods.

**Special difficulties:** The project was developed simultaneously on two different Prolog platforms: *SWI Prolog* on Linux and *SICStus Prolog* on Solaris. SWI Prolog is considerably more feature-rich than the copy of SICStus on the Thor teaching machine, so a large amount of effort was put into implementing the extra library routines. These will be summarised in Appendix A and discussed in the main text as appropriate.



# Declaration

I, Daniel Sheridan of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

**Signature:**

**Date:**



# Contents

|  |            |
|--|------------|
| <b>Proforma</b>  | <b>iii</b> |
| <b>Declaration</b>   | <b>v</b>   |
| <b>1 Introduction</b>  | <b>1</b>   |
| <b>2 Preparation</b>   | <b>3</b>   |
| 2.1 Atoms and Free Variables . . . . .                                 | 3          |
| 2.2 <i>Clausify</i> Language . . . . .                                 | 3          |
| 2.2.1 Handling Quantifiers . . . . .                                   | 4          |
| 2.3 Definitions . . . . .  | 5          |
| 2.4 Renaming . . . . .   | 6          |
| 2.4.1 Selecting a Subformula . . . . .                                 | 7          |
| <b>3 Implementation</b>  | <b>11</b>  |
| 3.1 Conventional Approach . . . . .                                    | 11         |
| 3.1.1 <b>impls/2</b> : Eliminating Implications . . . . .              | 11         |
| 3.1.2 <b>negs/2</b> : Pushing Negations In . . . . .                   | 14         |
| 3.1.3 <b>cnf/2</b> : Converting to Clause Normal Form . . . . .        | 15         |
| 3.1.4 <b>quants/2</b> : Converting to Antiprenex Normal Form . . . . . | 16         |
| 3.1.5 <b>skolemize/2</b> : Skolemization . . . . .                     | 17         |
| 3.1.6 <b>alphasub/4</b> : Alpha Substitution . . . . .                 | 17         |
| 3.1.7 <b>claus/2</b> : Is this one of mine? . . . . .                  | 18         |
| 3.2 Implementing Renaming . . . . .                                    | 19         |
| 3.2.1 <b>search/9</b> : Gathering information . . . . .                | 19         |
| 3.2.2 <b>enlist/9</b> : Keeping a Record . . . . .                     | 20         |
| 3.2.3 <b>chklist/3</b> : Choosing Subformulæ . . . . .                 | 21         |
| 3.2.4 <b>rename/3</b> : Making the Changes . . . . .                   | 22         |
| 3.2.5 <b>subs/4</b> : Choosing the substitutions . . . . .             | 22         |
| <b>4 Evaluation</b>  | <b>25</b>  |
| 4.1 Sample Output . . . . .  | 25         |
| 4.2 Performance in Conventional Mode . . . . .                         | 26         |
| 4.3 Conventional Mode versus Renaming . . . . .                        | 27         |
| <b>5 Conclusions</b>   | <b>31</b>  |

|  |           |
|--|-----------|
| <b>A Library Routines</b>                                  | <b>33</b> |
| A.1 <b>append/3</b> : Concatenating Lists . . . . .        | 33        |
| A.2 <b>member/2</b> : List Membership . . . . .            | 33        |
| A.3 <b>delete/3</b> : Removal From Lists . . . . .         | 33        |
| A.4 <b>reverse/2</b> : List Reversal . . . . .             | 34        |
| A.5 <b>gensym/2</b> : Generating a Unique Symbol . . . . . | 34        |
| <b>Bibliography</b>  | <b>35</b> |
| <b>Index</b>   | <b>37</b> |
| <b>Project Proposal</b>                                    | <b>39</b> |

# Chapter 1

## Introduction

*An overview of the project; why we need conversions to clause form and how they can be made more efficient. A brief overview of renaming. What software was produced in the course of the project.*

---

Resolution style theorem provers such as *SETHEO* and *Otter* perform transformations on inputs in clause form, *i.e.* as a list of disjunctions of literals

$$\neg K_1 \vee \neg K_2 \vee \dots \vee \neg K_n \vee L_1 \vee L_2 \vee \dots \vee L_n$$

usually written as a set:

$$\{\neg K_1, \neg K_2, \dots, \neg K_n, L_1, L_2, \dots, L_n\}$$

The majority of real world problems are not naturally expressed in this form, general first-order logic being much more accessible to mankind's way of thinking.

Clearly there is a need for efficient conversions from general first-order logic to clause form. The way this is typically done by hand is well documented [Pau97], but for many problems this does not produce the smallest or most easily proven set of clauses.

This dissertation discusses the implementation and performance of this conventional method and also of an additional step in the conversion: selective renaming as conceived by Boy de la Tour [Boy90] and improved by Nonnengart, Rock and Weidenbach [NRW98]. This method takes advantage of a possible reduction in the number of clauses produced by replacing a subformula with some function which is defined by a definition function. For example, the function

$$\phi_1 \vee \phi_2$$

would be expected to generate  $mn$  clauses, where  $\phi_1$  generates  $m$  clauses and  $\phi_2$  generates  $n$  clauses. A renaming of  $\phi_1$  would be:

$$\underbrace{(P(x) \vee \phi_2)}_{\text{renamed formula}} \quad \wedge \quad \underbrace{\forall x(P(x) \Rightarrow \phi_1)}_{\text{definition of } P(x)}$$

It can be seen that the number of clauses generated by the renamed formula will be just  $m$  and the number of clauses generated by the definition will be

just  $n$ , giving a total of  $m + n$  clauses. Much of this project concentrates on selecting subformulae such that fewer clauses are indeed generated. As this would involve considering every subformula, a simple approach would take time exponential in the size of the input; we concentrate on a more involved method which reduces this to linear time.

The software discussed in this dissertation, *Clausify*, is a preprocessor designed to take input in general first order logic and produce an output suitable for the *SETHEO* theorem prover. *Clausify* offers a choice of the conventional or renaming approaches to clause form conversion, and was found to perform both conversions sufficiently fast that, for the test problems, the reduction in time spent in the theorem prover due to a smaller clause form was greater than the increase in time spent in the preprocessor due to the extra step of renaming.

# Chapter 2

## Preparation

*Definitions and explanations required in implementing the clause form conversion and renaming; how input to Clausify is formed, and how this form was decided upon. A detailed discussion of renaming.*

---

### 2.1 Atoms and Free Variables

Throughout this document the term “atom” will be used to refer to both single variables (such as  $A$ ,  $Tall$ , etc.) and functions of variables (such as  $f(A, B, C)$ ,  $sum(One, Two)$ ) — any formula which does not include any operators. The “free variables” in an atom are either the atom itself in the case of a single variable, or the function parameters in the case of a function (the function identifier itself is a constant). This is more formally set out below.

#### Definition 2.1.1

The set of Free Variables,  $fv(\phi)$ , of an expression,  $\phi$ , in first order logic is defined by induction on its structure.

$$\begin{aligned} \text{atom}(a) \Rightarrow (fv(a) &= \{a\}) \\ \text{atom}(a_1 \dots a_n) \Rightarrow (fv(f(a_1 \dots a_n)) &= \{a_1 \dots a_n\}) \\ fv(\neg\phi) &= fv(\phi) \\ fv(\phi_1 \vee \phi_2) = fv(\phi_1 \wedge \phi_2) &= fv(\phi_1) \cup fv(\phi_2) \\ fv(\forall a.\phi) = fv(\exists a.\phi) &= fv(\phi) \setminus \{a\} \end{aligned}$$

### 2.2 Clausify Language

Software dealing with statements in logic requires some language for inputting the formulæ. With no prior knowledge of existing languages for this purpose, the following specifications were drawn up:

- Must be easy to parse in Prolog:
  - All operators and quantifiers should take the form of functions or Prolog two-place operators

- Multiple quantified variables under one quantifier (e.g.  $\forall a_1, a_2, \dots, a_n. \phi$ ) should be avoided — there is no easy way of encoding this in a Prolog-friendly form
- Identifiers should be lower case: Prolog reserves upper case to indicate variables. In this situation, it is not useful to have Prolog able to unify identifiers with other identifiers or constants, and may cause unforeseen problems.
- Must be easily human readable:
  - Every symbol must have an intuitive meaning
  - Scoping should behave as expected but be controllable

Table 2.1 summarises the equivalent expressions in first order logic and the language of *Clausify*. For most of the operators, it is sufficient to declare (or redeclare) them in Prolog, which then allows pattern matching on formulæ in an intuitive way.

| Logic                           | <i>Clausify</i>         | Precedence | declaration                           |
|---------------------------------|-------------------------|------------|---------------------------------------|
| $\neg \phi$                     | $\sim p$                | 1          | <code>:-op(1, fy, ~).</code>          |
| $\phi_1 \wedge \phi_2$          | $p1 * p2$               | 3          | <code>:-op(3, yfx, *).</code>         |
| $\phi_1 \vee \phi_2$            | $p1 + p2$               | 4          | <code>:-op(4, yfx, +).</code>         |
| $\phi_1 \Rightarrow \phi_2$     | $p1 \rightarrow p2$     | 5          | <code>:-op(5, xfy, -&gt;).</code>     |
| $\phi_1 \Leftrightarrow \phi_2$ | $p1 \leftrightarrow p2$ | 6          | <code>:-op(6, xfy, &lt;-&gt;).</code> |
| $\forall A. \phi$               | $@a : p$                | [1]        | [1]                                   |
| $\exists A. \phi$               | $\#a : p$               | [1]        | [1]                                   |
| $f(A, B)$                       | $f(a, b)$               |            |                                       |
| $\forall a_1, a_2, a_3. \phi$   | $@a1 : @a2 : @a3 : p$   |            |                                       |

[1] See text

Table 2.1: First order logic to *Clausify* language mappings

### 2.2.1 Handling Quantifiers

The treatment of quantifiers demands a more detailed explanation. Disregarding multiple variables under one quantifier, it is clear that the conventional symbols ( $\forall, \exists$ ) are *two-place prefix operators*. That is, the quantified variable as the first parameter is delimited by a dot or bracketed with the quantifier, and followed by the quantified formula. This works in practice because the first parameter cannot contain anything other than an atomic variable. The closest analogue to this in Prolog is to define the two place operator `:` as

`:-op(2, xfy, :).`

On the left hand side of this operator, a quantifier always appears, and on the right hand side, the quantified formula. `xfy` is the Prolog specification that the operator is right associative: it cannot appear on its own left hand side. Thus multiple quantifiers behave as expected: `\#a : \#b : \#c : p` is equivalent to `\#a : (\#b : (\#c : p))`.

## 2.3 Definitions

There are several concepts used by Nonnengart, Rock and Weidenback [NRW98] which are outside the scope of the Part IB logic course [Pau97]. They are reviewed here. A mathematical representation of the position of a subformula within a formula enables us to reason about the properties that subformula has in context. The polarity of a subformula gives us information about where negations will occur after implications have been removed and negations have been pushed in.

### Definition 2.3.1

A position is a list of natural numbers indicating the location of a subformula within a formula. It is equivalent to an ordered list of branch numbers indicating the path taken to descend a parse tree to the subformula.

A list is written  $[i_1, i_2, \dots, i_n]$ . For such a list,  $L, [i_1 | [i_2, \dots, i_n]] = L$

The set  $\text{pos}(\phi)$  of positions of a given formula  $\phi$  is defined as follows:

- The empty list  $\emptyset \in \text{pos}(\phi)$
- If  $\phi = \phi_1 \circ \dots \circ \phi_n$  where  $\circ$  denotes any first order operator, and  $0 \leq i \leq n$ , then

$$\forall i, p. (p \in \text{pos}(\phi_i) \Rightarrow [i | p] \in \text{pos}(\phi))$$

The vertical bar is used to refer to a subformula:

$$\phi|_{\emptyset} = \phi$$

$$\phi|_{[i | p]} = \phi_i|_p$$

where  $\phi = \phi_1 \circ \dots \circ \phi_n$  as above.

For example:

$$\begin{aligned} \text{pos}(a \vee (b \wedge c)) &= \{\emptyset, [1], [2], [2, 1], [2, 2]\} \\ (a \vee (b \wedge c))|_{\emptyset} &= a \vee (b \wedge c) \\ (a \vee (b \wedge c))|_{[1]} &= a|_{\emptyset} = a \\ (a \vee (b \wedge c))|_{[2]} &= (b \wedge c)|_{\emptyset} = b \wedge c \\ (a \vee (b \wedge c))|_{[2, 1]} &= (b \wedge c)|_{[1]} = b|_{\emptyset} = b \\ (a \vee (b \wedge c))|_{[2, 2]} &= (b \wedge c)|_{[2]} = c|_{\emptyset} = c \end{aligned}$$

### Definition 2.3.2

The polarity of a subformula at position  $p$  in formula  $\phi$  (i.e.,  $\phi|_p$ ) is denoted by  $\text{Pol}(\phi, p)$ .

This definition makes use of the last element of a list which, for list  $L = [i_1, i_2, \dots, i_n]$  is denoted  $[[i_1, i_2, \dots, i_{n-1}] | i_n]$ .

$\text{Pol}(\phi, p)$  can now be defined thus:

- $\text{Pol}(\phi, \emptyset) = 1$
- If the formula  $\phi|_p$  is a quantified formula, or is a conjunction, disjunction, or an implication with  $i = 2$ , then  $\text{Pol}(\phi, [p | i]) = \text{Pol}(\phi, p)$

- If the formula  $\phi|_p$  is negation or an implication with  $i = 1$ , then  $\text{Pol}(\phi, [p|i]) = -\text{Pol}(\phi, p)$
- If the formula  $\phi|_p$  is an equivalence then  $\text{Pol}(\phi, [p|i]) = 0$

## 2.4 Renaming

A *renaming* of a subformula is the act of moving it to a separate clause by replacing the original subformula with a function which is defined in a new clause.

For example:

$$\begin{aligned} & \forall a. ( \underbrace{c \vee \neg d}_{f(c, d)} \Rightarrow (\neg c \wedge d) ) \\ & (\forall a. ( \underbrace{f(c, d)}_{\text{Definition}} \Rightarrow (\neg c \wedge d) )) \quad \wedge \quad \underbrace{(\forall c, d. (c \vee \neg d \Rightarrow f(a)))}_{\text{Definition}} \end{aligned}$$

The advantage of doing this is that the renamed subformula cannot generate a potentially large number of clauses under the distributive rules discussed in section 3.1.3. The removal of a particular subformula can reduce the number of clauses dramatically.

### Definition 2.4.1

A renaming of a subformula at position  $\pi$  in formula  $\phi$  (i.e.,  $\phi|_\pi$ ) is given by

$$\phi[\pi/R(\bar{x})] \wedge \text{Def}_\pi^\phi \tag{2.1}$$

Where:

- $\phi[\pi/f]$  denotes the replacement of the subformula at position  $\pi$  in  $\phi$  by the formula  $f$ .
- $\{\bar{x}\} = \text{fv}(\phi|_\pi)$ , the free variables in the subformula.
- $R$  is a predicate not otherwise referenced in  $\phi$ .
- $\text{Def}_\pi^\phi$  is a definition of the new predicate  $R$ :

$$\text{Def}_\pi^\phi = \forall \bar{x} \begin{cases} R(\bar{x}) \Rightarrow \phi|_\pi & \text{if } \text{Pol}(\phi, \pi) = 1 \\ \phi|_\pi \Rightarrow R(\bar{x}) & \text{if } \text{Pol}(\phi, \pi) = -1 \\ R(\bar{x}) \Leftrightarrow \phi|_\pi & \text{if } \text{Pol}(\phi, \pi) = 0 \end{cases} \tag{2.2}$$

The interesting problem is in deciding *which* subformulae to rename: it is possible that, due to the introduction of the implication in the definition, the number of clauses may be increased by a badly chosen renaming. A number of different schemes for selecting formulae have been suggested, but the one developed by Boy de la Tour [Boy90] is discussed here.

| $\phi$                          | $p(\phi)$   | $\bar{p}(\phi)$                                       |
|---------------------------------|---|---|
| $\neg\phi_1$                    | $\bar{p}(\phi_1)$                                     | $p(\phi_1)$   |
| $\phi_1 \wedge \phi_2$          | $p(\phi_1) + p(\phi_2)$                               | $\bar{p}(\phi_1)\bar{p}(\phi_2)$                      |
| $\phi_1 \vee \phi_2$            | $p(\phi_1)p(\phi_2)$                                  | $\bar{p}(\phi_1) + \bar{p}(\phi_2)$                   |
| $\phi_1 \Rightarrow \phi_2$     | $\bar{p}(\phi_1)p(\phi_2)$                            | $p(\phi_1) + \bar{p}(\phi_2)$                         |
| $\phi_1 \Leftrightarrow \phi_2$ | $p(\phi_1)\bar{p}(\phi_2) + \bar{p}(\phi_1)p(\phi_2)$ | $p(\phi_1)p(\phi_2) + \bar{p}(\phi_1)\bar{p}(\phi_2)$ |
| $\forall x.\phi_1$              | $p(\phi_1)$   | $\bar{p}(\phi_1)$                                     |
| $\exists x.\phi_1$              | $p(\phi_1)$   | $\bar{p}(\phi_1)$                                     |

Table 2.2:  $p(\phi)$ : the number of clauses produced

### 2.4.1 Selecting a Subformula

Clearly we only want to rename a subformula when the resulting number of clauses will be reduced. In table 2.2 recursively define the function  $p(\phi)$  to be the number of clauses produced by a conventional conversion to clause form.  $\bar{p}(\phi)$  is written for  $p(\neg\phi)$ .

A brief discussion of the origin of the results in the table is in order.

$\phi = \neg\phi_1$ : Trivial from the definition of  $\bar{p}(\phi)$ :  
 $p(\neg\phi) = \bar{p}(\phi)$  and  $\bar{p}(\neg\phi) = p(\neg\neg\phi) = p(\phi)$

$\phi = \phi_1 \wedge \phi_2$ :

$\phi = \phi_1 \vee \phi_2$ : From the distributive laws of disjunctions over conjunctions,  $\phi_1 \vee \phi_2$  will be expanded to the product of the disjunctions of each conjunct in  $\phi_1$  with each conjunct in  $\phi_2$ , so the total number of clauses produced will be the sum of the number produced by each term in the product. There are  $p(\phi_1)p(\phi_2)$  terms, so  $p(\phi_1 \vee \phi_2) = p(\phi_1)p(\phi_2)$ .  
 As above,  $\bar{p}(\phi_1 \vee \phi_2) = p(\neg(\phi_1 \vee \phi_2)) = p(\neg\phi_1 \wedge \neg\phi_2)$

$\phi = \phi_1 \Rightarrow \phi_2$ : From equation 3.1,  $p(\phi_1 \Rightarrow \phi_2) = p(\neg\phi_1 \vee \phi_2) = p(\neg\phi_1)p(\phi_2) = \bar{p}(\phi_1)p(\phi_2)$ .  
 Similarly,  $\bar{p}(\phi_1 \Rightarrow \phi_2) = p(\neg(\neg\phi_1 \vee \phi_2)) = p(\phi_1 \wedge \neg\phi_2) = p(\phi_1) + p(\neg\phi_2) = p(\phi_1) + \bar{p}(\phi_2)$

$\phi = \phi_1 \Leftrightarrow \phi_2$ : As  $p(\phi)$  is used only when the polarity of the subformula is 1, the expansion in equation 3.2 is used:  $p(\phi_1 \Leftrightarrow \phi_2) = p((\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)) = p(\phi_1 \Rightarrow \phi_2) + p(\phi_2 \Rightarrow \phi_1) = p(\phi_1)\bar{p}(\phi_2) + \bar{p}(\phi_1)p(\phi_2)$ .  
 Similarly,  $\bar{p}(\phi)$  is used when the polarity of the subformula is -1, so the expansion in equation 3.4 is used:  $\bar{p}(\phi_1 \Leftrightarrow \phi_2) = p(\neg(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \neg\phi_2)) = p((\neg\phi_1 \vee \neg\phi_2) \wedge (\phi_1 \vee \phi_2)) = p(\phi_1)p(\phi_2) + \bar{p}(\phi_1)\bar{p}(\phi_2)$

**Quantifiers:** No clauses are produced by  $\forall$  or  $\exists$ , so it is sufficient for  $p(\exists x.\phi) = p(\phi)$  and  $p(\forall x.\phi) = p(\phi)$ .

**Atoms:** Are one clause on their own, whether negated or not.

Note that while Boy de la Tour defines  $p(\phi)$  for  $n$ -place conjunctions and disjunctions, we have restricted these to being two-place operators, in accordance with the *Clausify* language definition in section 2.2, thus simplifying the definition a little.

| $\phi_\tau$                     | $a_{[\tau 1]}^\phi$   | $b_{[\tau 1]}^\phi$   |
|---------------------------------|---|---|
| $\neg\phi_1$                    | $b_\tau^\phi$   | $a_\tau^\phi$   |
| $\phi_1 \vee \phi_2$            | $a_\tau^\phi p(\phi_2)$   | $b_\tau^\phi$   |
| $\phi_1 \wedge \phi_2$          | $a_\tau^\phi$   | $b_\tau^\phi \bar{p}(\phi_2)$                                       |
| $\phi_1 \Rightarrow \phi_2$     | $b_\tau^\phi$   | $a_\tau^\phi p(\phi_2)$   |
| $\phi_1 \Rightarrow \phi_2[1]$  | $a_\tau^\phi \bar{p}(\phi_2)$                                       | $b_\tau^\phi$   |
| $\phi_1 \Leftrightarrow \phi_2$ | $a_\tau^\phi = a_\tau^\phi \bar{p}(\phi_2) + b_\tau^\phi p(\phi_2)$ | $b_\tau^\phi = a_\tau^\phi p(\phi_2) + b_\tau^\phi \bar{p}(\phi_2)$ |
| $\forall x.\phi_1$              | $a_\tau^\phi$   | $b_\tau^\phi$   |
| $\exists x.\phi_1$              | $a_\tau^\phi$   | $b_\tau^\phi$   |
| $\phi$                          | 1   | 0   |

[1] For this case, the columns refer to  $a_{[\tau|1]}^\phi$  and  $b_{[\tau|1]}^\phi$

Table 2.3:  $a_\pi^\phi$  and  $b_\pi^\phi$ : The influence of subformulae

It can now be seen that it is possible to check the desirability of a renaming subformula  $\phi|_\pi$  with the inequality<sup>1</sup>

$$p(\phi) \geq p(\phi[\pi/R(\bar{x})]) + p(Def_\pi^\phi) \quad (2.3)$$

The time taken to compute  $p(\phi)$  is linear in the size of  $\phi$ . To determine which subformulae to rename,  $p(\phi|_\pi)$  must be computed for all  $\pi \in \text{pos}(\phi)$ . As  $|\text{pos}(\phi)|$  is exponential in the size of the  $\phi$ , the time taken to determine the subformulae to rename is also exponential in the size of the  $\phi$ .

Nonnengart, Rock and Weidenbach [NRW98] solve this by computing coefficients  $a_\pi^\phi$  and  $b_\pi^\phi$  which correspond to the number of times that  $\phi|_\pi$  and  $\neg\phi|_\pi$  are repeated in the standard conversion to clause form. Subformulae which will not influence the number of clauses produced by the subformula at position  $\pi$  are not considered, since they will remain constant on both sides of the inequality.

Thus the value  $a_\pi^\phi p(\phi|_\pi) + b_\pi^\phi \bar{p}(\phi|_\pi)$  corresponds to the number of clauses including parts of  $\phi|_\pi$ . Since the clauses uninfluenced by the renaming will remain the same, and the replacement formula is atomic (so the coefficients are both 1) it is sufficient to reduce the inequality to

$$a_\pi^\phi p(\phi|_\pi) + b_\pi^\phi \bar{p}(\phi|_\pi) \geq a_\pi^\phi + b_\pi^\phi + p(\phi|_\pi) + \bar{p}(\phi|_\pi) \quad (2.4)$$

The values of  $a_\pi^\phi$  and  $b_\pi^\phi$  in table 2.3 are determined recursively by writing the number of clauses produced by  $\phi_\pi$  in terms of both the coefficients of  $\phi|_\pi$  and of the parent formula, as follows ( $\pi = [\tau|1]$  unless otherwise stated):

$$\phi|_\tau = \neg\phi_1: \quad a_\tau^\phi p(\phi|_\tau) + b_\tau^\phi \bar{p}(\phi|_\tau) = a_\tau^\phi p(\neg\phi_1) + b_\tau^\phi \bar{p}(\neg\phi_1) = a_\tau^\phi \bar{p}(\phi_1) + b_\tau^\phi p(\phi_1),$$

giving  $a_\tau^\phi = b_\tau^\phi$  and  $b_\tau^\phi = a_\tau^\phi$ .

$\phi|_\tau = \phi_1 \vee \phi_2$ :  $\phi_2$  will contribute to the number of clauses produced by  $\phi_1$ ; under negation,  $\neg\phi|_\tau = \neg\phi_1 \wedge \neg\phi_2$ , so  $\neg\phi_2$  will not contribute to the number

<sup>1</sup>While the  $\geq$  relation may seem excessive — it is pointless to perform a time-consuming substitution if the number of clauses will not be reduced — it turns out later that the inequalities are greatly simplified by the this point.

of clauses produced.

Thus  $a_\pi^\phi p(\phi|_\pi) + b_\pi^\phi \bar{p}(\phi|_\pi) = a_\tau^\phi p(\phi_1 \vee \phi_2) + b_\tau^\phi \bar{p}(\phi_2) = a_\tau^\phi p(\phi_1)p(\phi_2) + b_\tau^\phi \bar{p}(\phi_2)$ , giving  $a_\pi^\phi = a_\tau^\phi p(\phi_2)$  and  $b_\pi^\phi = b_\tau^\phi$ .

$\phi|_\tau = \phi_1 \wedge \phi_2$ : By comparison with above,  $\phi_2$  will not contribute to the number of clauses produced by  $\phi_1$ , but under negation,  $\neg\phi|_\tau = \neg\phi_1 \vee \neg\phi_2$ , so  $\neg\phi_2$  will contribute to the number of clauses produced.

Thus  $a_\pi^\phi p(\phi|_\pi) + b_\pi^\phi \bar{p}(\phi|_\pi) = a_\tau^\phi p(\phi_2) + b_\tau^\phi \bar{p}(\phi_1 \vee \phi_2) = a_\tau^\phi p(\phi_2) + b_\tau^\phi \bar{p}(\phi_1)\bar{p}(\phi_2)$  giving  $a_\pi^\phi = a_\tau^\phi$  and  $b_\pi^\phi = b_\tau^\phi \bar{p}(\phi_2)$ .

$\phi|_\tau = \phi_1 \Rightarrow \phi_2$ : By considering  $\phi' = \phi[\tau/(\neg\phi_1 \vee \phi_2)]$  (notice that  $\phi|_\pi = \phi'|_{[\pi|1]}$ ), we can see that  $a_{[\pi|1]}^{\phi'} = b_\pi^{\phi'}$  from the rule for negation, and  $b_\pi^{\phi'} = b_\tau^{\phi'} = b_\tau^\phi$  from the rule for disjunctions, thus  $a_\pi^\phi = b_\tau^\phi$ .

Similarly,  $b_{[\pi|1]}^{\phi'} = a_\pi^{\phi'}$  from the negation rule, and  $a_\pi^{\phi'} = a_\tau^{\phi'} p(\phi_2) = a_\tau^\phi p(\phi_2)$  from the disjunction rule, thus  $b_\pi^\phi = a_\tau^\phi p(\phi_2)$ .

$\phi|_\tau = \phi_1 \Rightarrow \phi_2$ : [In the case  $\pi = [\tau|2]$ ] Again, by considering the substitution  $\phi' = \phi[\tau/(\neg\phi_1 \vee \phi_2)]$ , we can see that  $\phi_2$  is an argument of a disjunction, so by comparison with the disjunction rule above, and by using the commutivity of disjunctions,  $a_\pi^\phi = a_\tau^\phi p(\phi_1)$  and  $b_\pi^\phi = b_\tau^\phi$ .

$\phi|_\tau = \phi_1 \Leftrightarrow \phi_2$ : Consider  $\phi' = \phi[\tau/((\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1))]$  (from equation 3.2) and  $\phi'' = \phi[\tau/((\phi_1 \wedge \phi_2) \vee (\neg\phi_2 \wedge \neg\phi_1))]$  (from equation 3.4). Since  $\phi_1$  is duplicated, the clauses generated by the two instances are added together, using  $\phi'$  for  $a_\pi^\phi$  and  $\phi''$  for  $b_\pi^\phi$ :

$$\begin{aligned} a_\pi^\phi p(\phi|_\pi) + b_\pi^\phi \bar{p}(\phi|_\pi) &= \\ & a_{[\tau|1|1]}^{\phi'} p(\phi'|_{[\tau|1|1]}) + b_{[\tau|1|1]}^{\phi''} \bar{p}(\phi''|_{[\tau|1|1]}) + a_{[\tau|2|2]}^{\phi'} p(\phi'|_{[\tau|2|2]}) + b_{[\tau|2|2]}^{\phi''} \bar{p}(\phi''|_{[\tau|2|2]}) \\ &= a_{[\tau|1]}^{\phi'} p(\phi'|_{[\tau|1]}) + b_{[\tau|1]}^{\phi''} \bar{p}(\phi''|_{[\tau|1]}) + a_{[\tau|2]}^{\phi'} p(\phi'|_{[\tau|2]}) + b_{[\tau|2]}^{\phi''} \bar{p}(\phi''|_{[\tau|2]}) \\ &= a_{[\tau|1]}^{\phi'} p(\phi_1 \Rightarrow \phi_2) + b_{[\tau|1]}^{\phi''} \bar{p}(\phi_1 \wedge \phi_2) + a_{[\tau|2]}^{\phi'} p(\phi_2 \Rightarrow \phi_1) + b_{[\tau|2]}^{\phi''} \bar{p}(\neg\phi_2 \wedge \neg\phi_1) \\ &= a_\tau^{\phi'} \bar{p}(\phi_1)p(\phi_2) + b_\tau^{\phi''} p(\phi_1)p(\phi_2) + a_\tau^{\phi'} \bar{p}(\phi_2)p(\phi_1) + b_\tau^{\phi''} \bar{p}(\phi_1)\bar{p}(\phi_2) \\ &= a_\tau^\phi \bar{p}(\phi_1)p(\phi_2) + b_\tau^\phi p(\phi_1)p(\phi_2) + a_\tau^\phi \bar{p}(\phi_2)p(\phi_1) + b_\tau^\phi \bar{p}(\phi_1)\bar{p}(\phi_2) \end{aligned}$$

So we end up with  $a_\pi^\phi = a_\tau^\phi \bar{p}(\phi_2) + b_\tau^\phi p(\phi_2)$  and  $b_\pi^\phi = a_\tau^\phi p(\phi_2) + b_\tau^\phi \bar{p}(\phi_2)$

**Quantifiers:** As for  $p(\phi)$ , quantifiers can be ignored.

**Whole formula:** Because the recursion of  $a_\pi^\phi$  and  $b_\pi^\phi$  is *ascending* in the structure of the formula, the base case is the entire formula. Clearly, this is only going to appear once, unnegated, thus  $a_\emptyset^\phi = 1$  and  $b_\emptyset^\phi = 0$ .

From this last point, it can be deduced that in the case that  $\text{Pol}(\phi, \pi)$  is +1,  $b_\pi^\phi = 0$ , and similarly when  $\text{Pol}(\phi, \pi)$  is -1,  $a_\pi^\phi = 0$ . This gives three cases from the one in equation 2.4:

$$a_\pi^\phi p(\phi|\pi) \geq a_\pi^\phi + p(\phi|\pi) \quad \text{if } \text{Pol}(\phi, \pi) = 1 \quad (2.5)$$

$$b_\pi^\phi \bar{p}(\phi|\pi) \geq b_\pi^\phi + \bar{p}(\phi|\pi) \quad \text{if } \text{Pol}(\phi, \pi) = -1 \quad (2.6)$$

$$a_\pi^\phi p(\phi|\pi) + b_\pi^\phi \bar{p}(\phi|\pi) \geq a_\pi^\phi + b_\pi^\phi + p(\phi|\pi) + \bar{p}(\phi|\pi) \quad \text{if } \text{Pol}(\phi, \pi) = 0 \quad (2.7)$$

Taking the first case, by rewriting as  $a_\pi^\phi p(\phi|\pi) + 1 \geq a_\pi^\phi + p(\phi|\pi) + 1$ , it may be rearranged thus:

$$\begin{aligned} a_\pi^\phi p(\phi|\pi) + 1 \geq a_\pi^\phi + p(\phi|\pi) + 1 &\Rightarrow a_\pi^\phi(p(\phi|\pi) - 1) + 1 \geq p(\phi|\pi) + 1 \\ &\Rightarrow a_\pi^\phi(p(\phi|\pi) - 1) - p(\phi|\pi) + 1 \geq 1 \\ &\Rightarrow (a_\pi^\phi - 1)(p(\phi|\pi) - 1) \geq 1 \end{aligned}$$

By observation, since  $a_\pi^\phi$  and  $p(\phi|\pi)$  cannot less than or equal to zero, they must both be strictly greater than 1 for the condition to hold. That is, it is sufficient to test

$$(a_\pi^\phi > 1) \wedge (p(\phi|\pi) > 1) \quad (2.8)$$

Similarly, the second case may be rearranged to  $(b_\pi^\phi - 1)(\bar{p}(\phi|\pi) - 1) \geq 1$ , which by a similar argument leads to the condition

$$(b_\pi^\phi > 1) \wedge (\bar{p}(\phi|\pi) > 1) \quad (2.9)$$

Finally, the third expression may be rearranged thus

$$\begin{aligned} a_\pi^\phi p(\phi|\pi) + b_\pi^\phi \bar{p}(\phi|\pi) &\geq a_\pi^\phi + b_\pi^\phi + p(\phi|\pi) + \bar{p}(\phi|\pi) \\ &\Rightarrow a_\pi^\phi p(\phi|\pi) + b_\pi^\phi \bar{p}(\phi|\pi) + 2 \geq a_\pi^\phi + b_\pi^\phi + p(\phi|\pi) + \bar{p}(\phi|\pi) + 2 \\ &\Rightarrow a_\pi^\phi(p(\phi|\pi) - 1) + b_\pi^\phi(\bar{p}(\phi|\pi) - 1) + 2 \geq p(\phi|\pi) + \bar{p}(\phi|\pi) + 2 \\ &\Rightarrow a_\pi^\phi(p(\phi|\pi) - 1) + b_\pi^\phi(\bar{p}(\phi|\pi) - 1) - p(\phi|\pi) + 1 - \bar{p}(\phi|\pi) + 1 \geq 2 \\ &\Rightarrow (a_\pi^\phi - 1)(p(\phi|\pi) - 1) + (b_\pi^\phi - 1)(\bar{p}(\phi|\pi) - 1) \geq 2 \end{aligned}$$

As above, it is sufficient to test for

$$\begin{aligned} &(a_\pi^\phi > 1) \wedge (p(\phi|\pi) > 1) \wedge (b_\pi^\phi > 1) \wedge (\bar{p}(\phi|\pi) > 1)) \\ &\vee (a_\pi^\phi > 1) \wedge (p(\phi|\pi) > 2) \\ &\vee (a_\pi^\phi > 2) \wedge (p(\phi|\pi) > 1) \\ &\vee (b_\pi^\phi > 1) \wedge (\bar{p}(\phi|\pi) > 2) \\ &\vee (b_\pi^\phi > 2) \wedge (\bar{p}(\phi|\pi) > 1) \end{aligned} \quad (2.10)$$

# Chapter 3

## Implementation

*Turning the theory into efficient Prolog; overcoming the difficulties inherent in choosing formulæ to rename*

---

The implementation of *Clausify* breaks down into two sections: the conventional approach to the conversion to clause form and Nonnengart, Rock and Weidenbach’s renaming approach.

### 3.1 Conventional Approach

We first look at the conventional approach to conversion to clause form. This breaks down into six steps:

1. Removing Implications
2. Conversion to Negation Normal Form
3. Conversion to Conjunctive Normal Form
4. (Anti)prenexing
5. Skolemization
6. Outputting Clauses

#### 3.1.1 **impls/2**: Eliminating Implications

All predicates that perform this type of transformation separate into three sections: performing substitutions where appropriate, recursing through the structure of the expression, and a simple termination case. In this case, the substitutions we are doing for **impls/2**<sup>1</sup> are:

$$\phi_1 \Rightarrow \phi_2 \longrightarrow \neg\phi_1 \vee \phi_2 \quad (3.1)$$

$$\phi_1 \Leftrightarrow \phi_2 \longrightarrow (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1) \quad (3.2)$$

$$(3.3)$$

---

<sup>1</sup>This notation will be used throughout this dissertation to refer to a Prolog procedure. The name is shown emboldened followed by the number of parameters that the procedure takes.

Thus only  $\Rightarrow$  and  $\Leftrightarrow$  need to be caught for substitution — for all other operators we recurse through the expression. The termination case is a single atom or function, but due to the cut (!) operator, this does not need to be explicitly tested for as the clause will only be tried if all others have failed, so the parameter is already atomic.

The first few predicates take the same form: input and output functions are the same, and the predicate is called again on all of the function parameters. *e.g.*

```
impls(A+B, C+D) :- !,
                    impls(A, C),
                    impls(B, D).

impls(Q:B, Q:C) :- !,
                    impls(B, C).
```

Similarly, for the predicates which perform the substitutions, we recursively call the predicate on the substituted functions.

```
impls(A->B, Z) :- !,
                    impls(~A+B, Z).

impls(A<->B, Z) :- !,
                    impls((A->B)*(B->A)).
```

Finally, the termination case is very simple:

```
impls(A, A).
```

These predicates as given correspond to the substitutions 3.1; they can, however be greatly improved. Let us look at the case of implications. The expected sequence of execution is as follows:

```
call impls(A->B, Z).
  call impls(~A+B, Z).
    call impls(~A, Z).
      call impls(A, Ai).
      call impls(B, Bi).
      return impls(~A, ~Ai).
    return impls(~A+B, ~Ai+Bi)
  return impls(A->B, ~Ai+Bi).
```

Since this sequence of execution is guaranteed to happen whenever an implication is encountered, there are clearly a number of redundant calls: we know that the first two calls will always be a conjunction and a negation. By including these in the definition we get:

```
impls(A->B, ~C+D) :- !,
                    impls(A, C),
                    impls(B, D).
```

With the resulting sequence of execution:

```
call impls(A->B, Z).
  call impls(A, Ai).
  call impls(B, Bi).
return impls(A->B, ~Ai+Bi).
```

A similar analysis could be applied to equivalences, however the complexity of the resulting expression would be such that the risk of introducing errors would be escalated, combined with a loss of maintainability. It was thus decided not to change this predicate.

More importantly, a look at a couple of real examples reveals an effective improvement that can be made to the equivalence substitution. Consider  $a \Leftrightarrow b$ . The conversion to clause form proceeds as follows:

$$\frac{(\neg a \vee b) \wedge (\neg b \vee a)}{(a \Rightarrow b) \wedge (b \Rightarrow a)} \\ a \Leftrightarrow b$$

Now consider  $\neg(a \Leftrightarrow b)$ :

$$\frac{(a \vee b) \wedge (\neg a \vee \neg b)}{(a \vee b) \wedge (a \vee \neg a) \wedge (\neg b \vee b) \wedge (\neg b \vee a)} \\ \frac{\neg((a \wedge \neg b) \vee (b \wedge \neg a))}{\neg((\neg a \vee b) \wedge (\neg b \vee a))} \\ \frac{\neg((a \Rightarrow b) \wedge (b \Rightarrow a))}{\neg(a \Leftrightarrow b)}$$

We can't expect the preprocessor to spot and remove redundant clauses in this way, but we can predict when it would be more efficient to use this latter expansion directly: when the polarity of the subformula is negative. Note that rather than replacing  $\neg(a \Leftrightarrow b)$  with  $(a \vee b) \wedge (\neg a \vee \neg b)$ , we are replacing simply  $a \Leftrightarrow b$ , the negation simply contributing to the polarity of the expression. Thus the replacement function becomes  $\neg((a \vee b) \wedge (\neg a \vee \neg b))$  which is rearranged to become

$$(a \wedge b) \vee (\neg a \wedge \neg b) \tag{3.4}$$

Maintaining a record of the polarity of the current subformula demands an extra parameter to the predicates. For most of the recursion predicates, this parameter is simply passed through. For  $\neg$  and  $\Rightarrow$ , it must be changed appropriately (see definition 2.3.2). The following includes an example of a recursion predicate, the others being modified similarly.

```
impls(~A, ~Z, P) :- !,
  Pneg is ~P,
impls(A, Z, Pneg).
```

$$\begin{aligned} \mathbf{impls}(A+B, C+D, P) :- \quad &!, \\ &\mathbf{impls}(A, C, P), \\ &\mathbf{impls}(B, D, P). \end{aligned}$$

$$\begin{aligned} \mathbf{impls}(A \rightarrow B, \sim C+D, P) :- \quad &!, \\ &P_{\text{neg}} \text{ is } \sim P, \\ &\mathbf{impls}(A, C, P_{\text{neg}}), \\ &\mathbf{impls}(B, D, P). \end{aligned}$$

For equivalences, a decision is now made on the basis of the polarity:

$$\begin{aligned} \mathbf{impls}(A \leftrightarrow B, Z, 1) :- \quad &!, \\ &\mathbf{impls}((A \rightarrow B) * (B \rightarrow A), Z, 1). \end{aligned}$$

$$\begin{aligned} \mathbf{impls}(A \leftrightarrow B, Z, -1) :- \quad &!, \\ &\mathbf{impls}((A * B) + (\sim A * \sim B), Z, -1). \end{aligned}$$

Note that in  $a \leftrightarrow b$ , the subformulae  $a$  and  $b$  have polarity 0 from the definition, but we pass 1 or  $-1$  because, of course, the subformula is no longer an equivalence, so the rule no longer applies.

### 3.1.2 negs/2: Pushing Negations In

Pushing negations in until they apply only to atoms or functions takes broadly the same three-phase form as implications. This time the substitutions are as follows:

$$\neg(\neg\phi) \longrightarrow \phi \tag{3.5}$$

$$\neg(\phi_1 \vee \phi_2) \longrightarrow \neg\phi_1 \wedge \neg\phi_2 \tag{3.6}$$

$$\neg(\phi_1 \wedge \phi_2) \longrightarrow \neg\phi_1 \vee \neg\phi_2 \tag{3.7}$$

$$\neg(\neg\forall\phi) \longrightarrow \exists\phi \tag{3.8}$$

$$\neg(\neg\exists\phi) \longrightarrow \forall\phi \tag{3.9}$$

The conversion of these rules to Prolog predicates follows the same outline as **impls/2**. The only interesting part concerns quantifiers. The  $:$  operator (see section 2.2.1) may be regarded as a simple recursion case which distributes the negation across its arguments. This leaves special cases for the quantifiers as listed above:

$$\begin{aligned} \mathbf{negs}(A:B, An:Bn) :- \quad &!, \\ &\mathbf{negs}(A, An), \\ &\mathbf{negs}(B, Bn). \end{aligned}$$

$$\begin{aligned} \mathbf{negs}(\sim(A:B), Z) :- \quad &!, \\ &\mathbf{negs}(\sim A: \sim B, Z). \end{aligned}$$

$$\mathbf{negs}(\sim(@A), \#A) :- \quad !.$$

$$\mathbf{negs}(\sim(\#A), @A) :- \quad !.$$

### 3.1.3 **cnf/2: Converting to Clause Normal Form**

Conversion to clause normal form is simply the application of the law for distributing disjunctions over conjunctions — forcing all conjunctions outside enclosing formulae, leaving only clauses. Thus substitution performed is:

$$(\phi_1 \wedge \phi_2) \vee \phi_3 \longrightarrow (\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3) \quad (3.10)$$

This is expanded into two cases, to handle the commutivity of the logical or operation.

**cnf**((A\*B)+C, Z) :- !,  
           **cnf**((A+C)\*(B+C), Z).

**cnf**(C+(A\*B), Z) :- !,  
           **cnf**((C+A)\*(C+B), Z).

While it seems appropriate to replace the complex recursive calls with the right hand sides of **cnf**(A\*B) as we did with **impls/2**, a closer analysis of the recursive cases reveals a hidden complexity.

The obvious coding of the recursive cases would be:

**cnf**(A+B, Ac+Bc) :- !,  
           **cnf**(A, Ac),  
           **cnf**(B, Bc).

The problem with this comes in the analysis of formulae such as  $((a \wedge b) \vee c) \vee d$ . The sequence of execution in this case is given in figure 3.1. The final result here is  $((a \vee c) \wedge (b \vee c)) \vee d$ , which is clearly not in CNF. Another recursive call to **cnf/2** is required in the recursion cases (A+B and A\*B) with the whole result as the argument. A decision must be made as to when to make this extra recursive call — it cannot be made every time, as in simple cases such as  $a \vee b$ , the predicate will loop forever.

A simple heuristic would be to check whether the two recursive calls **cnf**(A, Ac) and **cnf**(B, Bc) make any modification to their arguments; if not we can safely assume that there is no need to recheck the entire formula. The comparison and decision is performed by the predicate **compcnf/6**.

#### **compcnf/6: Shall I do it Again?**

The two pairs of variables to be compared are given as the first four arguments; the formula as it would have been returned is given as the fifth, and this is passed to another call of **cnf/2** if necessary, before returning in argument six.

The new recursive cases along thus become:

**cnf**(A+B, Z) :- !,  
           **cnf**(A, Ac),  
           **cnf**(B, Bc),  
           **compcnf**(A, Ac, B, Bc, Ac+Bc, Z).

```

call cnf((a*b)+c)+d, Ac+Bc).
  call cnf((a*b)+c, Z).
    call cnf((a+c)*(b+c), Ac*Bc).
      call cnf(a+c, Ac+Bc).
        call cnf(a, a).
        call cnf(b, b).
      return cnf(a+c, a+c).
    call cnf(b+c, Ac+Bc).
      call cnf(b, b).
      call cnf(c, c).
    return cnf(b+c, b+c).
  return cnf((a+c)*(b+c), (a+c)*(b+c)).
return cnf((a*b)+c, (a+c)*(b+c)).
call cnf(d, d).
return cnf((a*b)+c)+d, ((a+c)*(b+c))+d)

```

Figure 3.1: Execution sequence for  $((a \wedge b) \vee c) \vee d$ 

```

cnf(A*B, Z) :- !,
    cnf(A, Ac),
    cnf(B, Bc),
    compcnf(A, Ac, B, Bc, Ac*Bc, Z).

```

```

compcnf(A, A, B, B, Z, Z) :- !.

```

```

compcnf(_, _, _, _, X, Y) :- !,
    cnf(X, Y).

```

### 3.1.4 quants/2: Converting to Antiprenex Normal Form

Antiprenex form, that is, pushing quantifiers in until they apply to a minimum size of formula (either a term or a quantified term) is generally regarded [Egl94] as producing better clause forms than the (easier) Prenex form conversion.

The rules for the conversion are as follows:

$$\forall a.(\phi_1 \wedge \phi_2) \longrightarrow \forall a.\phi_1 \wedge \exists a.\phi_2 \quad (3.11)$$

$$\exists a.(\phi_1 \wedge \phi_2) \longrightarrow \exists a.\phi_1 \wedge \exists a.\phi_2 \quad (3.12)$$

$$\forall a.(\phi_1 \wedge \phi_2) \longrightarrow \forall a.\phi_1 \wedge \phi_2 \quad (3.13)$$

$$\forall a.(\phi_1 \vee \phi_2) \longrightarrow \forall a.\phi_1 \vee \phi_2 \quad (3.14)$$

$$\exists a.(\phi_1 \wedge \phi_2) \longrightarrow \exists a.\phi_1 \wedge \phi_2 \quad (3.15)$$

$$\exists a.(\phi_1 \vee \phi_2) \longrightarrow \exists a.\phi_1 \vee \phi_2 \quad (3.16)$$

Obviously the last four rules are only valid if  $a \notin FV(\phi_2)$ .

It was decided to ignore the final four rules: quantifying variables which are not free will make no difference to the final clause form representation, and

the superfluous quantifiers will be removed during Skolemization — it seems pointless to have to perform the same tests twice.

While quantifiers could be pushed through negations, this would involve changing  $\forall$  to  $\exists$  and *vice versa*; an exercise which is fairly pointless since this conversion was probably performed in the opposite direction **negs/2**.

The recursion predicates are trivial, as is distributing quantifiers as in rules 3.11 and 3.12. The interesting part of the code comes in the termination case: we must determine when we have reached an “atom, negated atom or quantified atom”. This is achieved through the predicate **qatomic/1**.

### 3.1.5 skolemize/2: Skolemization

Skolemization is the replacement of quantified atoms by functions thus:

$$\forall a_1. \forall a_2. \dots \forall a_n. \exists e_1. \forall b_1. \forall b_2. \dots \forall b_n. \exists e_2. \phi \dots \longrightarrow \phi[e_1/f_1(a_1, a_2, \dots, a_n)][e_2/f_2(a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n)] \dots \quad (3.17)$$

where  $\phi[a/b]$  is the expression  $\phi$  with all occurrences of  $a$  replaced with  $b$ .

The problem breaks down into two parts: the search for a quantified subformula (a trivial recursive procedure) and the handling of such a subformula. This latter part is handled by a new predicate **skolemize/3**. The extra variable is now used to accumulate the universally quantified variables until the first existential quantifier is found. A function must now be created with the arguments from the universal quantifiers. To ensure the uniqueness of the function identifier, we use **gensym/2** (see section A.5)

Once again, the `=..` operator is used, this time to construct a term from a list, where the head of the list is the function name, and the tail of the list is the accumulated parameters. All that remains is to substitute the new formula for the atom throughout the quantified subformula. The predicate **alphasub/4** performs this alpha substitution, and is discussed in the next section.

```
skolemize(#A:X, V, Z) :- !,
    gensym(A,A1),
    F=..[A1|V],
    alphasub(X, A, F, Y),
    skolemize(Y, V, Z).
```

### 3.1.6 alphasub/4: Alpha Substitution

**alphasub/4** takes the three-part form discussed above: recursion, work, termination. The recursive cases are simple. Leaving two other cases: for single variable atoms, for which the output variable is simply unified with the substituted variable, and for functions, which entail breaking down the parameters into a list and performing substitutions on the whole list:

```
alphasub(A, V1, V2, Aa) :- A=..[Af|Av],
    substitute(Av, V1, V2, Ava),
    Aa=..[Af|Ava].
```

The **substitute/4** predicate substitutes *V1* for *V2* in its first argument, resulting in its fourth argument. This is a task ideally suited to Prolog's list handling capabilities, and is presented here without further explanation.

```

substitute([], -, -, []) .
substitute([V1|T1], V1, V2, [V2|T2]) :- !, substitute(T1, V1, V2, T2).
substitute([H|T1], V1, V2, [H|T2]) :- !, substitute(T1, V1, V2, T2).

```

### 3.1.7 **claus/2: Is this one of mine?**

The final part of the preprocessor must be to produce a new line separated list of the clauses produced, ready for feeding to the theorem prover. This gives an ideal opportunity to exploit the backtracking properties of Prolog. The first step in the conversion is a predicate which succeeds only when its second argument is a constituent clause of its (conjunctive normal form) first argument. By backtracking repeatedly, the second argument is instantiated to all of the clauses in turn.

The simple cases to consider are where the expression and the clause are the same and either atomic (which is trivially a clause, and is determined by **atomic/2**) or a disjunction (which must be a clause because we know that the expression is in conjunctive normal form: all disjunctions must be inside a clause). This leaves the conjunctions which obviously separate the clauses; the clause being checked must either be on the left or the right of any conjunction found. Notice that neither of these cases are followed by the cut (!) operator: since any conjunction will be matched by both cases, this gives the opportunity for backtracking to check both and producing a complete set of clauses.

For completeness we include a case to ignore any quantifiers.

```

claus(_*A, Z) :- claus(A, Z).
claus(A*_ , Z) :- claus(A, Z).
claus(_:A, Z) :- !,
                claus(A, Z).

```

The termination case is simply the atomic variable, although because of the repeated backtracking, we cannot simply rely on the preceding clauses to eliminate all possible operators as in **impls/2**; instead, we make use of the **atomic/2** operator, which succeeds when its input is a single variable or a function.

### **prclauses/1 and prclause/1: Final Output**

All that remains is to find all of the clauses using the Prolog builtin **findall/3**, which takes a variable name to iterate over and the predicate to be evaluated, and returns a list of results. **prclause/1** is called on each member of the list to print the clause itself in comma separated form by recursively dividing the summation down into atoms.

To comply with *SETHEO*'s syntax and typing constraints, all variable names have to begin with a capital letter, so are preceded by the letter V on output. Also, as a variable cannot be used outside of a function, all such variables have function  $v()$  applied to them, which we define as succeeding only when its parameter is true.

## 3.2 Implementing Renaming

### 3.2.1 search/9: Gathering information

The main difficulty with the implementation of the renaming scheme as discussed in section 2.4 is that functions  $a_\pi^\phi$ ,  $b_\pi^\phi$ ,  $\text{Pol}(\phi_\pi)$  and  $p(\phi|\pi)$  are not only mutually recursive, but traverse formula  $\phi$  in opposite directions, that is, for  $p(\phi|_p)$ , the termination cases are the atoms, but for  $a_\pi^\phi$ ,  $b_\pi^\phi$  and  $\text{Pol}(\phi_\pi)$  the termination case is the entire formula. As the result of  $p_\pi^\phi$  may be required in the evaluation of  $a_\pi^\phi$  and  $b_\pi^\phi$  for any  $\pi$ , there is a potentially serious efficiency problem.

We solve this by making very heavy use of Prolog unification to defer the computation of each inequality until the entire formula has been traversed: starting with the whole formula (*i.e.*, the top of the parse tree) variables A, B, and Pol, corresponding to  $a_\pi^\phi$ ,  $b_\pi^\phi$ ,  $\text{Pol}(\phi_\pi)$  respectively, can be computed directly or in terms of P and Pbar, corresponding to  $p(\phi|_p)$  and  $\bar{p}(\phi|_p)$  respectively. The values of P and Pbar are unknown until we have recursed down to all atoms reachable from the current point, so the values passed down of A and B are unevaluated Prolog terms. Upon return from the recursive calls, P and Pbar can now be determined from the results of those calls.

In addition to these variables, the current value of  $\pi$  is constructed in reverse order (because it is easier to add terms to the front of a list in Prolog) in Pos, and a *candidate list* of possible candidates for renaming is maintained (through predicate **enlist/8**) of some of the information about the current position, to be used in computing the inequalities later. This list is ordered such that the elements appear in reverse order of the subformulæ visited. It will become apparent in section 3.2.4 how useful this is.

```
search(X*Y, P, Pbar, A, B, Pol, Pos, Old, New) :-
    enlist(X*Y, P, Pbar, A, B, Pol, Pos, Old, Old1),
    search(X, PX, PbarX, A, B*PbarY, Pol, [1|Pos], Old1, Old2),
    search(Y, PY, PbarY, A, B*PbarX, Pol, [2|Pos], Old2, New),
    P is PX+PY,
    Pbar is PbarX*PbarY.
```

```
search(X+Y, P, Pbar, A, B, Pol, Pos, Old, New) :-
    enlist(X+Y, P, Pbar, A, B, Pol, Pos, Old, Old1),
    search(X, PX, PbarX, A*PY, B, Pol, [1|Pos], Old1, Old2),
    search(Y, PY, PbarY, A*PX, B, Pol, [2|Pos], Old2, New),
    P is PX*PY,
    Pbar is PbarX+PbarY.
```

```

search( $X \rightarrow Y$ , P, Pbar, A, B, Pol, Pos, Old, New) :-
  enlist( $X \rightarrow Y$ , P, Pbar, A, B, Pol, Pos, Old, Old1),
  NPol is -Pol,
  search(X, PX, PbarX, B, A*PY, NPol, [1|Pos], Old1, Old2),
  search(Y, PY, PbarY, A*PX, B, Pol, [2|Pos], Old2, New),
  P is PbarX*PY,
  Pbar is PX+PbarY.

```

```

search( $X \leftrightarrow Y$ , P, Pbar, A, B, Pol, Pos, Old, New) :-
  enlist( $X \leftrightarrow Y$ , P, Pbar, A, B, Pol, Pos, Old, Old1),
  search(X, PX, PbarX, A*PbarY+B*PY, A*PY+B*PbarY, 0,
    [1|Pos], Old1, Old2),
  search(Y, PY, PbarY, A*PbarX+B*PX, A*PX+B*PbarX, 0,
    [2|Pos], Old2, New),
  P is PX*PbarY+PbarX*PY,
  Pbar is PX*PY+PbarX*PbarY.

```

For the case of quantifiers, we again see the advantages of using the `:` operator; note, however, that while we consider it to be a normal operator in the sense that the quantified formula as assigned position 2, we do not record the formula and quantifier as a candidate for renaming. Clearly, if the quantified formula is to be renamed, this would imply that the (pointless) renaming of the quantifier and formula was necessary. This is avoided by not adding subformulae with the top level operator `:` to the list of candidates for renaming.

```

search( $\_ : X$ , P, Pbar, A, B, Pol, Pos, Old, New) :-
  search(X, P, Pbar, A, B, Pol, [2|Pos], Old1, New).

```

### 3.2.2 **enlist/9: Keeping a Record**

Consider again the three test conditions in equations 2.8, 2.9, and 2.10. The first two are of the same form, so subformulae with polarity of +1 or -1 are added to the list as the term  $\mathbf{s}(\text{Pos}, \text{Pol}, A, P)$  or  $\mathbf{s}(\text{Pos}, \text{Pol}, B, \text{Pbar})$  ( $\mathbf{s}$  stands for 'simple' case).

```

enlist(X, P, Pbar, A, B, 1, Pos, Old, [s(Pos, 1, A, P)|Old]) .

```

```

enlist(X, P, Pbar, A, B, -1, Pos, Old, [s(Pos, 1, B, Pbar)|Old]) .

```

The third test case depends on all four variables, but not on the polarity of the formula, so subformulae with polarity 0 are added to the list as the term  $\mathbf{c}(\text{Pos}, A, B, P, \text{Pbar})$  ( $\mathbf{c}$  stands for 'complex' case).

```

enlist(X, P, Pbar, A, B, 0, Pos, Old, [c(Pos, A, B, P, Pbar)|Old]) .

```

### 3.2.3 **chklist/3**: Choosing Subformulæ

The next step is to work through the list produced by **enlist/9** removing the details about any subformulæ which are not to be renamed. As none of the details about the decision are required by the renaming algorithm except for  $\pi$  and  $\text{Pol}(\phi, \pi)$ , the output of **chklist/3** is a list of terms of the form  $\mathbf{r}(\text{Pr}, \text{Pol})$  where  $\text{Pr}$  is the list indicating the position of the subformula, reversed from the output from **search/9**, ready for use in finding the positions again in **rename/3**. The third parameter of **chklist/3** is maintained as the previous renamed position — the last position where a candidate for renaming was accepted; it is initialised to some non-list constant so that, until a renaming has been made, all list comparisons discussed below will fail.

The first series of tests eliminate two possible causes of excessive renaming: renaming both sides of an operator and nested renamings. The first of these is checked for by comparing the tail of the current position with the tail of the previous position, taking advantage of the reverse ordering of the position lists and the depth-first ordering of the candidate list to find positions which differ only in which side of the same operator they are on. Once one of these matches has been made, we know that because of the list ordering, no subsequent subformulæ will be descended from the stored ‘previous position’, so it is updated to the position which made the match, ensuring that no subformulæ from that side of the operator are renamed either.

```
chklist([s([H|Pos],-,-,)|T1],T2,[_|Pos]) :- !,
chklist(T1, T2, [H|Pos]).
```

```
chklist([c([H|Pos],-,-,)|T1], T2,[H|Pos]) :- !,
chklist(T1, T2, [H1|Pos]).
```

The second elimination, nested renamings, makes use of the **append/3** predicate, normally used to concatenate lists. By reversing the candidate list before the invocation of **chklist/3**, the test becomes one of whether the ‘previous list’ comes at the end of the ‘current list’, or whether something with the previous list appended is equal to the current list:

```
chklist([s(Pos, -, -,)|T1], T2, OldPos) :- append(-, OldPos, Pos),
!,
chklist(T1, T2, OldPos).
chklist([c(Pos, -, -,)|T1], T2, OldPos) :- append(-, OldPos, Pos),
!,
chklist(T1, T2, OldPos).
```

The rest of the tests simply take the form of comparisons on variables  $A$ ,  $B$ ,  $P$ , and  $Pbar$ ; since the recursion through the formula in **search/9** is complete, we know that all of the variables which make up terms  $A$  and  $B$  have been unified to numbers, and the comparison forces the arithmetic to be performed.

We only give two cases here as an example, but by reference to equation 2.10, there must be four more cases for the term **c/5**.

**chklist**([], []) .

**chklist**([s(P, Pol, A, B)|T1], [r(Pr, Pol)|T2], \_) :- A>1, B>1, !,  
**reverse**(P, Pr),  
**chklist**(T1, T2, P).

**chklist**([c(Pos, A, B, P, Q)|T1], [r(Pr, 0)|T2], \_) :- A>1, B>1, P>1, Q>1, !,  
**reverse**(Pos, Pr),  
**chklist**(T1, T2, Pos).

**chklist**([\_|T1], T2) :- **chklist**(T1, T2).

### 3.2.4 rename/3: Making the Changes

Now that we have a list of subformulae to rename, all that remains is to recurse through the formula making the appropriate replacements and gathering all of the definitions to append to the end of the formula. Here it becomes apparent how useful the depth-first, right-to-left ordering of the list is: a substitution may be made without affecting the positions of any subformulae later in the list. This is a crucial result, as it means that all substitutions can be made at once, without needing to calculate new positions for the renamed subformulae.

The renaming is a simple recursive function moving through the list and performing substitutions on the calls and appending the definitions (in the variable Aux) on the returns.

**rename**([], F, F) .

**rename**([r(Pos, Pol)|T], F, Fnew) :- **subs**(Pos, Pol, F, F1, Aux),  
**rename**(T, F1, F2),  
Fnew=F2\*Aux.

### 3.2.5 subs/4: Choosing the substitutions

Actually making the substitutions splits in several parts: finding the referenced subformula, generating the replacement function and definition and performing the substitution.

To find the subformula we recurse on the position list, extracting the right or left subformula from each operator, and recombining the result of the recursive call in the same place. This extracting and recombining is done with the Prolog =.. operator.

We have a special case for negation, since it is the only operator with only one parameter, so could not be easily handled by a general case. Note that we do not need to handle quantifiers explicitly because in section 3.2.1 we decided to treat : as a normal operator — so the only subformula referenced will be position 2, the quantified formula.

**subs**([1|T], Pol, ~F, ~Fnew, Aux) :- **subs**(T, Pol, F, Fnew, Aux).

**subs**([1|T], Pol, F, Fnew, Aux) :- F=..[Op, Left, Right],  
**subs**(T, Pol, Left, Leftnew, Aux),  
 Fnew=..[Op, Leftnew, Right].

**subs**([2|T], Pol, F, Fnew, Aux) :- F=..[Op, Left, Right],  
**subs**(T, Pol, Right, Rightnew, Aux),  
 Fnew=..[Op, Left, Rightnew].

Once the appropriate subformula has been found, we need a list of free variables in subformula. This is found with **fv/2** (a direct translation of inductive definition 2.1.1 into Prolog), and a new formula is constructed using the **=..** operator. Finally, the predicate **forall/3** is used to quantify the definition appropriately.

**subs**([], 1, F, Fnew, Aux) :- **fv**(F, Fv),  
**gensym**(ren, R),  
 Fnew=..[R|Fv],  
**forall**(Fv, Fnew->F, Aux).

**subs**([], -1, F, Fnew, Aux) :- **fv**(F, Fv),  
**gensym**(ren, R),  
 Fnew=..[R|Fv],  
**forall**(Fv, F->Fnew, Aux).

**subs**([], 0, F, Fnew, Aux) :- **fv**(F, Fv),  
**gensym**(ren, R),  
 Fnew=..[R|Fv],  
**forall**(Fv, F<->Fnew, Aux).

**forall/3** is very simple: it recurses through the free variable list and creates a quantifiers for each one, terminating with the definition function.

**forall**([], F, F) .

**forall**([H|T], F, @H:Hs) :- **forall**(T, F, Hs).



# Chapter 4

## Evaluation

*Running Clausify on some sample problems. How the conventional conversion compares to the text book answers. How the renaming method helps with reducing the number of clauses, and whether the fewer clauses result in less time spent in the theorem prover.*

---

The problems used to test *Clausify* are due to Pelletier [Pel86].

### 4.1 Sample Output

Taking as an example, Pelletier's first problem,  $(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$ , the output is as follows:

```
SICStus 3 #5: Tue Sep 2 11:02:14 BST 1997
{compiling /home/djs52/Project/sicstus-cl.pl...}
{/home/djs52/Project/sicstus-cl.pl compiled, 110 msec 11024 bytes}
{consulting /home/djs52/Project/clausify.pl...}
{/home/djs52/Project/clausify.pl consulted, 380 msec 34384 bytes}
{loading /group/clteach/sicstus/bin/library/charsio ql...}
{loaded /group/clteach/sicstus/bin/library/charsio ql in module charsio, 20 msec 15728 bytes}

(p+ ~q)*(p+p)*((~q+ ~q)*(~q+p))*(~p+q+(q+ ~p))

--clauses produced--
~v(Vp), v(Vq), v(Vq), ~v(Vp)
~v(Vq), v(Vp)
~v(Vq), ~v(Vq)
v(Vp), v(Vp)
v(Vp), ~v(Vq)

yes
inwasm V3.2.5 Copyright TU Munich (April 7, 1995)
command line: /home/djs52/setheo.solaris/inwasm -cons -foldup temp
temp.s generated in 0.02 seconds
Parsing
Preprocessing: purity
  Deleted: -
Preprocessing: orbranch reordering
Preprocessing: inserting tautology constraints
Preprocessing: inserting subsumption constraints
  Deleted: 5 1
Preprocessing: removing redundant constraints
  Deleted: 3
Preprocessing: fanning
Preprocessing: inserting reduction steps
Preprocessing: inserting symmetry constraints
  Deleted: 3.3
  Deleted: 4.3
Preprocessing: removing redundant constraints
  Deleted: 0
```

```

Preprocessing: subgoal reordering
Codegeneration.
wasm V3.2 Copyright TU Munich (March 25, 1994)
Command line: /home/djs52/setheo.solaris/wasm temp
Assembler optimization: 62 labels read, 14 labels output
temp.hex generated in 0.15 seconds
SAM V3.3 Copyright TU Munich (December 22, 1995)

Options : -dr -cons -dynsgreord 2 temp

using antilemma-constraints
using regularity-constraints
using tautology-constraints
using subsumption-constraints

Start proving...

-d: 2 time < 0.01 sec inferences = 7 fails = 4

***** SUCCESS *****

Number of inferences in proof      : 4
- E/R/F/L                          : 2/ 2/ 0/ 0
Intermediate free variables        : 3
Intermediate inferences            : 4
Intermediate open subgoals        : 2
Generated antilemmata             : 0
Number of unifications            : 7
- E/R/F/L                          : 5/ 2/ 0/ 0
Number of generated constraints    : 3
- anl/reg/ts                       : 0/ 0/ 3
Number of fails                   : 4
- depth bound                      : 4
Number of folding operations       : 3
- one level                        : 1
- root                             : 3
Instructions executed              : 53
Abstract machine time (seconds)    : < 0.01
Overall time (seconds)            : 0.04

```

The output from *Clausify* itself is almost at the very top: the formula after conversion to clause normal form followed by the five clauses fed to *SETHEO* for (successful) proving. Note that there are five clauses generated to Pelletier's four; there is one repeated clause ( $\{p, \neg q\}$ ). Also one clause has repeated elements ( $\{\neg p, q, q, \neg p\}$ ). It was never intended for *Clausify* to be able to detect and eliminate such clauses, and it can be seen that *SETHEO* eliminates them early on in its subsumption and symmetry constraints steps.

## 4.2 Performance in Conventional Mode

The number of clauses produced by *Clausify* in using the conventional conversion to clause form for several input formulæ is given in table 4.1. In all cases where *Clausify* generates more clauses than given by Pelletier, it can be shown that there are two or more equivalent clauses or one or more trivially true clauses (such as  $\{p, \neg p\}$ ). In all other cases, the clauses produced by *Clausify* are identical to those given by Pelletier (down to alpha-conversion where applicable)

It is difficult to know what conclusions to draw from these results; *Clausify* is clearly performing correctly, and while it is not producing as few clauses as given by Pelletier, it is producing equivalent results which are reduced to the minimal set in the early stages of a theorem prover. In this respect, it is successfully fulfilling its requirements: it is a minimal implementation of conventional

| Problem Number | Formula  | No. Clauses |                 |
|----------------|--|-------------|-----------------|
|                |  | Pelletier   | <i>Clausify</i> |
| 1              | $(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$                                  | 4           | 5               |
| 2              | $\neg\neg p \Leftrightarrow p$   | 2           | 2               |
| 3              | $\neg(p \Rightarrow q) \Rightarrow (q \Rightarrow p)$  | 4           | 4               |
| 4              | $(\neg p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow p)$                                  | 4           | 5               |
| 5              | $((p + q) \Rightarrow (p \vee r)) \Rightarrow (p \vee (q \Rightarrow r))$                        | 4           | 5               |
| 18             | $\exists y. \forall x. (f(y) \Rightarrow f(x))$  | 2           | 2               |
| 19             | $\exists x. \forall y. \forall z. ((p(y) \Rightarrow q(z)) \Rightarrow (p(x) \Rightarrow q(x)))$ | 2           | 2               |

Table 4.1: Numbers of clauses produced by *Clausify* for sample inputs

clause form conversion.

### 4.3 Conventional Mode versus Renaming

The majority of simple problems submitted to *Clausify* did not, unfortunately, require renaming; of those that did, let us consider problem 12:  $((p \Leftrightarrow q) \Leftrightarrow r) \Leftrightarrow (p \Leftrightarrow (q \Leftrightarrow r))$ , for which Pelletier gives eight clauses. The conventional method of *Clausify* gives the following clauses:

|  |  |
|--|--|
| $\{\neg r, \neg q, p, q, r, p\}$           | $\{\neg r, \neg q, p, \neg q, \neg r, p\}$           |
| $\{\neg r, \neg p, q, q, r, p\}$           | $\{\neg r, \neg p, q, \neg q, \neg r, p\}$           |
| $\{\neg r, \neg q, p, \neg p, \neg r, q\}$ | $\{\neg r, \neg q, p, \neg p, \neg q, r\}$           |
| $\{\neg r, \neg p, q, \neg p, \neg r, q\}$ | $\{\neg r, \neg p, q, \neg p, \neg q, r\}$           |
| $\{p, q, r, q, r, p\}$                     | $\{p, q, r, \neg q, \neg r, p\}$                     |
| $\{\neg p, \neg q, r, q, r, p\}$           | $\{\neg p, \neg q, r, \neg q, \neg r, p\}$           |
| $\{p, q, r, \neg p, \neg r, q\}$           | $\{p, q, r, \neg p, \neg q, r\}$                     |
| $\{\neg p, \neg q, r, \neg p, \neg r, q\}$ | $\{\neg p, \neg q, r, \neg p, \neg q, r\}$           |
| $\{\neg q, p, r, p, \neg r, q\}$           | $\{\neg q, p, r, p, \neg q, r\}$                     |
| $\{\neg p, q, r, p, \neg r, q\}$           | $\{\neg p, q, r, p, \neg q, r\}$                     |
| $\{\neg q, p, r, \neg p, q, r\}$           | $\{\neg q, p, r, \neg p, \neg q, \neg r\}$           |
| $\{\neg p, q, r, \neg p, q, r\}$           | $\{\neg p, q, r, \neg p, \neg q, \neg r\}$           |
| $\{p, q, \neg r, p, \neg r, q\}$           | $\{p, q, \neg r, p, \neg q, r\}$                     |
| $\{\neg p, \neg q, \neg r, p, \neg r, q\}$ | $\{\neg p, \neg q, \neg r, p, \neg q, r\}$           |
| $\{p, q, \neg r, \neg p, q, r\}$           | $\{p, q, \neg r, \neg p, \neg q, \neg r\}$           |
| $\{\neg p, \neg q, \neg r, \neg p, q, r\}$ | $\{\neg p, \neg q, \neg r, \neg p, \neg q, \neg r\}$ |

Twenty-four of these thirty-two clauses can be shown to be trivially true, leaving eight which, after deleting duplicate terms, correspond to the Pelletier clauses.

The same formula submitted to the renaming method of *Clausify* gives:

|   |
|---|
| $\{\neg ren1(r, q, p), \neg r, \neg q, p\}$ |
| $\{\neg ren1(r, q, p), \neg r, \neg p, q\}$ |
| $\{\neg ren1(r, q, p), p, q, r\}$           |
| $\{\neg ren1(r, q, p), \neg p, \neg q, r\}$ |
| $\{\neg q, p, r, ren1(r, q, p)\}$           |

$$\begin{aligned}
&\{\neg p, q, r, \text{ren1}(r, q, p)\} \\
&\{p, q, \neg r, \text{ren1}(r, q, p)\} \\
&\{\neg p, \neg q, \neg r, \text{ren1}(r, q, p)\} \\
&\{\text{ren1}(r, q, p), q, r, p\} \\
&\{\text{ren1}(r, q, p), \neg q, \neg r, p\} \\
&\{\text{ren1}(r, q, p), \neg p, \neg r, q\} \\
&\{\text{ren1}(r, q, p), \neg p, \neg q, r\} \\
&\{\neg \text{ren1}(r, q, p), p, \neg r, q\} \\
&\{\neg \text{ren1}(r, q, p), p, \neg q, r\} \\
&\{\neg \text{ren1}(r, q, p), \neg p, q, r\} \\
&\{\neg \text{ren1}(r, q, p), \neg p, \neg q, \neg r\}
\end{aligned}$$

That is, sixteen unique clauses with no easy optimisations available. However, these clauses are shorter — four terms as opposed to six — and from the timings in table 4.2, we can see that the renaming has nevertheless resulted in an overall speed improvement; the extra overhead in preprocessing is negligible compared to the reduction in time spent in the theorem prover. (The renamed subformulae are underlined in the table)

It is particularly interesting to look at problem 30, since the use of renaming does not reduce the number of clauses; recall that this case was explicitly allowed in equation 2.3 to simplify the inequalities later on. The difference in overall time taken with and without renaming is negligible, indicating that this tradeoff was one which has not caused any performance penalty. There was slightly less time spent in the preprocessor, however, indicating that renaming has nevertheless produced a set of formula which is slightly easier to convert to clause form.

The behaviour of *Clausify* on these sample problems has been very successful: there has been a speed increase and a reduction in clauses and clause complexity in every case. The increase in complexity of the preprocessor with renaming has resulted in a faster preprocessor, indicating that the overhead of testing for renaming does not need to be a consideration in whether this method is used; indeed, the results indicate that renaming as implemented in *Clausify* brings speed improvements even where the number of clauses is unchanged. There is no possibility of renaming increasing the number of clauses produced, because of the selection method discussed in section 2.4.1

Table 4.2: Averaged timings for *Claustify* with and without renaming

| No. | Formula   | <i>Claustify</i> |       | Overall |        | Clauses/Terms[1] |      |
|-----|---|------------------|-------|---------|--------|------------------|------|
|     |   | w/o              | with  | w/o     | with   | w/o              | with |
| 12  | $\frac{(p \Leftrightarrow q) \Leftrightarrow r}{(p \Leftrightarrow (q \Leftrightarrow r))} \Leftrightarrow$   | 0.96s            | 0.82s | 2.39s   | 1.90s  | 32/6             | 24/4 |
| 13  | $\frac{(p \vee (q \wedge r)) \Leftrightarrow}{((p \vee q) \wedge (p \vee r))} \Leftrightarrow$  | 0.72s            | 0.72s | 1.17s   | 1.15s  | 12/4             | 10/3 |
| 14  | $\frac{(p \Leftrightarrow q) \Leftrightarrow ((q \vee \neg p) \wedge (\neg q \vee p))}{\wedge (\neg q \vee p)}$   | 0.71s            | 0.66s | 1.13s   | 1.125s | 12/4             | 10/3 |
| 30  | $\frac{(\forall x.(F(x) \vee G(x) \Rightarrow \neg H(x)) \wedge \forall x.((G(x) \Rightarrow \neg I(x)) \Rightarrow F(x) \wedge H(x))) \Rightarrow \forall x.I(x)}{\Rightarrow \forall x.I(x)}$   | 0.71s            | 0.69s | 1.02s   | 1.02   | 7/2              | 7/2  |
| 33  | $\frac{\forall x.(P(a) \wedge (P(x) \Rightarrow P(b)) \Rightarrow P(c)) \Leftrightarrow \forall x.((\neg P(a) \vee (P(x) \vee P(c))) \wedge (\neg P(a) \vee (\neg P(b) \vee P(c))))}{\wedge (\neg P(a) \vee (\neg P(b) \vee P(c)))}$  | 0.96s            | 0.90s | 2.07s   | 1.83s  | 31/5             | 18/4 |
| 34  | $\frac{(\exists x.\forall y.(P(x) \Leftrightarrow P(y)) \Leftrightarrow (\exists x.Q(x) \Leftrightarrow \forall y.P(y))) \Leftrightarrow (\exists x.\forall y.(Q(x) \Leftrightarrow Q(y)) \Leftrightarrow (\exists x.P(x) \Leftrightarrow \forall y.P(y)))}{(\exists x.P(x) \Leftrightarrow \forall y.P(y))}$ | 4.67s            | 1.19s | [2]     | [2]    | 128/8            | 32/5 |

[1] (Number of clauses produced)/(Maximum number of terms in each clause)

[2] *SETHEO* did not terminate in a reasonable amount of time for these runs.



## Chapter 5

# Conclusions

*A summary of the successes of the project*

---

The preprocessor implemented in this project, *Clausify*, has been found to successfully produce clause forms from general first order logic which, while not being in their smallest form, may be trivially reduced to such a form.

In addition, the renaming method described in section 2.4 as implemented in *Clausify* has been found to produce at most as many, and often fewer, clauses when used. Its performance is such that in the preprocessor itself, the overhead of performing renaming is more than outweighed by the increase in speed of the subsequent conversion to clause form. The increase in speed of the theorem prover being studied, *SETHEO*, reinforces the success of the method.

*Clausify* has successfully fulfilled the goals set out in the project aims, and it has been shown that the renaming method as implemented in *Clausify* is a viable and useful improvement to clause form conversions.



# Appendix A

## Library Routines

Several builtin library routines in SWI Prolog were found to be unavailable in the copy of SICStus Prolog available on the Thor teaching system; to solve this problem several library routines have been reimplemented and are given here.

### A.1 **append/3: Concatenating Lists**

```
append([], L, L) .
```

```
append([H|T], L, [H|R]) :- append(T, L, R).
```

### A.2 **member/2: List Membership**

```
member(A,[A|_]) .
```

```
member(A,[_|T]) :- member(A,T).
```

### A.3 **delete/3: Removal From Lists**

```
delete([], _, []) :- !.
```

```
delete([Elem|Tail], Elem, Result) :- !,  
                                     delete(Tail, Elem, Result).
```

```
delete([Head|Tail], Elem, [Head|Rest]) :- delete(Tail, Elem, Rest).
```

## A.4 reverse/2: List Reversal

```
reverse(L1, L2) :- reverse(L1, [], L2).
```

```
reverse([], List, List) .
```

```
reverse([Head|List1], List2, List3) :- reverse(List1, [Head|List2], List3).
```

## A.5 gensym/2: Generating a Unique Symbol

**gensym/2** is a difficult function because it requires persistent state: each generated symbol must be given a unique number, so a record of used numbers must be kept. The version of **gensym/2** presented here is taken from Clocksin and Mellish [CM87] pages 155–156.

```
get_num(Root, Atom) :- name(Root, Name1),
                       integer_name(Num, Name2),
                       append(Name1, Name2, Name),
                       name(Atom, Name).
```

```
get_num(Root, Num) :- retract(current_num(Root, Num1)),
                      !,
                      Num is Num1+1,
                      asserta(current_num(Root, Num)).
```

```
integer_name(Int, List) :- integer_name(Int, [], List).
```

```
integer_name(I, Sofar, [C|Sofar]) :- I < 10,
                                     !,
                                     C is I+48.
```

```
integer_name(I, Sofar, List) :- Tophalf is I/10,
                                Bothalf is I mod 10,
                                C is Bothalf+48,
                                integer_name(Tophalf, [C|Sofar], List).
```

# Bibliography

- [Boy90] Thierry Boy de la Tour. Minimizing the number of clauses by renaming. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, LNAI 449, pages 558–572. Springer, 1990.
- [CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 3rd edition, 1987.
- [Egl94] Uwe Egly. On the value of antiprenexing. In *Logic Programming and Automated Reasoning, 5th International Conference, LPAR'94*, volume 822 of LNAI, pages 69–83. Springer, July 1994.
- [NRW98] Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In Claude Kirchner and Hélène Kirchner, editors, *Automated Deduction — CADE-15 International Conference*, LNAI 1421, pages 397–411. Springer, 1998.
- [Pau97] Lawrence C Paulson. *Logic and Proof (Computer Science Tripos Part IB Lecture Notes)*. Computer Laboratory, University of Cambridge, 1997.
- [Pel86] Francis Jeffrey Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.



# Index

∴, 4, 14, 20

=., 17

alpha-conversion, 26

atom, 3

backtracking, 18

candidate list, 19, 21

definition, 6, 22

equivalences, 13

free variables, 3

implications, 12

nested renamings, 21

Otter, 1

polarity, 5, 13

position, 5

quantifiers, 4

redundant calls, 12

redundant clauses, 13

renaming, 6, 19

SETHEO, 1

substitutions, 11, 14, 17, 22

three-phase, 14

unification, 19

unified, 17, 21



Dan Sheridan (djs52)  
King's College

Part II Computer Science Project Proposal  
**Preprocessing for Resolution Proof Procedures**

21st October, 1998

**Project Originator:** Lawrence C Paulson

**Special Resources:** Extra disc space allocation on Thor: 15Mb

**Project Supervisor:** Lawrence C Paulson

**Signature:**

**Director of Studies:** Ken Moody

**Signature:**

**Overseers:** Jean Bacon and David Greaves

## Introduction

Theorems are normally expressed in general first order logic; in order for resolution theorem provers to deal with such theorems, they must first be converted to clause form. A conversion which produces fewer clauses might be expected to produce input that is quicker to prove.

Nonnengart, Rock and Weidenbach describe an algorithm for renaming subformulae which is based on work by Boy de la Tour. This approach guarantees to produce the minimum number of clauses by carefully selecting the subformulae to be renamed.

The paper also describes a second technique based on variants of Skolemization which can also lead to the production of fewer clauses.

## Work to be Undertaken

This project will concentrate on implementing the algorithm for the renaming of subformulae. In the early stages of the project all subformulae will be renamed, the goal is to use Boy de la Tour's approach of only replacing a subformula if it would decrease the number of clauses produced.

Obviously this work will be of no use if the preprocessing does not result in an improvement in the performance of a theorem prover. Hence an important part of the project will be to examine whether renaming algorithm does reduce the amount of time spent in the theorem prover, and also whether the extra time spent computing Boy de la Tour's method of choosing subformulae results in sufficient reduction in proving time to make it worthwhile.

The project will be implemented on the Thor teaching system in *SICStus Prolog* and initially the theorem prover *SETHEO* will be used for the testing. A number of test sets of first-order problems are available, and part of the preliminary work for the project will be to select one to be used throughout.

The Skolemization based methods will be investigated if time allows, as will how the different conversion procedures affect the performance of another theorem prover, *Otter*.

## **Timetable**

### **Weeks 1 and 2 (31st Oct – 13th Nov)**

Preparatory work: reading the papers referred to above and following up any required references; getting to know *SETHEO* and *SICStus Prolog*. Researching problem sets for testing the software.

### **Weeks 3 and 4 (14th – 27th Nov)**

Write the framework of the software: a program that parses a statement in general first order logic, converts to some internal tree representation, converts to clause normal form and outputs clauses suitably for input to *SETHEO* along with the number of clauses and symbols in the output. This will later be used as a benchmark for the more complex system to be developed.

### **Weeks 5 and 6 (28th Nov – 11th Dec)**

Write the basic renaming system, where all subformulae are renamed. Compare performance with the base CNF system.

### **Weeks 7 and 8 (9th – 22nd Jan)**

Begin work on Boy de la Tour's selection method.

### **Week 9 (23rd – 29th Jan)**

Prepare progress report.

### **Weeks 10 to 12 (23rd Jan – 19th Feb)**

Complete the selection method code and begin comparing its performance with the CNF and naïve renaming methods. Tools to automate this will need to be developed — it may be worth repeating each run of *SETHEO* several times to obtain an average run time.

### **Weeks 13 and 14 (20th Feb – 5th Mar)**

Collate results of the test runs

**Weeks 15 and 16 (2nd Apr – 16th Apr)**

(This is during the Easter Vacation) These weeks will be used either to catch up with missed deadlines or to implement Skolemization, depending on whether the work is up to date.

Begin dissertation if time allows.

**Weeks 17 to 19 (17th Apr – 7th May)**

Dissertation, aiming to complete by week 19.

**Week 20 (8th – 14th May)**

Tidying up dissertation, proofreading, printing.