

Gareth Evans | IPM 4097 Project | 2001-2002

Knuth-Bendix Rewriting Methods

Abstract

The goal of this project was to implement efficient rewriting methods for monoids by using the computer software library known as 'frmon' provided by Professor Larry Lambe of the *Multidisciplinary Software Systems Research Corporation* (MSSRC). This involved implementing the usual Knuth-Bendix critical pairs completion algorithm on the free monoid, and also an extended Knuth-Bendix critical pairs completion algorithm for induced monoid actions, based on the work carried out by Brown/Heyworth [A]. The resulting programs are accessible by first logging in to the math2 server and then going to the /mw3/usr/lib/freemon2/gareth/kbia directory.

The free monoid library 'frmon' is essentially an extension of the C programming language, in the sense that the library provides variables and functions for representing and manipulating monoids that can be used in normal C source code. We can think of the library as providing a base upon which we can build complex mathematical algorithms, using merely a simple text editor in the *FreeBSD Unix* Operating System.

The Problem

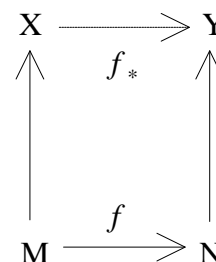
Given a monoid presentation for a monoid M, such as the shown monoid presentation for the symmetric group S(3), our first task is to find a complete rewrite system for M such that given any word 'w' in the generators of M, we can reduce that word to a 'normal form' i.e. an element of M. Considering S(3), which has six elements, the task in this case is to determine which of the six elements of S(3) corresponds to an arbitrary word w in the monoid generators of S(3) after reduction, reduction being the process where we use rewrite rules of the form $l \rightarrow r$ to simplify a given word w.

$$\begin{aligned} A^+ \times A^- &\rightarrow \lambda \\ A^- \times A^+ &\rightarrow \lambda \\ B^+ \times B^- &\rightarrow \lambda \\ B^- \times B^+ &\rightarrow \lambda \\ (A^+)^3 &\rightarrow \lambda \\ (B^+)^2 &\rightarrow \lambda \\ (A^+ \times B^+)^2 &\rightarrow \lambda \end{aligned}$$

In order to perform the above task, we need algorithms for calculating the *elements* of a monoid, for finding a *complete rewrite system* for a monoid, and for *reducing* a given word using a given rewrite system. These algorithms were implemented by converting similar algorithms found in 'IdRel' [B], a package forming part of the third version of the GAP program, and they then provided templates for my part of the coding process, namely the implementation of a Knuth-Bendix critical pairs completion algorithm for induced monoid actions.

Induced Monoid Actions

Consider that we are given two monoids M and N, a monoid M-set X, and a monoid morphism $f: M \rightarrow N$. Our task here is to find a monoid N-set Y with the property that f preserves the M-action, so that there exists a function f_* such that $f_*(x^m) = (f_*(x))^{f(m)}$, and a universal property is satisfied. If we can find such an N-set then the N-action is called the action of N **induced** from that of M by f.



It can be shown that one such set Y is the set $X \times N / \langle\langle R \rangle\rangle$, where $\langle\langle R \rangle\rangle$ is the equivalence relation generated by the set of rewrite rules $(x, f(m)n) \rightarrow (x^m, n)$. In this case, on applying the Knuth-Bendix critical pairs completion algorithm to this set of rewrite rules, our goal is to provide a complete rewrite system for the set $X \times N$, so that any element of the set $X \times N$ can be reduced to a unique normal form (by using rewrite rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$). Given such a complete rewrite system, we can then go on to find all of the equivalence classes that make up the set Y.

In order to do all of the above, we again need suitable algorithms for calculating elements, for finding complete rewrite systems and for reducing words, and the purpose of this report is to guide the reader through the processes involved in obtaining and then testing these algorithms.

Table of Contents

Page	Description
2	<i>Abstract</i>
5-8	<i>Chapter 1: Preliminary Results</i>
5	<i>1.1: Basic Facts</i>
6	<i>1.2: Monoid Theory</i>
7	<i>1.3: Monoid Actions</i>
9-30	<i>Chapter 2: String Rewriting Systems</i>
9	<i>2.1: Introduction</i>
10	<i>2.2: The Knuth-Bendix Critical Pairs Completion Algorithm</i>
11	<i>2.3: Using the Knuth-Bendix Critical Pairs Completion Algorithm</i>
15	<i>2.4: Completion</i>
16	<i>2.5: Obtaining the Elements of a Monoid</i>
18	<i>2.6: Knuth-Bendix Rewriting for Actions</i>
18	<i>2.6.1: Defining an M-set</i>
19	<i>2.6.2: Induced Monoid Actions</i>
21	<i>2.6.3: Induced Monoid Actions: An Example</i>
23	<i>2.7: The Knuth-Bendix Critical Pairs Completion Algorithm for Induced Monoid Actions</i>
25	<i>2.8: Using the Algorithm in Section 2.7</i>
27	<i>2.9: Obtaining the Elements of the set Y</i>
30	<i>2.10: How much information do we need to find the elements of the set Y?</i>

Table of Contents

Page	Description
31-55	<i>Chapter 3: Implementation</i>
31	<i>3.1: A Simple Example on the use of the library</i>
31	<i>3.2: Implementing the Algorithms discussed in Chapter 2</i>
32	<i>3.2.1: fMonWordReduce</i>
33	<i>3.2.2: fMonRulesReduce</i>
34	<i>3.2.3: fMonKnuthBendix</i>
38	<i>3.2.4: fMonMonoidElements</i>
39	<i>3.2.5: The 'kbe' and 'kba' Programs: Part 1</i>
40	<i>3.2.6: intFMonListWordReduce</i>
42	<i>3.2.7: intFMonRulesReduce</i>
43	<i>3.2.8: intFMonKnuthBendix</i>
48	<i>3.2.9: intFMonMonoidElements</i>
50	<i>3.2.10: The 'kbe' and 'kba' Programs: Part 2</i>
56-64	<i>Chapter 4: Analysis of the Programs</i>
56	<i>4.1: Using the Programs</i>
59	<i>4.2: Comparisons</i>
62	<i>4.3: Improvements</i>
65	<i>Conclusions</i>
66	<i>Bibliography</i>
67-	<i>Appendices</i>
A1-A23	<i>Appendix 1: The User Manual</i>
B1-B35	<i>Appendix 2: Source Code</i>
C1-C16	<i>Appendix 3: Program Output</i>

Chapter 1: Preliminary Results

In order to talk about what the computer programs associated with this project accomplish, we must first set up some underlying theory...

1.1: Basic Facts

Definition 1.1: A **set** is a collection of objects where the objects are called elements or members of the set. It is usual to list the elements of a set in a pair of curly braces { }.

Example 1.2: The set A of all integers between 0 and 5 is given by $A = \{0, 1, 2, 3, 4, 5\}$, or $A = \{n \in \mathbf{Z}: 0 \leq n \leq 5\}$.

Definition 1.3: If the number of elements in a set S is a finite number 'n', then we say that S has **order** 'n' and we write $|S| = n$.

Example 1.4: For the set A above, $|A| = 6$.

Definition 1.5: A **binary operation** * on a set S is a function which associates with each ordered pair (a, b) of elements of S a uniquely defined element $(a*b) \in S$, i.e. we have $*: S \times S \rightarrow S$.

Example 1.6: Addition, +, is a binary operation over the real numbers:
For any two real numbers $a, b \in \mathbf{R}$, the sum of the two numbers is also a real number: $a + b = c \in \mathbf{R}$.

Definition 1.7: Let $\{S, *\}$ be a set S with a binary operation *. $\{S, *\}$ is called a **monoid** if the following axioms are satisfied:

- (1) The binary operation * is associative, i.e.
 $x*(y*z) = (x*y)*z \quad \forall x, y, z \in S$;
- (2) The set S has an identity element λ such that
 $\lambda*x = x*\lambda = x \quad \forall x \in S$.

Example 1.8: $\{\mathbf{R}, \times\}$, the set of real numbers under multiplication, forms a monoid.

Proof: Because multiplication is an *associative operation* $(x \times (y \times z)) = (x \times y) \times z$ for all $x, y, z \in \mathbf{R}$, and because the *identity element* for multiplication is the number 1 ($1 \times x = x \times 1 = x$ for all $x \in \mathbf{R}$), then $\{\mathbf{R}, \times\}$ is a monoid.

Note that there is no need in the above to exclude zero from the set \mathbf{R} because no *inverses* are required (as would be the case when dealing with groups).

Definition 1.9: A monoid is **Abelian** if $x*y = y*x$ for all $x, y \in S$.

1.2: Monoid Theory

Let us now give some theory regarding monoids and monoid presentations.

Definition 1.10: A **presentation** for a monoid M is a way of representing the elements of the monoid by giving a list of the *generators* of the monoid (which form the *alphabet* associated with M) together with a set of *relators* which specify which words in the generators of M can be rewritten as the identity word.

Example 1.11: Consider the symmetric group on 3 elements, $S(3)$, which we normally think of as consisting of the six elements (1) , $(1\ 2)$, $(1\ 3)$, $(2\ 3)$, $(1\ 2\ 3)$ and $(1\ 3\ 2)$. $S(3)$ can be thought of as a monoid by giving the following *monoid presentation* for the group $S(3)$:

$$S(3) = \langle A^+, B^+, A^-, B^- \mid A^+ \times A^- = \lambda, A^- \times A^+ = \lambda, B^+ \times B^- = \lambda, B^- \times B^+ = \lambda, (A^+)^3 = \lambda, (B^+)^2 = \lambda, (A^+ \times B^+)^2 = \lambda \rangle.$$

Notice that because a monoid has *no inverses*, we must specify positive and negative generators as well as providing the relators which say that a ‘*positive*’ generator multiplied by a ‘*negative*’ generator must be rewritten as the identity word.

Remark: From now on, we will drop the “ $= \lambda$ ” from our presentation notation so that, for example, the above monoid presentation for $S(3)$ changes to the following:

$$S(3) = \langle A^+, B^+, A^-, B^- \mid A^+ \times A^-, A^- \times A^+, B^+ \times B^-, B^- \times B^+, (A^+)^3, (B^+)^2, (A^+ \times B^+)^2 \rangle.$$

Definition 1.12: A *free monoid* on an alphabet A is denoted by A^* and is the set of all possible words in the alphabet A unrestricted by any relators.

Definition 1.13: Let G and H be monoids. A *homomorphism* from G to H is a map $\phi: G \rightarrow H$ such that, for all x and y in G , we have $\phi(xy) = \phi(x)\phi(y)$.

Definition 1.14: Let G and H be monoids. A *monoid morphism* from G to H is a map $\theta: G \rightarrow H$ such that θ is a **homomorphism** and the following property holds: $\theta(\lambda_G) = \lambda_H$, where λ_G and λ_H are the *identity words* in G and H respectively. Alternatively, we may think of a monoid morphism as being a map θ which preserves multiplication and maps identities to identities.

Definition 1.15: An isomorphism is a monoid morphism from G to H which is a **bijection**. In this case, we write $G \cong H$.

Remark: If $G \cong H$, then there exists monoid morphisms $\theta: G \rightarrow H$ and $\theta': H \rightarrow G$ such that $\theta\theta'$ is the identity map of G and $\theta'\theta$ is the identity map of H . In this situation, we can refer to the map θ' as the inverse of the map θ .

Definition 1.16: A relation R on a monoid M (i.e. a set of ordered pairs $(x, y) = xRy$, with $x, y \in M$) is an **equivalence relation** if R is reflexive ($xRx \ \forall x \in M$), symmetric ($xRy \Rightarrow yRx \ \forall x, y \in M$) and transitive (xRy and $yRz \Rightarrow xRz \ \forall x, y, z \in M$).

Definition 1.17: Given a relation R on a monoid M which is an equivalence relation, the **quotient monoid** M/R is the set of *equivalence classes* $[m]$ ($m \in M$) associated with R (with monoid multiplication $[m][n] = [mn]$ ($m, n \in M$)), where the equivalence class $[m]$ of an element $m \in M$ is the set $\{n \in M: nRm\}$.

Remark:

Given a monoid presentation for a monoid M , $M = \langle X \mid R \rangle$, we can now think of the monoid M as being the quotient monoid of the free monoid on the generators X by the equivalence relation generated by the relators R , i.e. $M \cong X^*/\langle\langle R \rangle\rangle$. We will comment further on the nature of $\langle\langle R \rangle\rangle$ (in chapter 2) once we have developed some more theory.

Definition 1.18:

Given a monoid M , a monoid morphism of the form $\theta: M \rightarrow M$ is called a monoid endomorphism.

1.3: Monoid Actions

In this section, we shall first give *two definitions* of what a monoid M -set is, and we shall then go on to show that these two definitions are in fact equivalent.

Definition 1.19:

Given a monoid M and a non-empty set X , the set X is a monoid M -set if there exists a homomorphism $\theta: M \rightarrow T_X$, where T_X is the full transformation monoid, the set of all endomorphisms from X to X .

For $m \in M$ and $x \in X$, we write the image $\theta(m)(x)$ as x^m or as $x \bullet m$.

Remark:

If we number the elements of X as $X = \{1, \dots, n\}$, then we may adopt a notation similar to permutation notation to write down the elements of T_X . For example, if $n = 2$, then it follows that

$$T_X = \left\{ \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 2 & 2 \end{pmatrix} \right\}.$$

We can easily see from the above that if $|X| = n$, then $|T_X| = n^n$.

Definition 1.20:

Consider that we have a monoid M and a non-empty set X . The set X is a monoid M -set if there is an *associative mapping* with an *identity* from $X \times M$ into X , under which the product of any member of X with any member of M is a member of X , such that for each $x \in X$ and each $m, n \in M$, we have $x \bullet (mn) = (x \bullet m) \bullet n$ and $x \bullet \lambda = x$, where λ is the identity element of M .

Remark:

We may also refer to a monoid M -set as defined in either of the two definitions above as the **action** of the monoid M on the set X .

Proposition 1.21:

The two definitions of a monoid M -set as defined in Definitions 1.19 and 1.20 are equivalent.

Proof:

Assume first that Definition 1.19 holds. We must show that given a monoid M -set as defined in Definition 1.19, we can construct a mapping with the properties defined in Definition 1.20.

Consider that the mapping is defined by $x \bullet m = \theta(m)(x)$. Let us now verify that this mapping has the required properties.

Because θ is a homomorphism, then we have $\theta(mn) = \theta(m)\theta(n)$.

It follows that $\theta(mn)(x) = (\theta(m)\theta(n))(x)$;

$$x \bullet (mn) = \theta(n)(x \bullet m) \text{ (composing left to right);}$$

$$x \bullet (mn) = (x \bullet m) \bullet n.$$

To show that $x \bullet \lambda = x$, all we need do is to show that $\theta(\lambda)$ is the identity transformation (because $x \bullet \lambda = \theta(\lambda)(x) = x$ if $\theta(\lambda)$ is the identity transformation).

Now $(x \bullet \lambda) \bullet \lambda = x \bullet (\lambda^2)$ by the above result;
 $(x \bullet \lambda) \bullet \lambda = x \bullet \lambda$ because $\lambda^2 = \lambda$
 $\Rightarrow y \bullet \lambda = y$ if we let $y = x \bullet \lambda$
 $\Rightarrow \theta(\lambda)(y) = y$
 $\Rightarrow \theta(\lambda)$ must be the identity transformation (it leaves y unchanged).

This completes the first half of the proof.

Assume now that we have a mapping that has the properties given in Definition 1.20. Because in this case $x \bullet m = y \in X$ is defined for all $x \in X$ and all $m \in M$, then we can *construct* a function θ of the required form by letting $x \bullet m = \theta(m)(x)$.

Now $x \bullet (mn) = (x \bullet m) \bullet n$ for all $x \in X$ and all $m, n \in M$
 $\Rightarrow \theta(mn)(x) = \theta(n)(x \bullet m)$ for all $x \in X$ and all $m, n \in M$
 $\Rightarrow \theta(mn)(x) = \theta(n)(\theta(m)(x))$ for all $x \in X$ and all $m, n \in M$
 $\Rightarrow \theta(mn)(x) = (\theta(m)\theta(n))(x)$ for all $x \in X$ and all $m, n \in M$
 $\Rightarrow \theta(mn) = \theta(m)\theta(n)$ for all $m, n \in M$
 $\Rightarrow \theta$ is a homomorphism, as required. QED.

Example 1.22:

Considering the symmetric group $S(3)$ on three elements again, we can form a monoid $S(3)$ -set X by specifying that X is the set consisting of the *elements* of $S(3)$ (i.e. $X = \{(1), (1\ 2), (1\ 3), (2\ 3), (1\ 2\ 3), (1\ 3\ 2)\}$), and that the homomorphism $\theta: S(3) \rightarrow T_X$ is given by letting (for every $a \in S(3)$) $\theta(a)$ be the transformation of elements of X given by multiplying each element of X on the right by a .

For example, to find out which element of T_X corresponds to $\theta((2\ 3))$, all we have to do is to multiply each element of X on the right by $(2\ 3)$ to give the set $\{(2\ 3), (1\ 3\ 2), (1\ 2\ 3), (1), (1\ 3), (1\ 2)\}$ (after simplification).

It follows that $\theta((2\ 3)) = \begin{pmatrix} (1) & (1\ 2) & (1\ 3) & (2\ 3) & (1\ 2\ 3) & (1\ 3\ 2) \\ (2\ 3) & (1\ 3\ 2) & (1\ 2\ 3) & (1) & (1\ 3) & (1\ 2) \end{pmatrix}$,
so that e.g. $\theta((2\ 3))(1\ 2) = (1\ 3\ 2)$.

Alternatively, in order to calculate $\theta(a)(x)$ for arbitrary $a \in S(3)$ and $x \in X$, we see that we only need calculate the monoid product ax . For the example where $a = (2\ 3)$ and $x = (1\ 2)$, we see that $\theta((2\ 3))(1\ 2) = (1\ 2)(2\ 3) = (1\ 3\ 2)$ as before.

Chapter 2: String Rewriting Systems

2.1: Introduction

Given a monoid presentation for a monoid M , can an arbitrary word w in the generators of M be simplified or reduced? The answer to this question depends on whether there exists a **set of rewrite rules** which say that a subword of w can be rewritten or simplified, thus producing another word w' .

Definition 2.1: A set of *rewrite rules* for a monoid M is a set of the form $\{(l_1 \rightarrow r_1), (l_2 \rightarrow r_2), \dots, (l_n \rightarrow r_n)\}$, where each *element* of the set is a rule which specifies that whenever we see the term l_i in any arbitrary word w , then we should replace it with the term r_i . We refer to the set of rewrite rules as a *rewrite system*.

As an example, consider the following set of ten rewrite rules:

- | | |
|--------------------------------------|------------------------------------------|
| (1) $A \times a \rightarrow \lambda$ | (6) $b \rightarrow B$ |
| (2) $a \times A \rightarrow \lambda$ | (7) $B \times a \rightarrow A \times B$ |
| (3) $B^2 \rightarrow \lambda$ | (8) $a \times B \rightarrow B \times A$ |
| (4) $a^2 \rightarrow A$ | (9) $B \times A \times B \rightarrow a$ |
| (5) $A^2 \rightarrow a$ | (10) $A \times B \times A \rightarrow b$ |

Given the arbitrary word $w = B^3 \times a \times b \times A^2 \times b^5$, we can simplify it using the above rules as follows (where we work from left to right and underline the piece of the word that changes):

$B^3 \times a \times b \times A^2 \times b^5$
→ $B \times a \times b \times A^2 \times b^5$ (using rule (3))
→ $A \times B \times \underline{b} \times A^2 \times b^5$ (using rule (7))
→ $A \times \underline{B} \times B \times A^2 \times b^5$ (using rule (6))
→ $A \times A^2 \times b^5$ (using rule (3))
→ $a \times A \times b^5$ (using rule (5))
→ b^5 (using rule (2))
→ B^5 (using rule (6) 5 times)
→ B (using rule (3) twice).

Now that we cannot reduce the word any further, we say that the **normal form** of the word $w = B^3 \times a \times b \times A^2 \times b^5$ is the word B .

Definition 2.2: A word w is a *normal form* if there are no rules in an associated rewrite system that **simplify** or **reduce** the word w .

It happens that the above rewrite system is the *complete* rewrite system for $S(3)$, the symmetric group on 3 elements — with the modifications $A = A^+$, $a = A^-$, $B = B^+$ and $b = B^-$. Note that this is the convention that we will use for the rest of this report — a capital letter will represent a 'positive' generator, whilst a lower case letter will represent a 'negative' generator.

Definition 2.3: A *complete* rewrite system is a rewrite system that rewrites any word w in the generators of a given monoid M as a *unique element* $m \in M$.

2.2: The Knuth-Bendix Critical Pairs Completion Algorithm

How can we be sure that a given rewrite system is *complete*? The answer lies in the application of an algorithm known as the Knuth-Bendix critical pairs completion algorithm, which attempts to *complete* any given rewrite system. Basically, this algorithm looks for overlaps of left hand sides of rewrite rules and produces new rules by reducing overlapped words in two ways, producing what is known as a **critical pair**. If the two elements in a critical pair differ after reduction, then we produce a new rule based on the order in which we define words to be ordered in M .

We continue to do this until no new rules are found during a particular iteration of the algorithm — we therefore know that after this our rewrite system is complete, and thus we may then reduce an arbitrary word w in the generators of a monoid M to a unique element $m \in M$.

Definition 2.4: A *total order* on a monoid M is a system which tells you which word is the ‘**bigger**’ out of any two arbitrary words. For example, the *lexicographic order*, known simply as ‘*lex*’, orders words **alphabetically**, while the ‘*length-lex*’ order orders the words by **length** first, and then (if the lengths are equal) by alphabetical order. Unless otherwise stated, we shall use the *length-lex* order for the remainder of this report.

Example 2.5: Consider a monoid M which has an associated alphabet $\alpha = \{a, b, c, d\}$. Using *lex order* ($a < b < c < d$), it follows that $ac > ab$ and $dca > baacd$, whilst using *length-lex order*, it follows that $dd > a$, $baacd > dca$ and $ab > aa$.

Definition 2.6: A *well-ordering* on a monoid M is a total order on M in which there is no infinite sequence of words $w_1 > w_2 > \dots$, where the w_i are words in the generators of M .

Definition 2.7: A total order on a monoid M with generators G is *admissible* if $y > z$ implies $uyv > uzv$ for all $y, z, u, v \in G^*$.

Remark: The total orders *lex* and *length-lex* are easily shown to be admissible.

Definition 2.8: A rewrite system R is *compatible* with an *admissible well-ordering* if $l > r$ for all $(l, r) \in R$.

Definition 2.9: Two words w_1 and w_2 are said to overlap if they share a common subword, i.e. we have $w_1 = ulv$ and $w_2 = plq$, where u, v, p and q are arbitrary (perhaps empty) words, and l is **not** the empty word. We refer to l as the overlap word.

Let us now consider the Knuth-Bendix critical pairs completion algorithm in more detail, and, in particular, we shall consider how we could implement the algorithm on a computer.

Algorithm 2.10: **The Knuth-Bendix Critical Pairs Completion Algorithm for a Free Monoid.**

Input: An *initial rewrite system* A (using an *admissible well-ordering*) which is associated with a presentation for a monoid M .

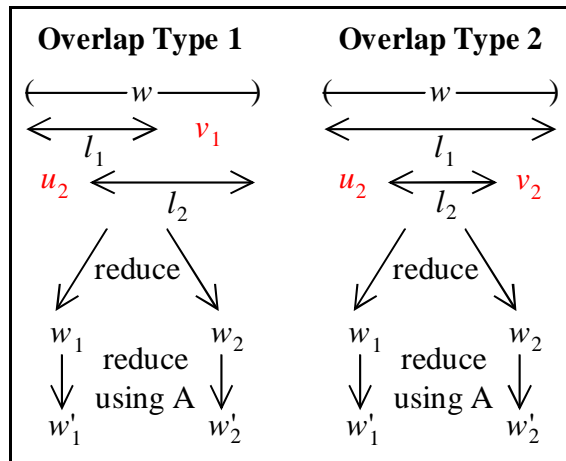
Output: A *complete rewrite system for* M , so that any word in the generators of M reduces to a **unique** element of M (a normal form).

Algorithm: Let $A = \{a_1, \dots, a_n\}$ be the set of initial rewrite rules. For any pair of rules $a_i = (l_i \rightarrow r_i)$ and $a_j = (l_j \rightarrow r_j)$ ($i, j = 1, \dots, n$), look for any *overlaps* between l_i and l_j . If an overlap is found, then reduce the overlap word w in two ways — first using rule a_i and then using rule a_j — to give two words w_1 and w_2 . Reduce w_1 and w_2 using A to give a **critical pair** of words w'_1 and w'_2 .

If $w'_1 = w'_2$, then do nothing; but if $w'_1 \neq w'_2$, then add the rule $(w'_1 \rightarrow w'_2)$ or the rule $(w'_2 \rightarrow w'_1)$ to A , based on the *total order* chosen, i.e. add $(w'_1 \rightarrow w'_2)$ if $w'_1 > w'_2$, and add $(w'_2 \rightarrow w'_1)$ if $w'_2 > w'_1$.

Repeat the above until no more new rules are added to A during any particular *pass* of the algorithm. Further, after every pass of the algorithm, analyse the list of rules to see if the list contains any redundant rules, i.e. rules which are implied by the other rules in the list.

Remark: There are *two* types of overlap word, summarised by the following diagram:



Overlap Type 1:

$$w = l_1 v_1 = u_2 l_2$$

$$w_1 = r_1 v_1, w_2 = u_2 r_2$$

Overlap Type 2:

$$w = l_1 = u_2 l_2 v_2$$

$$w_1 = r_1, w_2 = u_2 r_2 v_2$$

In order to enhance our understanding of the above algorithm, let us now consider an example on the use of the algorithm.

2.3: Using the Knuth-Bendix Critical Pairs Completion Algorithm

Consider the symmetric group on three elements, $S(3)$, with monoid presentation (as given on page 6)

$$S(3) = \langle A^+, B^+, A^-, B^- \mid A^+ \times A^-, A^- \times A^+, B^+ \times B^-, B^- \times B^+, (A^+)^3, (B^+)^2, (A^+ \times B^+)^2 \rangle.$$

As discussed previously, if we let $A = A^+$ and $a = A^-$, etc., then the monoid presentation changes to

$$S(3) = \langle A, a, B, b \mid Aa, aA, Bb, bB, A^3, B^2, ABAB \rangle.$$

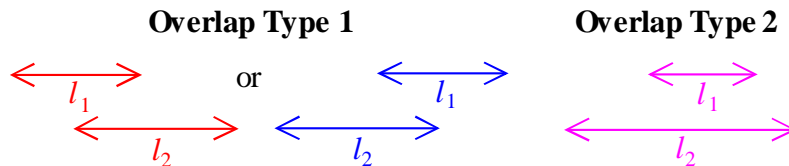
To *start* the algorithm with this presentation for $S(3)$ as input, we set up a list of initial rewrite rules taken from the given presentation:

INITIAL LIST OF REWRITE RULES

- | | |
|------------------------------|--------------------------------|
| (1) $Aa \rightarrow \lambda$ | (5) $A^3 \rightarrow \lambda$ |
| (2) $aA \rightarrow \lambda$ | (6) $B^2 \rightarrow \lambda$ |
| (3) $Bb \rightarrow \lambda$ | (7) $ABAB \rightarrow \lambda$ |
| (4) $bB \rightarrow \lambda$ | |

According to the algorithm, we must now iterate the process of looking for *overlaps* between left hand sides of rewrite rules until no new rule is added to the list of rewrite rules during a particular iteration of this process.

At this stage, let us consider the process of finding overlaps between the left hand sides of two rewrite rules. If we assume that we are comparing the left hand sides of the two rules $a_1 = (l_1 \rightarrow r_1)$ and $a_2 = (l_2 \rightarrow r_2)$, with $|l_1| = \alpha$ and $|l_2| = \beta$, then there are two types of overlap to consider, as we have seen from the definition of the algorithm. Assuming (w.l.o.g.) that $\alpha \leq \beta$ (if $\beta < \alpha$, then just swap around the elements of a_1 and a_2 in what follows), then an overlap word will either be of ‘Type 1’, in which case l_1 will overlap l_2 on the *left* or on the *right*; or will be of ‘Type 2’, in which case l_1 will appear *inside* l_2 :



Looking at things from the computer’s perspective, an overlap occurs when some subword of l_1 equals some subword of l_2 . The pseudo code for finding overlaps could therefore look as follows, where the symbol ‘==’ represents ‘the same as’, $\text{suffix}(l_1, i)$ represents the suffix of the word l_1 of length i , $\text{prefix}(l_2, i)$ represents the prefix of the word l_2 of length i , and $\text{subword}(l_2, i, i+\alpha)$ represents the subword of the word l_2 starting at position i and finishing at position $i+\alpha$.

Red overlap:

```
for i = 1 to  $\alpha-1$ 
{
  if (suffix( $l_1$ , i) == prefix( $l_2$ , i))
  then an overlap has been found
  else no overlap is found
}
```

Blue overlap:

```
for i = 1 to  $\alpha-1$ 
{
  if (suffix( $l_2$ , i) == prefix( $l_1$ , i))
  then an overlap has been found
  else no overlap is found
}
```

Purple overlap:

```
for i = 1 to  $\beta-\alpha+1$ 
{
  if ( $l_1$  == subword( $l_2$ , i, i+ $\alpha$ ))
  then an overlap has been found
  else no overlap is found
}
```

If an overlap word is found, then we reduce as specified in the algorithm to obtain a critical pair of words w'_1 and w'_2 . We then decide whether to add a rule involving w'_1 and w'_2 to our list of rewrite rules based on the total order chosen.

Let us now start applying the algorithm to our example by looking for overlaps between the left hand side of rule (1), Aa, with itself, i.e. we have $l_1 = l_2 = \text{Aa}$. We can see that as there is only **one** possible overlap in this case — when $w = l_1 = l_2$ (an overlap of type 2), and as the words then reduce to $w_1 = \lambda$ and $w_2 = \lambda$, then there is no need to add anything to our list — it follows that $w'_1 = w'_2 = \lambda$. However, for this first step, let us consider what the *computer* would do when attempting to find overlaps between l_1 and l_2 — we will follow the pseudo code written down previously for the ‘red’, ‘blue’ and ‘purple’ overlaps:

$$l_1 = \text{Aa}; r_1 = \lambda; \alpha = |l_1| = 2.$$

$$l_2 = \text{Aa}; r_2 = \lambda; \beta = |l_2| = 2.$$

‘Red’ overlaps: i goes from 1 to $2-1 = 1$.

$i = 1$: *Question*: Is $(\text{suffix}(l_1, 1) == \text{prefix}(l_2, 1))$?

Answer: No, as $\text{suffix}(l_1, 1) = \text{a}$, and $\text{prefix}(l_2, 1) = \text{A}$.

Conclusion: No overlap found.

‘Blue’ overlaps: i goes from 1 to $2-1 = 1$.

$i = 1$: *Question*: Is $(\text{suffix}(l_2, 1) == \text{prefix}(l_1, 1))$?

Answer: No, as $\text{suffix}(l_2, 1) = \text{a}$, and $\text{prefix}(l_1, 1) = \text{A}$.

Conclusion: No overlap found.

‘Purple’ overlaps: i goes from 1 to $2-2+1 = 1$.

$i = 1$: *Question*: Is $(l_1 == \text{subword}(l_2, 1, 2))$?

Answer: Yes, as $l_1 = \text{Aa}$, and $\text{subword}(l_2, 1, 2) = \text{Aa}$.

Conclusion: Overlap found, and the overlap word is Aa.

We now *reduce* the overlap word Aa, first using $a_1 = (l_1 \rightarrow r_1)$ to give λ , and then using $a_2 = (l_2 \rightarrow r_2)$ to also give λ . It follows that as we cannot reduce λ any further using the initial list of seven rewrite rules, and as the two reduced words are equal, then there is **no need** to add another rewrite rule to our list of rewrite rules in this case.

As we have now considered all the potential overlaps involving the left hand side of rule 1 with itself, we can now go on to look at the next comparison — rule 1 with rule 2, say. Here, $l_1 = \text{Aa}$ and $l_2 = \text{aA}$, and we can see that we have a single type 1 overlap in this case (detected under the ‘red’ section of our pseudo code), the overlap word being AaA. On reduction, however, we get $w'_1 = \text{A}$ and $w'_2 = \text{A}$, so again there is no need to add a new rewrite rule to our list of rewrite rules as the reduced words are equal to each other.

As in the above, no new rules are added to our list of rewrite rules when we compare the left hand side of rule 1 with the left hand sides of rules 3 and 4, but when we compare rule 1 with rule 5, we do get a new rule, so let us again consider what the computer would do in this situation.

$$l_1 = \text{Aa}; r_1 = \lambda; \alpha = |l_1| = 2.$$

$$l_2 = \text{A}^3 = \text{AAA}; r_1 = \lambda; \beta = |l_2| = 3.$$

‘Red’ overlaps: i goes from 1 to $2-1 = 1$.

$i = 1$: *Question*: Is $(\text{suffix}(l_1, 1) == \text{prefix}(l_2, 1))$?

Answer: No, as $\text{suffix}(l_1, 1) = \text{a}$, and $\text{prefix}(l_2, 1) = \text{A}$.

Conclusion: No overlap found.

'Blue' overlaps: i goes from 1 to $2-1 = 1$.

$i = 1$: *Question*: Is $(\text{suffix}(l_2, 1) == \text{prefix}(l_1, 1))$?

Answer: Yes, as $\text{suffix}(l_2, 1) = A$, and $\text{prefix}(l_1, 1) = A$.

Conclusion: Overlap found, and the overlap word is AAAa.

We now *reduce* the overlap word AAAa, first using $a_1 = (l_1 \rightarrow r_1)$ to give AA, and then using $a_2 = (l_2 \rightarrow r_2)$ to give a. These words cannot be reduced any further using our initial set of seven rewrite rules, and as the two reduced words are not equal, then we have to add a new rewrite rule to our list based on the total order in use. If we assume that the total order in use is the *length-lex order*, then we see that as we have $AA > a$, it follows that we have to add the rule $a_8 = AA \rightarrow a$ to our list of rewrite rules.

'Purple' overlaps: i goes from 1 to $3-2+1 = 2$.

$i = 1$: *Question*: Is $(l_1 == \text{subword}(l_2, 1, 2))$?

Answer: No, as $l_1 = Aa$, and $\text{subword}(l_2, 1, 2) = AA$.

Conclusion: No overlap found.

$i = 2$: *Question*: Is $(l_1 == \text{subword}(l_2, 2, 3))$?

Answer: No, as $l_1 = Aa$, and $\text{subword}(l_2, 2, 3) = AA$.

Conclusion: No overlap found.

During the remainder of the first iteration of the algorithm, we would find that we would generate new rewrite rules by comparing rule 2 with rule 7, rule 3 with rule 6, and rule 3 with rule 7. These rewrite rules can be seen in the following list of rewrite rules:

LIST OF REWRITE RULES AFTER PASS 1 OF THE ALGORITHM:

- | | |
|--------------------------------|--------------------------|
| (1) $Aa \rightarrow \lambda$ | (8) $A^2 \rightarrow a$ |
| (2) $aA \rightarrow \lambda$ | (9) $BAB \rightarrow a$ |
| (3) $Bb \rightarrow \lambda$ | (10) $b \rightarrow B$ |
| (4) $bB \rightarrow \lambda$ | (11) $ABA \rightarrow b$ |
| (5) $A^3 \rightarrow \lambda$ | |
| (6) $B^2 \rightarrow \lambda$ | |
| (7) $ABAB \rightarrow \lambda$ | |

Now that we have finished the first iteration of the algorithm, we must analyse the above list of rewrite rules to see if any rules in the list are **redundant**, i.e. implied by all the other rules in the list. The way we do this is to remove each rule $a_i = (l_i \rightarrow r_i)$ from the list in turn ($i = 1, \dots, n$), and look to see if l_i reduces to r_i using only the *other* rules in the list. If l_i does reduce to r_i using this method, then rule a_i is redundant and we may remove it from the list.

For the above list of rewrite rules, we see that rules 3, 4, 5 and 7 are redundant:

Rule 3: LHS = Bb

$\rightarrow BB$ (by rule 10)

$\rightarrow 1$ (by rule 6)

= RHS.

Rule 5: LHS = AAA

$\rightarrow aA$ (by rule 8)

$\rightarrow \lambda$ (by rule 2)

= RHS.

Rule 4: LHS = bB

$\rightarrow BB$ (by rule 10)

$\rightarrow \lambda$ (by rule 6)

= RHS.

Rule 7: LHS = ABAB

$\rightarrow bB$ (by rule 11)

$\rightarrow \lambda$ (by rule 4)

= RHS.

Therefore, after reduction, our new list of rewrite rules looks as follows:

LIST OF REWRITE RULES AFTER PASS 1 AND AFTER REDUCTION:

- | | |
|-------------------------------|-------------------------|
| (1) $Aa \rightarrow \lambda$ | (5) $BAB \rightarrow a$ |
| (2) $aA \rightarrow \lambda$ | (6) $b \rightarrow B$ |
| (3) $B^2 \rightarrow \lambda$ | (7) $ABA \rightarrow b$ |
| (4) $A^2 \rightarrow a$ | |

We now go on to perform the second pass of the algorithm on the above list, doing exactly the same things as before. The algorithm will then continue until during a particular iteration of the algorithm, no new rules are added to our list of rewrite rules (when this happens, the list of rewrite rules becomes a *complete rewrite system*). In this particular example, no new rules are found during the **third** pass of the algorithm, and the complete rewrite system thus obtained is as follows:

COMPLETE REWRITE SYSTEM FOR S(3):

- | | |
|-------------------------------|--------------------------|
| (1) $Aa \rightarrow \lambda$ | (6) $b \rightarrow B$ |
| (2) $aA \rightarrow \lambda$ | (7) $ABA \rightarrow b$ |
| (3) $B^2 \rightarrow \lambda$ | (8) $a^2 \rightarrow A$ |
| (4) $A^2 \rightarrow a$ | (9) $aB \rightarrow BA$ |
| (5) $BAB \rightarrow a$ | (10) $Ba \rightarrow AB$ |

2.4: Completion

For a given input to the Knuth-Bendix critical pairs completion algorithm, will the algorithm terminate or not? In general, this is an unanswerable question — but if the input has some of the following properties, then we will be able to comment on whether the algorithm terminates or not.

Definition 2.11: A *noetherian* rewrite system is a rewrite system R which is compatible with an admissible well ordering, i.e. given an *arbitrary* word w , the process of reducing w using R is a *finite* process.

Definition 2.12: A *confluent* rewrite system is a rewrite system in which every arbitrary word u has a **unique** normal form, i.e. if the word u reduces to words v and w , then v and w reduce to a *common* term.

Definition 2.13: A rewrite system R is *locally confluent* if for arbitrary words w , x and y , $w \rightarrow x$ and $w \rightarrow y$ imply that there exists a word z such that both x and y reduce to the word z .

Definition 2.14: A rewrite system R has the *Church-Rosser property* if given words x and y such that x reduces to y using R and y reduces to x using R , then there exists a word z such that both x and y reduce to the word z .

Given an *initial rewrite system* A associated with a presentation for a monoid M (where A uses an *admissible well-ordering*), the Knuth-Bendix critical pairs completion algorithm attempts to find a *complete rewrite system for M*. Using the above definitions, we see that a complete rewrite system for M can equivalently be thought of as a noetherian confluent rewrite system for M .

It can be shown (see [5], page 54) that given any rewrite system A associated with a presentation for a monoid M (where A uses an *admissible well-ordering*), the Knuth-Bendix critical pairs completion algorithm will produce a, *possibly infinite*, noetherian confluent rewrite system for M . Obviously, if no finite noetherian confluent system exists, then the algorithm cannot terminate.

We have therefore found a condition for the completion of the algorithm — for a given input, if there exists a *finite noetherian confluent rewrite system* that is equivalent to the input, then the algorithm will terminate; otherwise, the algorithm will not terminate.

Remark: The purpose of each iteration of the algorithm is to produce a rewrite system that is *equivalent* to the input rewrite system. Therefore, if the algorithm terminates, the finite noetherian confluent rewrite system that it produces as output will be equivalent to the input rewrite system.

2.5: Obtaining the Elements of a Monoid

Consider a monoid M which has an associated presentation. Assuming that we have applied the Knuth-Bendix critical pairs completion algorithm to M to obtain a *complete rewrite system* for M , let us now consider how we would obtain the elements of M given just this complete rewrite system and M 's generators.

Algorithm 2.15: An algorithm to obtain the elements of a monoid M .

Inputs: A complete rewrite system for M ;
A list of M 's generators.

Output: A list of the *elements* of M .

Algorithm: Let the list G contain the generators for M , and let us also set up two lists called '*used*' and '*found*', where we place the identity word in '*found*' and let '*used*' be empty to begin with. Repeat the following until the '*found*' list is empty (when the algorithm terminates, the '*used*' list will contain all the elements of M):

- (a) Take the first word from '*found*' and place it in the '*used*' list.
- (b) Multiply the above word on the right with each element from G to give a list of elements E .
- (c) For each word in E , reduce the word using the complete rewrite system for M to give another list E' .
- (d) For each word in E' , check to see if the word is in either '*used*' or in '*found*'. If the word is in neither list, place the word in '*found*'.

Example 2.16: Let us return to the example of the symmetric group on three elements, $S(3)$. Recall that in Section 2.3 we obtained the following complete rewrite system for $S(3)$:

COMPLETE REWRITE SYSTEM FOR $S(3)$:

- | | |
|-------------------------------|--------------------------|
| (1) $Aa \rightarrow \lambda$ | (6) $b \rightarrow B$ |
| (2) $aA \rightarrow \lambda$ | (7) $ABA \rightarrow b$ |
| (3) $B^2 \rightarrow \lambda$ | (8) $a^2 \rightarrow A$ |
| (4) $A^2 \rightarrow a$ | (9) $aB \rightarrow BA$ |
| (5) $BAB \rightarrow a$ | (10) $Ba \rightarrow AB$ |

To obtain the *elements* of $S(3)$, we need the above complete rewrite system together with the following list of $S(3)$'s generators:

GENERATORS FOR $S(3)$:

- | | | | |
|-----|---|-----|---|
| (1) | A | (3) | a |
| (2) | B | (4) | b |

We can now apply Algorithm 2.15 to $S(3)$ to obtain $S(3)$'s elements, and we start by placing the identity word λ in 'found' and initialising 'used' to be empty.

Pass 1 of the algorithm:

$found = \{ \lambda \}; used = \{ \}$.

- $found = \{ \}; used = \{ \lambda \}$.
(Take the first word from 'found' and place it in the 'used' list).
- $E = \{ \lambda A, \lambda B, \lambda a, \lambda b \} = \{ A, B, a, b \}$.
(Multiply λ on the right with each element from G to give a list of elements E).
- $E' = \{ A, B, a, B \} = \{ A, B, a \}$.
(For each word in E , reduce the word using the complete rewrite system for M to give another list E').
- As 'A', 'B' and 'a' are not in either 'found' or in 'used', we place these three words in the 'found' list.
(For each word in E' , check to see if the word is in either 'used' or in 'found'. If the word is in neither list, place the word in 'found').

Pass 2 of the algorithm:

$found = \{ A, B, a \}; used = \{ \lambda \}$.

- $found = \{ B, a \}; used = \{ \lambda, A \}$
- $E = \{ AA, AB, Aa, Ab \} = \{ A^2, AB, Aa, Ab \}$
- $E' = \{ a, AB, \lambda, AB \} = \{ a, AB, \lambda \}$
- As the word 'AB' is *not* in either 'found' or in 'used', we place this word in the 'found' list.

Pass 3 of the algorithm:

$found = \{ B, a, AB \}; used = \{ \lambda, A \}$.

- $found = \{ a, AB \}; used = \{ \lambda, A, B \}$
- $E = \{ BA, BB, Ba, Bb \} = \{ BA, B^2, Ba, Bb \}$
- $E' = \{ BA, \lambda, AB, \lambda \}$
- As the word BA is *not* in either 'found' or in 'used', we place this word in the 'found' list.

Pass 4 of the algorithm:

$found = \{ a, AB, BA \}; used = \{ \lambda, A, B \}$.

- $found = \{ AB, BA \}; used = \{ \lambda, A, B, a \}$
- $E = \{ aA, aB, aa, ab \} = \{ aA, aB, a^2, ab \}$
- $E' = \{ \lambda, BA, A, BA \}$
- As *all* the words in E' either appear in 'found' or in 'used', this time we *do not add* any words to the 'found' list.

Pass 5 of the algorithm:

$found = \{ AB, BA \}; used = \{ \lambda, A, B, a \}$.

- $found = \{ BA \}; used = \{ \lambda, A, B, a, AB \}$
- $E = \{ ABA, ABB, ABa, ABb \} = \{ ABA, AB^2, ABa, ABb \}$
- $E' = \{ B, A, BA, A \}$
- As *all* the words in E' either appear in 'found' or in 'used', this time we *do not add* any words to the 'found' list.

Pass 6 of the algorithm:

Pass 7 of the algorithm:

$found = \{ BA \}; used = \{ \lambda, A, B, a, AB \}.$

$found = \{ \}; used = \{ \lambda, A, B, a, AB, BA \}.$

- (a) $found = \{ \}; used = \{ \lambda, A, B, a, AB, BA \}$
- (b) $E = \{ BAA, BAB, BAa, BAb \}$
 $= \{ BA^2, BAB, BAa, BAb \}$
- (c) $E' = \{ BA, a, B, a \}$
- (d) As *all* the words in E' either appear in '*found*' or in '*used*', this time we *do not add* any words to the '*found*' list.

As the '*found*' list is now *empty*, the algorithm terminates and we can therefore list the elements of $S(3)$: they are the elements in the '*used*' list, namely

- (1) λ
- (2) A
- (3) B
- (4) a
- (5) AB
- (6) BA

This is to be expected as we know that $S(3)$ has **six** elements.

2.6: Knuth-Bendix Rewriting for Induced Actions

2.6.1: Defining an M-set

Given a monoid M , *one* way to construct a monoid M -set X is (as in Example 1.22 on page 8) to let X consist of the elements of M , and to let $x \bullet m$ be the monoid product $x \times m$. By doing this, we can represent the *action* of some generator g of M on X by constructing a transformation in which the elements of X are transformed by multiplying the elements of X on the right by g . To see this 'in action', let us consider the case where M is the symmetric group $S(3)$ again.

Example 2.17:

From our previous work with $S(3)$, we know that the six elements of $S(3)$ are as follows:

- (1) λ
- (2) A
- (3) B
- (4) a
- (5) AB
- (6) BA

Let $M = S(3) = \{\lambda, A, B, a, AB, BA\}$, and let $X = \{1, 2, 3, 4, 5, 6\}$, where $1 = \lambda, 2 = A, 3 = B, 4 = a, 5 = AB$ and $6 = BA$.

Define a monoid M -set by defining $x \bullet m$ ($x \in X, m \in M$) to be the number in X corresponding to the monoid product $x \times m$, where x is the monoid associated to the number $x \in X$. For example, if $x = 3$ and if $m = A$, then $x \bullet m = B \times A = 6$.

If we multiply each of the six elements of X on the right by A , say (where A is a generator of $S(3)$), then after reduction (using the *complete rewrite system* for $S(3)$ on page 15), the elements will have been transformed, for example element (1) above will become element (2) (because $\lambda \times A = A$ is element 2 above). Repeating this analysis for all elements produces the following list:

- (1) $\lambda \times A = A$ = element (2) in the original list
- (2) $A \times A = A^2 \rightarrow a$ = element (4) in the original list
- (3) $B \times A = BA$ = element (6) in the original list
- (4) $a \times A = \lambda$ = element (1) in the original list
- (5) $AB \times A = ABA \rightarrow b \rightarrow B$ = element (3) in the original list
- (6) $BA \times A = BAA \rightarrow Ba \rightarrow AB$ = element (5) in the original list

We can therefore write, for example, $1^A = 2$, or $5^A = 3$, and write down the transformation $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 4 & 6 & 1 & 3 & 5 \end{pmatrix}$ to represent the ‘A-action’. There are similar transformations corresponding to the other generators of M , i.e. we have the ‘a-action’, the ‘B-action’ and the ‘b-action’.

Remark: In the above example, we *numbered* the elements of the monoid $S(3)$ -set X and thus wrote down expressions such as $1^A = 2$ instead of $\lambda^A = A$. The *justification* for doing this is that as long as the calculations involving obtaining the generator actions are correctly handled, then the actual representation of the elements of X is arbitrary and doesn’t matter. The *reason* for numbering the elements is that it is much easier to work with numbers on a computer rather than work with the actual elements themselves — which may represent permutations or matrices, etc.

2.6.2: Induced Monoid Actions

Let us now turn our attention to the theory behind induced monoid actions, the main topic of this project. We start by defining an *equivalence relation* generated by a rewrite system.

Definition 2.18: Consider that we have a monoid M with an associated generating set G . Given a *noetherian confluent* rewrite system R for M , R *generates an equivalence relation (denoted by $\langle\langle R \rangle\rangle$)* as follows:

- Apply the **Knuth-Bendix critical pairs completion algorithm** to R in order to obtain a *complete* rewrite system R' for M .
- Define the equivalence relation $\langle\langle R \rangle\rangle = \sim$ as follows: $u \sim v$ ($u, v \in G^*$) iff u and v have the same normal form with regards to R' .

We can easily verify that the relation \sim is indeed an equivalence relation, and it follows that an **equivalence class** contains all words in G^* which reduce to the same (unique) normal form using R' . But as R' is a *complete* rewrite system, then normal forms will be elements of M , and so an equivalence class will contain all words in G^* which reduce to the same element of M .

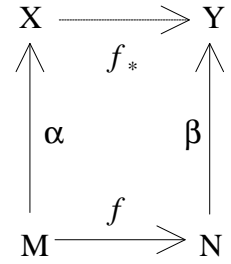
Assuming that the representative of an equivalence class is the normal form that all the members of the equivalence class reduce to using R' (and this will be the representative that we will use from now on by convention), then the set of equivalence class representatives will be *the same as* the set of the elements belonging to M . It follows that if M has m elements, then there will be m equivalence classes.

Remark: Recall that in the Remark following Definition 1.17, we discussed the situation where, given a monoid presentation for a monoid M , $M = \langle X \mid R \rangle$, we thought of the monoid M as being the quotient monoid of the free monoid on the generators X by the equivalence relation generated by the relators R , i.e. $M \cong X^* / \langle\langle R \rangle\rangle$. This now makes sense given the above definition in that we now know that the set of equivalence class representatives for $X^* / \langle\langle R \rangle\rangle$ is ‘*the same as*’ the set of elements of M .

Let us now move on to define what we mean by an *induced monoid action*.

Definition 2.19:

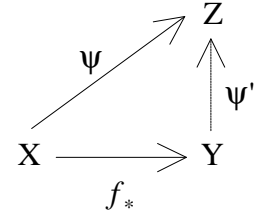
Consider that we are given two monoids M and N , a monoid M -set X , and a monoid morphism $f: M \rightarrow N$. The N -action associated with a monoid N -set Y with the property that f preserves the M -action (so that there exists a function f_* such that $f_*(x^m) = (f_*(x))^{f(m)}$ ($x \in X, m \in M$)) is called the action of N **induced** from that of M by f (it is the *induced monoid action*).



Remark:

(An application of the universal property):

With regards to the situation above in Definition 2.19, if $\psi: X \rightarrow Z$ is another function from the set X to a monoid N -set Z such that the property $\psi(x^m) = (\psi(x))^{f(m)}$ holds, then there exists a *unique* $\psi': Y \rightarrow Z$, a morphism of monoid N -sets, such that $\psi'f_* = \psi$. This defines $f_*: X \rightarrow Y$ uniquely up to *isomorphism*.



To verify the *existence* of a monoid N -set Y that satisfies the above definition, consider the following construction of such a set:

Definition 2.20:

Let a monoid N -set Y be defined as the quotient monoid $Y = X \times N / \langle\langle R \rangle\rangle$, where $X \times N = \{(x, n) : x \in X, n \in N\}$, and $\langle\langle R \rangle\rangle$ is the equivalence relation generated by the following set of rewrite rules on the set $X \times N$: $(x, f(m)n) \rightarrow (x^m, n)$, where $x \in X, m \in M$ and $n \in N$.

Further, let us define the induced monoid action as $[x, n]^{n'} = [x, nn']$ (where $x \in X, n, n' \in N$ and $[x, n]$ is the *equivalence class* of (x, n)), and let us define the associated function f_* as follows: $f_*(x) = [x, \lambda_N]$.

Proposition 2.21:

The monoid N -set Y defined above satisfies Definition 2.19.

Proof:

First of all, we have to verify that the function f_* satisfies $f_*(x^m) = (f_*(x))^{f(m)}$:

LHS = $f_*(x^m) = [x^m, \lambda_N]$;

RHS = $(f_*(x))^{f(m)} = [x, \lambda_N]^{f(m)} = [x, f(m)] \rightarrow [x^m, \lambda_N] = \text{LHS. QED.}$

Secondly, we have to verify that the N -action $[x, n]^{n'} = [x, nn']$ ($x \in X, n, n' \in N$) is valid, i.e. we must show that (i) $[x, n]^{n_1 n_2} = ([x, n]^{n_1})^{n_2}$, and (ii) $[x, n]^\lambda = [x, n]$, where n_1 and n_2 are elements of N , and λ is the identity element of N :

(i) LHS = $[x, n]^{n_1 n_2} = [x, nn_1 n_2] = [x, nn_1]^{n_2} = ([x, n]^{n_1})^{n_2} = \text{RHS};$

(ii) LHS = $[x, n]^\lambda = [x, n\lambda] = [x, n] = \text{RHS. QED.}$

Remark:

Now that we know that there is at least one monoid N -set Y that satisfies Definition 2.19, we can now use the *Universal Property* to say that all other monoid N -sets Y that satisfy Definition 2.19 can be obtained from the monoid N -set defined in Definition 2.20 by a morphism of monoid N -sets.

In Definition 2.20, we constructed a rewrite system for the set $X \times N$ made up of rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$ ($x_1, x_2 \in X; n_1, n_2 \in N$). We cannot use the Knuth-Bendix critical pairs completion algorithm for a free monoid (Algorithm 2.10) *directly* on this rewrite system, as there are **two** kinds of rewriting taking place for an arbitrary word (x, n^*) (where x is an element of the monoid M -set X and n^* is any word in the generators of N): the reduction of words n^* in the generators of N that we perform with a suitable complete rewrite system for N , and the rewriting of elements of $X \times N$ using rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$.

For the rest of this chapter, we shall strive towards creating an algorithm (the ‘Knuth-Bendix critical pairs completion algorithm for induced monoid actions’) that will apply the Knuth-Bendix methods to the above situation, where the input to the algorithm will be an initial rewrite system with rules of the form $(x, f(m)n) \rightarrow (x^m, n)$ ($x \in X, m \in M, n \in N$), and the output of the algorithm will be a complete rewrite system for the elements of the set $X \times N^*$. Providing that the algorithm terminates, we can then go on to use the output obtained to construct the elements (equivalence classes) of the set $Y = X \times N / \langle\langle R \rangle\rangle$.

2.6.3: Induced Monoid Actions: An Example

Consider a monoid morphism f between two monoids M and N , $f: M \rightarrow N$. The monoid morphism can be defined by its effect on the generators of M . For example, if the generators of M are ‘A’ and ‘B’, and if the generators of N are ‘C’, ‘D’ and ‘E’, then the monoid morphism f can be defined in this case by its effect on A and B, e.g. $f(A) = CE$ and $f(B) = D^2EC$. Any word in A’s and B’s can subsequently be written in terms of C’s, D’s and E’s, for example $f(BAB) = D^2ECCED^2EC = D^2EC^2ED^2EC$.

It follows immediately by construction that f is a homomorphism and, if we also define $f(1_M) = 1_N$, then it follows that f is a valid monoid morphism.

Example 2.22: Consider the two symmetric groups $S(3)$ and $S(4)$, with monoid presentations

$$S(3) = \langle A, a, B, b \mid Aa, aA, Bb, bB, A^3, B^2, ABAB \rangle, \text{ and}$$

$$S(4) = \langle C, c, D, d \mid Cc, cC, Dd, dD, C^4, D^3, CDCD \rangle.$$

Let us define a monoid morphism $f: S(3) \rightarrow S(4)$ by its effect on the generators of $S(3)$: $f(\lambda_{S(3)}) = \lambda_{S(4)}, f(A) = D^2, f(B) = C^3DC^2, f(a) = d^2$ and $f(b) = c^2dc^3$.

It follows, for example, that the word A^2ba goes to $D^4c^2dc^3d^2$ under the effect of the monoid morphism.

If we now define an $S(3)$ -action by letting a monoid $S(3)$ -set X consist of the elements of $S(3)$ (as in Example 2.17 on page 18), then we can calculate (as before) transformations to represent each of the generator actions, which can be summarised as follows:

Generator Action	Transformation	Generator Action	Transformation
<i>A-action</i>	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 4 & 6 & 1 & 3 & 5 \end{pmatrix}$	<i>a-action</i>	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 5 & 2 & 6 & 3 \end{pmatrix}$
<i>B-action</i>	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 1 & 6 & 2 & 4 \end{pmatrix}$	<i>b-action</i>	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 1 & 6 & 2 & 4 \end{pmatrix}$

If we now define a set Y as in Definition 2.20, our first task on the road to finding the elements of Y is to construct the set of rewrite rules R that forms part of the definition of Y (according to Definition 2.20, R will be made up of all the rewrite rules of the form $(x, f(m)n) \rightarrow (x^m, n)$, where $x \in X$, $m \in S(3)$ and $n \in S(4)$).

But because of the way that we have *specified* the monoid morphism f , then it is sufficient to only list the rules in R of the type where m is a *generator* of $S(3)$. Further, we can list only the rules in R of the type where $n = \lambda_N$, because each rule of the form $(x, f(m)) \rightarrow (x^m, \lambda_N)$ gives rise to all other rules of the form $(x, f(m)n) \rightarrow (x^m, n)$, where n is an arbitrary element of N .

It follows that the following set of rewrite rules specifies R completely:

$$R = \{ \begin{array}{lll} (1, D^2) \rightarrow (2, \lambda_N), & (2, D^2) \rightarrow (4, \lambda_N), & (3, D^2) \rightarrow (6, \lambda_N), \\ (4, D^2) \rightarrow (1, \lambda_N), & (5, D^2) \rightarrow (3, \lambda_N), & (6, D^2) \rightarrow (5, \lambda_N), \\ (1, C^3DC^2) \rightarrow (3, \lambda_N), & (2, C^3DC^2) \rightarrow (5, \lambda_N), & (3, C^3DC^2) \rightarrow (1, \lambda_N), \\ (4, C^3DC^2) \rightarrow (6, \lambda_N), & (5, C^3DC^2) \rightarrow (2, \lambda_N), & (6, C^3DC^2) \rightarrow (4, \lambda_N), \\ (1, d^2) \rightarrow (4, \lambda_N), & (2, d^2) \rightarrow (1, \lambda_N), & (3, d^2) \rightarrow (5, \lambda_N), \\ (4, d^2) \rightarrow (2, \lambda_N), & (5, d^2) \rightarrow (6, \lambda_N), & (6, d^2) \rightarrow (3, \lambda_N), \\ (1, c^2dc^3) \rightarrow (3, \lambda_N), & (2, c^2dc^3) \rightarrow (5, \lambda_N), & (3, c^2dc^3) \rightarrow (1, \lambda_N), \\ (4, c^2dc^3) \rightarrow (6, \lambda_N), & (5, c^2dc^3) \rightarrow (2, \lambda_N), & (6, c^2dc^3) \rightarrow (4, \lambda_N) \end{array} \}.$$

In order to apply some sort of Knuth-Bendix completion algorithm to the above rewrite system, we first need to define an *admissible well ordering* for the elements of the set $X \times N^*$.

Definition 2.23:

The following is a total order on the set $X \times N^*$, where X is a monoid M -set with numerical representatives, and N^* is the set of all words in the generators of a monoid N :

$$\begin{array}{l} (x'_\beta, n'_\beta) > (x'_\alpha, n'_\alpha) \text{ if } n'_\beta > n'_\alpha \text{ (} x'_\alpha, x'_\beta \in X; n'_\alpha, n'_\beta \in N^* \text{);} \\ (x'_\alpha, n'_\alpha) > (x'_\beta, n'_\beta) \text{ if } n'_\alpha > n'_\beta; \\ (x'_\beta, n'_\beta) > (x'_\alpha, n'_\alpha) \text{ if } n'_\beta = n'_\alpha \text{ and } x'_\beta > x'_\alpha; \\ (x'_\alpha, n'_\alpha) > (x'_\beta, n'_\beta) \text{ if } n'_\alpha = n'_\beta \text{ and } x'_\alpha > x'_\beta; \\ (x'_\beta, n'_\beta) = (x'_\alpha, n'_\alpha) \text{ if } n'_\beta = n'_\alpha \text{ and } x'_\beta = x'_\alpha. \end{array}$$

It is easily verified that the above total order is an *admissible well-ordering* for the elements of the set $X \times N^*$ as long as there is an admissible well-ordering (such as *length-lex*) in place for the elements of the set N^* .

Remark:

Unless otherwise stated, the total order in the above definition will be the total order that we will use for the remainder of this report.

Let us now consider how we would obtain a complete rewrite system for the set $X \times N^*$ by using a complete rewrite system for the set N and an initial rewrite system R for the set $X \times N^*$ like the one shown above, where X is a monoid M -set with numerical representatives, and N^* is the set of *all words* in the generators of a monoid N . Once we have an algorithm to perform such a task, we can then use it to find the equivalence classes which make up the set Y .

2.7: The Knuth-Bendix Critical Pairs Completion Algorithm for Induced Monoid Actions

Algorithm 2.24: (The Knuth-Bendix Critical Pairs Completion Algorithm for Induced Monoid Actions):

- Inputs:*
- (1) A set of *initial rewrite rules* R for a set $X \times N^*$ of the form $(x_1, n_1) \rightarrow (x_2, n_2)$, where x_1 and x_2 are integer representatives of elements belonging to a monoid M -set X , n_1 and n_2 are words in the generators of a monoid N , and R uses an *admissible well-ordering*.
 - (2) A complete rewrite system for N of the form $C_N = \{(l_1 \rightarrow r_1), \dots, (l_n \rightarrow r_n)\}$.

Output: A complete rewrite system for the set $X \times N^*$ with rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$ ($x_1, x_2 \in X$; $n_1, n_2 \in N$), so that any expression of the form (x, n) ($x \in X, n \in N^*$) can be reduced to a unique normal form (x', n') ($x' \in X, n' \in N$).

- Algorithm:*
- (1) Use the complete rewrite system for N to make sure that all the n_i 's in the initial rewrite system R are in normal form, i.e. if any of the n_i 's are *not* in normal form, then use the complete rewrite system for N to reduce the words to normal form.

This leaves us with an initial rewrite system A with rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$, where x_1 and x_2 are integer representatives of elements from the monoid M -set X , and n_1 and n_2 are now elements of the monoid N .

- (2) Let $A = \{a_1, \dots, a_n\}$ be the set of initial rewrite rules obtained above in part (1). For any pair of rules $a_i = (x_{i1}, n_{i1}) \rightarrow (x_{i2}, n_{i2})$ and $a_j = (x_{j1}, n_{j1}) \rightarrow (x_{j2}, n_{j2})$ ($i, j = 1, \dots, n$), look for any *overlaps* between (x_{i1}, n_{i1}) and (x_{j1}, n_{j1}) . If an overlap is found (see below for all the different possible overlaps), reduce the overlap word (x, n) in two ways — first using rule a_i and then using rule a_j — to give two words (x_α, n_α) and (x_β, n_β) . Reduce these words using the rules in A to give a **critical pair** of words (x'_α, n'_α) and (x'_β, n'_β) .

If $(x'_\alpha, n'_\alpha) = (x'_\beta, n'_\beta)$, then do nothing; but if $(x'_\alpha, n'_\alpha) \neq (x'_\beta, n'_\beta)$, then add the rule $((x'_\alpha, n'_\alpha) \rightarrow (x'_\beta, n'_\beta))$ or the rule $((x'_\beta, n'_\beta) \rightarrow (x'_\alpha, n'_\alpha))$ to A , based on the *total order* chosen, i.e. add $((x'_\alpha, n'_\alpha) \rightarrow (x'_\beta, n'_\beta))$ if $(x'_\alpha, n'_\alpha) > (x'_\beta, n'_\beta)$, and add $((x'_\beta, n'_\beta) \rightarrow (x'_\alpha, n'_\alpha))$ if $(x'_\beta, n'_\beta) > (x'_\alpha, n'_\alpha)$.

- (3) For every rule $a_i = (x_{i1}, n_{i1}) \rightarrow (x_{i2}, n_{i2})$ and every rule $c_j = (l_j \rightarrow r_j)$ in C_N , look for any overlaps between n_{i1} and l_j . If an overlap is found (see below for all the different possible overlaps), reduce the overlap word (x, n) in two different ways — first using rule a_i and then using rule c_j — to give two words (x_α, n_α) and (x_β, n_β) . Reduce these words using the rules in A to give a **critical pair** of words (x'_α, n'_α) and (x'_β, n'_β) , and then proceed as we did in part (2).
- (4) Repeat steps (2) and (3) above until no more new rules are added to A during any particular pass of the algorithm. Further, after every pass of the algorithm, analyse the list of rules to make sure that the list doesn't contain any redundant rules, i.e. rules which are implied by the other rules in the list.

Remark: There are *five* different types of overlap word (of two variations), summarised as follows:

Overlap Variation 1		
Type A	Type B	Type C
$(x_{i1}, (\text{---}n_{i1}\text{---}))$	$(x_{i1}, (\text{---}n_{i1}\text{---}))$	$(x_{i1}, (\text{---}n_{i1}\text{---}))$
$(x_{j1}, (\text{---}n_{j1}\text{---}))$	$(x_{j1}, (\text{---}n_{j1}\text{---}))$	$(x_{j1}, (\text{---}n_{j1}\text{---}))$

Overlap Variation 2	
Type D	Type E
$(x_{i1}, (\text{---}n_{i1}\text{---}))$	$(x_{i1}, (\text{---}n_{i1}\text{---}))$
\longleftrightarrow l_j	\longleftrightarrow l_j

Description of overlap types:

Type A: $x_{i1} = x_{j1}$, and n_{i1} appears as a prefix of n_{j1} .

Type B: $x_{i1} = x_{j1}$, and n_{j1} appears as a prefix of n_{i1} .

Type C: $x_{i1} = x_{j1}$, and $n_{j1} = n_{i1}$.

Type D: $l_j = n_{i1}$, or l_j appears inside n_{i1} .

Type E: l_j overlaps on the right with n_{i1} .

Examples of overlap types:

Type A: $a_i = (2, DCB)$, $a_j = (2, DCBDB^2)$.

Type B: $a_i = (4, C^2B)$, $a_j = (4, C^2)$.

Type C: $a_i = (3, CB)$, $a_j = (3, CB)$.

Type D: $a_i = (2, DCB)$, $l_j = DC$.

Type E: $a_i = (2, DCB)$, $l_j = CBD$.

In order to *enhance our understanding* of the above algorithm, let us now consider an example on the use of the algorithm.

2.8: Using the Algorithm in Section 2.7

Example 2.25:

Let us again consider the monoid morphism $f: S(3) \rightarrow S(4)$ defined in Example 2.22 ($f(\lambda_{S(3)}) = \lambda_{S(4)}, f(A) = D^2, f(B) = C^3DC^2, f(a) = d^2$ and $f(b) = c^2dc^3$). Recall that in Example 2.22 we derived the following initial rewrite system for the set $X \times S(4)^*$, where the monoid $S(3)$ -set X is the set of $S(3)$'s elements:

INITIAL REWRITE SYSTEM FOR THE SET $X \times S(4)^*$ ($\lambda = \lambda_{S(4)}$):

$$R = \{ \begin{array}{lll} (1, D^2) \rightarrow (2, \lambda), & (2, D^2) \rightarrow (4, \lambda), & (3, D^2) \rightarrow (6, \lambda), \\ (4, D^2) \rightarrow (1, \lambda), & (5, D^2) \rightarrow (3, \lambda), & (6, D^2) \rightarrow (5, \lambda), \\ (1, C^3DC^2) \rightarrow (3, \lambda), & (2, C^3DC^2) \rightarrow (5, \lambda), & (3, C^3DC^2) \rightarrow (1, \lambda), \\ (4, C^3DC^2) \rightarrow (6, \lambda), & (5, C^3DC^2) \rightarrow (2, \lambda), & (6, C^3DC^2) \rightarrow (4, \lambda), \\ (1, d^2) \rightarrow (4, \lambda), & (2, d^2) \rightarrow (1, \lambda), & (3, d^2) \rightarrow (5, \lambda), \\ (4, d^2) \rightarrow (2, \lambda), & (5, d^2) \rightarrow (6, \lambda), & (6, d^2) \rightarrow (3, \lambda), \\ (1, c^2dc^3) \rightarrow (3, \lambda), & (2, c^2dc^3) \rightarrow (5, \lambda), & (3, c^2dc^3) \rightarrow (1, \lambda), \\ (4, c^2dc^3) \rightarrow (6, \lambda), & (5, c^2dc^3) \rightarrow (2, \lambda), & (6, c^2dc^3) \rightarrow (4, \lambda) \end{array} \}.$$

By applying Algorithm 2.10 to the monoid presentation for $S(4)$ shown in Example 2.22, we can obtain the following complete rewrite system for $S(4)$:

COMPLETE REWRITE SYSTEM FOR $S(4)$:

$$\begin{array}{ll} (1) & cDcD \rightarrow DC^2d \\ (2) & DcDc \rightarrow CDcD \\ (3) & DC^2dC \rightarrow CDcDc \\ (4) & DC^2D \rightarrow cDc \\ (5) & dCd \rightarrow DcD \\ (6) & DCd \rightarrow cD \\ (7) & dCD \rightarrow Dc \\ (8) & dC^2 \rightarrow Cdc \\ (9) & cDC \rightarrow C^2d \\ (10) & dc \rightarrow CD \\ (11) & c^2 \rightarrow C^2 \\ (12) & cd \rightarrow DC \\ (13) & d^2 \rightarrow D \\ (14) & dD \rightarrow \lambda \\ (15) & Dd \rightarrow \lambda \\ (16) & cC \rightarrow \lambda \\ (17) & Cc \rightarrow \lambda \\ (18) & C^3 \rightarrow c \\ (19) & DCD \rightarrow c \\ (20) & CDC \rightarrow d \\ (21) & D^2 \rightarrow d \end{array}$$

Let us now apply Algorithm 2.24 to the above data so that we may obtain a complete rewrite system for the set $X \times S(4)^*$.

We start by checking that each $S(4)$ -word in the initial rewrite system R is in normal form. We do this by using the above complete rewrite system for $S(4)$ to reduce each $S(4)$ -word in R to normal form, and it turns out that all the reductions can be summarised as follows:

$$\begin{array}{l} D^2 \rightarrow d; \\ C^3DC^2 \rightarrow cDC^2 \rightarrow C^2dC; \\ d^2 \rightarrow D; \\ c^2dc^3 \rightarrow C^2dc^3 \rightarrow C^2dC^2c \rightarrow C^2dC. \end{array}$$

Having made all of the necessary reductions, we can say that the 'normal form' of the initial rewrite system R is as follows (see over):

INITIAL REWRITE SYSTEM FOR THE SET $X \times S(4)^*$

IN NORMAL FORM ($\lambda = \lambda_{S(4)}$):

$$R = \{ \begin{array}{lll} (1, d) \rightarrow (2, \lambda), & (2, d) \rightarrow (4, \lambda), & (3, d) \rightarrow (6, \lambda), \\ (4, d) \rightarrow (1, \lambda), & (5, d) \rightarrow (3, \lambda), & (6, d) \rightarrow (5, \lambda), \\ (1, C^2dC) \rightarrow (3, \lambda), & (2, C^2dC) \rightarrow (5, \lambda), & (3, C^2dC) \rightarrow (1, \lambda), \\ (4, C^2dC) \rightarrow (6, \lambda), & (5, C^2dC) \rightarrow (2, \lambda), & (6, C^2dC) \rightarrow (4, \lambda), \\ (1, D) \rightarrow (4, \lambda), & (2, D) \rightarrow (1, \lambda), & (3, D) \rightarrow (5, \lambda), \\ (4, D) \rightarrow (2, \lambda), & (5, D) \rightarrow (6, \lambda), & (6, D) \rightarrow (3, \lambda), \\ (1, C^2dC) \rightarrow (3, \lambda), & (2, C^2dC) \rightarrow (5, \lambda), & (3, C^2dC) \rightarrow (1, \lambda), \\ (4, C^2dC) \rightarrow (6, \lambda), & (5, C^2dC) \rightarrow (2, \lambda), & (6, C^2dC) \rightarrow (4, \lambda) \end{array} \}.$$

We now iterate parts (2) and (3) of the algorithm until no new rules are added to the system during a particular iteration of the algorithm.

To save us from having to refer to a ‘rule from the complete rewrite system for $S(4)$ ’ and a ‘rule from the rewrite system for the induced $S(4)$ -action’ in what follows, let us make the following definition:

Definition 2.26:

Let any rule of the form $(l \rightarrow r)$ be known as a “Type 1” rule, and let any rule of the form $((i_1, w_1) \rightarrow (i_2, w_2))$ be known as a “Type 2” rule.

Starting the iterative part of the algorithm on our data, we see that we must first look for overlaps between all the left hand sides of the ‘Type 2’ rules in the system. There are no such overlaps for our initial system of rewrite rules, and so we go on to look for overlaps between the left hand sides of all the Type 2 rules in the system and the left hand sides of all the Type 1 rules belonging to the complete rewrite system for $S(4)$.

There *are* overlaps of this type (over 100 in all!), the first one found being the overlap between the type 2 rule $(1, d) \rightarrow (2, \lambda)$ and the type 1 rule $dCd \rightarrow DcD$. Reducing the overlap word $(1, dCd)$ using the *type 2 rule* gives the word $(2, Cd)$, whilst reducing the overlap word using the *type 1 rule* gives the word $(1, DcD)$. Reducing these words using R gives us a critical pair of words $(2, Cd)$ and $(4, cD)$, and because these words are different, then we must add the rule $(4, cD) \rightarrow (2, Cd)$ to R (because $(4, cD) > (2, Cd)$ using the total order defined in Definition 2.23, where the total order on N is defined to be $d > c > D > C$).

In all, *39 new type 2 rules* are added to R during the first iteration of the algorithm, so that R then contains 63 type 2 rules. The algorithm now specifies that we must **remove redundant rules** from R , and we do this by using a technique similar to the technique used on page 14: for each rule in R , we try to reduce the left hand side of the rule to the right hand side of the rule using the **other** rules in the system. If we can do this, then we may remove the rule from R — otherwise the rule stays in R .

Applying this method to the 63 type 2 rules in R removes 21 of them, so that after the first iteration of the algorithm, R has a total of 42 type 2 rules. At this stage, we may also go on to look for rule ‘*shortcuts*’, in that if we have two rules $(A, B) \rightarrow (C, D)$ and $(C, D) \rightarrow (E, F)$, then the second rule can **simplify** the first rule, giving us a new pair of rules $(A, B) \rightarrow (E, F)$ and $(C, D) \rightarrow (E, F)$.

The usefulness of the above is that if we had to reduce the word (A, B), say, then we could do it by using **one** rule with the new pair of rules as opposed to having to use **two** rules with the old pair of rules, and thus there is a ‘shortcut’ available when reducing the word (A, B). In the computerised algorithm, we have indeed implemented an algorithm to look for shortcuts of this type.

The algorithm now enters its *second* iteration, where again we look for overlaps of types A to E. It can be shown that **no** overlaps of types A, B or C are found during this second iteration of the algorithm, while around 300 overlaps of types D and E are found. Crucially though, these 300 or so overlaps *do not produce* any **new** rewrite rules (every reduced critical pair has identical elements), so that the algorithm can now terminate, with the rewrite system R (which still has 42 rules) now being a *complete rewrite system* for the set $X \times S(4)^*$ (with the help, of course, of the complete rewrite system for $S(4)$).

COMPLETE REWRITE SYSTEM FOR THE SET $X \times S(4)^*$:

- | | |
|-------------------------------------------------|-------------------------------------------------|
| (1) (3, Cd) \rightarrow (4, C ²) | (22) (5, D) \rightarrow (6, λ) |
| (2) (6, C ² d) \rightarrow (4, c) | (23) (6, D) \rightarrow (3, λ) |
| (3) (6, Cd) \rightarrow (2, C ²) | (24) (4, cD) \rightarrow (6, C ²) |
| (4) (5, C ² d) \rightarrow (2, c) | (25) (2, CD) \rightarrow (4, c) |
| (5) (6, C ² D) \rightarrow (2, C) | (26) (1, CD) \rightarrow (2, c) |
| (6) (2, Cd) \rightarrow (6, C ²) | (27) (1, cD) \rightarrow (3, C ²) |
| (7) (4, C ² d) \rightarrow (6, c) | (28) (4, CD) \rightarrow (1, c) |
| (8) (3, C ² D) \rightarrow (4, C) | (29) (5, cD) \rightarrow (2, C ²) |
| (9) (5, Cd) \rightarrow (1, C ²) | (30) (6, CD) \rightarrow (5, c) |
| (10) (3, C ² d) \rightarrow (1, c) | (31) (3, CD) \rightarrow (6, c) |
| (11) (5, C ² D) \rightarrow (1, C) | (32) (2, cD) \rightarrow (5, C ²) |
| (12) (1, d) \rightarrow (2, λ) | (33) (6, cD) \rightarrow (4, C ²) |
| (13) (2, d) \rightarrow (4, λ) | (34) (5, CD) \rightarrow (3, c) |
| (14) (3, d) \rightarrow (6, λ) | (35) (3, cD) \rightarrow (1, C ²) |
| (15) (4, d) \rightarrow (1, λ) | (36) (4, C ² D) \rightarrow (3, C) |
| (16) (5, d) \rightarrow (3, λ) | (37) (1, C ² d) \rightarrow (3, c) |
| (17) (6, d) \rightarrow (5, λ) | (38) (4, Cd) \rightarrow (3, C ²) |
| (18) (1, D) \rightarrow (4, λ) | (39) (1, C ² D) \rightarrow (5, C) |
| (19) (2, D) \rightarrow (1, λ) | (40) (2, C ² d) \rightarrow (5, c) |
| (20) (3, D) \rightarrow (5, λ) | (41) (1, Cd) \rightarrow (5, C ²) |
| (21) (4, D) \rightarrow (2, λ) | (42) (2, C ² D) \rightarrow (6, C) |

We may now use the above complete rewrite system to reduce an arbitrary word (x, n) , where x is an element of the monoid $S(3)$ -set X and n is any word in the generators of $S(4)$, as follows: *first*, use the complete rewrite system for $S(4)$ to reduce n to its normal form n' , so that we now have a word (x, n') to reduce ($x \in X, n' \in S(4)$), and *then* use the above rewrite system to reduce the word (x, n') to its normal form in the set $X \times S(4)$.

2.9: Obtaining the Elements of the set Y

Now that we have an algorithm for finding the *complete rewrite system* associated with the set $X \times N^*$, so that any word (x, n) ($x \in X, n \in N^*$) may be reduced to a unique element of the set $X \times N$, we may now use the algorithm to find the equivalence classes which make up the set $Y = X \times N / \langle\langle R \rangle\rangle$ as defined earlier. The algorithm is similar to Algorithm 2.15, and can be specified as follows (see over):

Algorithm 2.27:**Obtaining the elements of an induced N-set $Y = X \times N / \langle\langle R \rangle\rangle$.**

Input: A complete rewrite system $C_{X \times N^*}$ for a set $X \times N^*$, where X is a monoid M -set with numerical representatives, and N^* contains all words in the generators of a monoid N .

Output: A list of the elements of the set $X \times N / \langle\langle R \rangle\rangle$, where $\langle\langle R \rangle\rangle$ is the equivalence relation generated by the following set of rewrite rules on the set $X \times N$ (where $x \in X$, $m \in M$, $n \in N$ and $f: M \rightarrow N$ is a monoid morphism): $(x, f(m)n) \rightarrow (x^m, n)$.

Algorithm: Let the list G contain the generators of the monoid N , and let us also set up two other lists called ‘used’ and ‘found’.

To start with, place the equivalence classes $[x, \lambda_N]$ (for all $x \in X$) in the ‘found’ list, and let ‘used’ be the empty set. In order to check that the equivalence classes $[x, \lambda_N]$ are all **distinct**, reduce each word (x, λ_N) separately using $C_{X \times N^*}$ to obtain a normal form (x', n') for the word (x, λ_N) . If (x', n') is of the form (y, λ_N) , where y is an element of X different from x , then remove the equivalence class $[x, \lambda_N]$ from the ‘found’ list (because (x, λ_N) and (y, λ_N) belong to the same equivalence class). This leaves us with an initial set of equivalence classes to work with.

Repeat the following until the ‘found’ list becomes empty (when this happens, the ‘used’ list will then contain all the elements of the set $X \times N / \langle\langle R \rangle\rangle$):

- (a) Take the first equivalence class $[x, n]$ ($x \in X$, $n \in N$) from the ‘found’ list and place it in the ‘used’ list.
- (b) With the action $[x, n]^{n'} = [x, nn']$ ($x \in X$, $n, n' \in N$), act on the equivalence class $[x, n]$ with each element from G to give a list of possible equivalence classes E .
- (c) For each (possible) equivalence class in E , reduce each equivalence class representative using $C_{X \times N^*}$ to give a list of words E' .
- (d) For each word (x', n') in E' ($x' \in X$, $n' \in N$), check to see whether (x', n') is an equivalence class representative either in the ‘used’ list or in the ‘found’ list. If (x', n') is in neither list, then place the equivalence class $[x', n']$ at the end of the ‘found’ list.

Example 2.28:

Let us now finish off the example that we have been considering throughout the duration of Chapter 2 — recall that we are considering a monoid morphism f between the symmetric groups $S(3)$ and $S(4)$, we have an $S(3)$ -set X which is made up of the elements of $S(3)$, and we have found a complete rewrite system for the elements of the set $X \times S(4)^*$. The final stage of the example is to find the equivalence classes that make up the set $X \times N / \langle\langle R \rangle\rangle$, and we can do this by using the above algorithm.

When we apply Algorithm 2.27 to the complete rewrite system $C_{X \times S(4)^*}$ found in Example 2.25 (the one with 42 rewrite rules), the following is the sequence of events that occurs:

We start by placing the equivalence classes $[x, \lambda_{S(4)}]$ (for all $x \in \{1, 2, 3, 4, 5, 6\}$) in the 'found' list, and then check to see whether they are valid, i.e. we try to reduce each word $(x, \lambda_{S(4)})$ using $C_{X \times S(4)^*}$ to see if it reduces to another equivalence class representative. In this instance, all the initial equivalence classes are valid, so that we now start the algorithm proper, with

$found = \{[1, \lambda], [2, \lambda], [3, \lambda], [4, \lambda], [5, \lambda], [6, \lambda]\}$ ($\lambda = \lambda_{S(4)}$),
 $used = \{ \}$, and $G = \{C, D, c, c\}$ (G contains all the generators of $S(4)$).

Pass 1 of the algorithm:

$found = \{[1, \lambda], [2, \lambda], [3, \lambda], [4, \lambda], [5, \lambda], [6, \lambda]\}$, $used = \{ \}$.

- (a) $found = \{[2, \lambda], [3, \lambda], [4, \lambda], [5, \lambda], [6, \lambda]\}$, $used = \{[1, \lambda]\}$.
(Place the first member of the 'found' list in the 'used' list).
- (b) $E = \{[1, \lambda]^c, [1, \lambda]^d, [1, \lambda]^e, [1, \lambda]^f\} = \{[1, C], [1, D], [1, c], [1, d]\}$.
(Act on the equivalence class $[1, \lambda]$ on the right with each element from G to give a list of possible equivalence classes E).
- (c) $E' = \{(1, C), (4, \lambda), (1, c), (2, \lambda)\}$.
(For each (possible) equivalence class in E , reduce each equivalence class representative using $C_{X \times N^}$ to give a list of words E').*
- (d) As the words $(1, C)$ and $(1, c)$ are not equivalence class representatives of elements either in the 'found' list or in the 'used' list, then we place the equivalence classes $[1, C]$ and $[1, c]$ at the end of the 'found' list.
(For each word (x', n') in E' ($x' \in X, n' \in N$), check to see whether (x', n') is an equivalence class representative either in the 'used' list or in the 'found' list. If (x', n') is in neither list, then place the equivalence class $[x', n']$ at the end of the 'found' list).

We now start iteration 2 of the algorithm, with

$found = \{[2, \lambda], [3, \lambda], [4, \lambda], [5, \lambda], [6, \lambda], [1, C], [1, c]\}$, and $used = \{[1, \lambda]\}$.

When the algorithm eventually *terminates* (after 25 iterations), the 'used' list will now contain all the equivalence classes that we want, and they can be listed as follows:

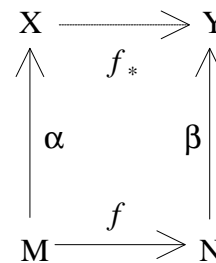
- | | | | | | | | |
|-----|----------------|------|----------|------|----------|------|------------|
| (1) | $[1, \lambda]$ | (7) | $[1, C]$ | (13) | $[4, C]$ | (19) | $[1, C^2]$ |
| (2) | $[2, \lambda]$ | (8) | $[1, c]$ | (14) | $[4, c]$ | (20) | $[5, C^2]$ |
| (3) | $[3, \lambda]$ | (9) | $[2, C]$ | (15) | $[5, C]$ | (21) | $[3, C^2]$ |
| (4) | $[4, \lambda]$ | (10) | $[2, c]$ | (16) | $[5, c]$ | (22) | $[2, C^2]$ |
| (5) | $[5, \lambda]$ | (11) | $[3, C]$ | (17) | $[6, C]$ | (23) | $[6, C^2]$ |
| (6) | $[6, \lambda]$ | (12) | $[3, c]$ | (18) | $[6, c]$ | (24) | $[4, C^2]$ |

This concludes our example in that we now know the elements that make up the set $Y = X \times S(4) / \langle\langle R \rangle\rangle$, so that we can now say which equivalence class contains an arbitrary word (x, n^*) ($x \in X, n^* \in S(4)^*$).

The way we do this is to reduce the arbitrary word (x, n^*) using $C_{X \times S(4)^*}$ to obtain a normal form (y, n) for the word (x, n^*) ($y \in X, n \in S(4)$). But because *normal forms* for the set $X \times S(4)^*$ correspond to equivalence classes in the above list of 24 equivalence classes, this enables us to say that the arbitrary word (x, n^*) belongs to the equivalence class $[y, n]$.

2.10: How much information do we need to find the elements of the set Y?

Recall that we are considering the situation where we have two monoids M and N , a monoid M -set X , and a monoid morphism $f: M \rightarrow N$. Further, the induced monoid N -set Y that we are considering is the set $Y = X \times N / \langle\langle R \rangle\rangle$, where $X \times N = \{(x, n) : x \in X, n \in N\}$, and $\langle\langle R \rangle\rangle$ is the equivalence relation generated by the following set of rewrite rules on the set $X \times N$: $(x, f(m)n) \rightarrow (x^m, n)$ ($x \in X, m \in M, n \in N$). In this section, we will consider the question of how much information do we need to find the elements of the monoid N -set Y .



Our starting point in answering the above question is that in order to find the elements of the monoid N -set Y (using Algorithm 2.27), we must have a complete rewrite system for the set $X \times N^*$ to hand, which we will only have if Algorithm 2.24 terminates with a suitable *complete rewrite system* for the set N and a suitable *initial rewrite system* for the set $X \times N^*$ as input.

Now as discussed in Example 2.22, in order to construct an initial rewrite system for the set $X \times N^*$, if we assume that the monoid morphism f is specified by its effect on the generators of M , then an initial rewrite system for the set $X \times N^*$ need only consist of rules of the form $(x, f(m)) \rightarrow (x^m, \lambda_N)$, where x is an element of the monoid M -set X , and m is a generator of the monoid M . Therefore, in order to build up this initial rewrite system, *first of all* we need to know the images $f(m)$ of each generator of M , which we have already assumed are available; *secondly*, we need to know the size of the monoid M -set X , so that we can know how many integer representatives ‘ x ’ there are; and *finally*, we need to know the effect of the M -action in use, so that we can calculate all expressions of type x^m .

If we have all the above information available, plus a complete rewrite system for the monoid N , then we have enough information to use Algorithm 2.24 to try to obtain a complete rewrite system for the set $X \times N^*$. Notice that we do not need to know anything about M except what its *generators* are, and that the only things that we need to know about X are its *size* and how elements of X are acted upon by generators of M .

Up until now, we have only considered the case where we are given *presentations for M and N* and assume that the monoid M -set X is the set of *elements* of the monoid M , so that we automatically know the size of X and can also calculate the M -action in terms of transformations. In general, this need not be the case, so that we could have situations in which we would only be given the minimum amount of data necessary to apply Algorithm 2.24.

In the next chapter, we will implement **two** different programs that use implementations of Algorithms 2.24 and 2.27. The first program will assume that presentations for M and N are available, that the monoid morphism f is given in terms of its effect on the generators of M , and that the monoid M -set X is the set of elements of M so that we can therefore calculate the M -action in terms of transformations. This program could calculate the elements of the induced monoid N -set Y as we have been doing in the examples in this chapter, culminating in Example 2.28.

The second program will only assume that a presentation for N is available, that the monoid morphism f is given in terms of its effect on the generators of M , and that we are given the effect of the action of all the generators of M on all the elements of the monoid M -set X . This program could calculate the elements of the induced monoid N -set Y where only the *minimum amount of data necessary* was given, and we note that the set X need not be the set of elements of M in this case.

Chapter 3: Implementation

The goal of this project was to implement computationally the algorithms and ideas discussed in the previous chapter. In order to do this, we used the C programming language (ANSI C) together with the *frmon* library of functions.

3.1: A Simple Example on the use of the Library

In order to understand how the library works with the C programming language, let us consider that we want to *multiply two monoid words* taken from the symmetric group $S(3)$ and then display the result on screen. The following is the C source code that enables this to be done:

```
1  /*
2  * File: example1.c
3  * Author: Gareth Evans
4  * Last Modified: 3rd March 2002
5  */
6
7  #include "frmon.h"
8
9  int
10 main(void)
11 {
12     FMon x, y, z;
13
14     x = parseStrToFMon("A*B");
15     y = parseStrToFMon("B*A");
16     z = fMonTimes(x, y);
17
18     printf("(%s)*(%s) = %s", fMonToStr(x), fMonToStr(y), fMonToStr(z));
19 }
```

After some comments in lines 1-5, on line 7 we include the *frmon* library in our piece of code so that we may use the functions and procedures available in the library. The program then starts properly on line 12 where we declare three variables of type *FMon*, where *FMon* is a type declared in the library to store monoid words or elements.

In lines 14-15, we assign values to the variables *x* and *y* by asking the library to convert a string we supply it into a monoid word. We then go on (in line 16) to multiply these two monoid words together (again using a function provided by the library), and finish on line 18 by outputting some information to the screen.

If we were to run the above program (using a terminal in FreeBSD), we would obtain the following single line of output:

```
(A B)*(B A) = A B^2 A
```

3.2: Implementing the Algorithms discussed in Chapter 2

The action plan for implementing the discussed algorithms was to build upon the procedures and functions provided by C and the *frmon* library so that we could write more sophisticated functions and procedures for performing word reduction, for finding a complete rewrite system for a monoid, for finding the elements of a monoid, etc.

Let us now consider how we changed the pseudo-code for the algorithms in Chapter 2 to source code that successfully compiles to give correctly functioning executable programs.

3.2.1: fMonWordReduce

One of the basic functions required when dealing with rewrite systems is the ability to reduce an arbitrary word w using any suitable rewrite system. In the situation where we are given a monoid M that has an associated rewrite system R , in order to reduce an arbitrary word w to normal form using R , the idea is that we keep on cycling through every rule ($l \rightarrow r$) in R looking for occurrences of every l in w until no more reductions are made during any particular cycle through all the rules in R , in which case we can say that the word that we are reducing is in normal form, and thus we may terminate the procedure.

Notice that if we do find an occurrence of some l in w during a cycle of R in the procedure (i.e. we must have $w = u/v$, where u and/or v may be the identity word), then we may replace l by r in w to give a reduced word w' ($w' = urv$), which we then go on to analyse for occurrences of left hand sides of rules in.

The following is the piece of code that we used to implement the process of reducing a word w using a suitable rewrite system R :

```
1  FMon
2  fMonWordReduce( word, rules ) // reduce word as far as possible using rules
3  FMon word;
4  FMonPairList rules;
5  {
6  // Define variables
7  FMon l, r, redword, result;
8  FMonPair rule, div;
9  FMonPairList rws;
10 Short flag;
11 int numrws, pass = 0, num = 1, j = 0, check;
12
13 redword = word; // redword is the 'reduced word'
14 numrws = fMonPairListLength( rules );
15 while ( num > 0 )
16 {
17     num = 0; // num counts the number of substitutions made
18     j = 0; // j loops through the rules, 1 <= j <= numrules
19     pass++; // pass counts the number of passes made
20
21     // Make copy of rules
22     rws = fMonPairListCopy( rules );
23     while ( j < numrws )
24     {
25         j++;
26         l = rws -> lft;
27         r = rws -> rt;
28         rws = rws -> rest;
29
30         // See if the rule simplifies our word
31         check = 0;
32         while ( check == 0 )
33         {
34             num++;
35             result = fMonSubst( redword, l, r );
36             check = fMonEqual( redword, result );
37             redword = result;
38         }
39         num--;
40     } // end while ( j < numrws )
41 } // end while ( num > 0 )
42 return redword;
43 }
44 }
```

The function **fMonWordReduce** has *two* inputs, namely the input ‘word’ of type ‘**FMon**’ which is the monoid word to be reduced, and the input ‘rules’ of type ‘**FMonPairList**’ which is the rewrite system to be used to reduce ‘word’. Notice that an **FMonPairList** is just a (linked) list of **FMonPairs**, which themselves are just pairs of **FMon**s, so that an **FMonPairList** is ideal for storing a list of rules of the form ($l \rightarrow r$), which is exactly what a rewrite system is.

On line 15 of the code we enter a *while loop*, the purpose of which is to execute only if the word that we are reducing has been **changed** during the previous iteration of the while loop. Notice that we will always execute the while loop when we first encounter it, by virtue of the fact that we set the variable ‘*num*’ to store the number 1 on line 11 (the while loop only executes if the variable ‘*num*’ currently holds a value that is greater than zero). The execution of subsequent iterations of the while loop is dependent upon what happens *during* a particular iteration of the while loop.

Inside the while loop, on line 23, we start a *second* while loop, the purpose of which is to cycle through all the rules in the rewrite system whilst looking for a match between the left hand side of a particular rewrite rule and a subword of the word that we are trying to reduce to normal form. This happens on line 35, where we try to *substitute* the left hand side of a rewrite rule into the word that we are trying to reduce to normal form. If a substitution is made, then we (i) try to substitute the left hand side of the same rule into our reduced word again (there may be more than one occurrence of the left hand side of the rule in the word that we are trying to reduce to normal form), and (ii) we increment the variable *num* so that the first ‘outer’ while loop is forced to execute again (other rules which we have previously looked at may now simplify the word that we are now analysing).

After the outer while loop terminates on line 42, we now know that all the rules in the rewrite system have been used to simplify the input word as much as possible, so that the input word must now be in *normal form*. But as the output of the algorithm is required to be the normal form of the input word, we therefore return the normal form of the input word to the piece of code that called the algorithm on line 43, and then we terminate the algorithm.

Example 3.1:

If $w = DC$, and if $rules = \{(D, E)\}$, then the command

```
n = fMonWordReduce(w, rules);
```

places the word EC in the variable ‘*n*’ — because the normal form of the word DC when reducing it using the rewrite system specified by the variable ‘*rules*’ is the word EC.

3.2.2: fMonRulesReduce

In the Knuth-Bendix critical pairs completion algorithm (Algorithm 2.10), after each iteration, the algorithm insists that we must analyse our current list of rewrite rules to make sure that it doesn't contain any **redundant** rules, i.e. rules that are implied by all the other rules in the list. For example, if we had the three rules $(D \rightarrow C)$, $(D \rightarrow E)$ and $(C \rightarrow E)$, then the first rule would be redundant because it wouldn't matter if we applied the rule $D \rightarrow C$ or not — both D and C reduce to the word E using the second and third rules.

The way we decide if a rule is redundant or not is to reduce the left hand side and the right hand side of the rule using the **other** rewrite rules in the system. If the reduced left hand side is equal to the reduced right hand side, then the original rule is redundant, and we may remove it from the list.

Input: A list of rewrite rules K.

Output: K minus the redundant rules from K.

Here is the code that removes all redundant rewrite rules from a list of rewrite rules:

```
1 FMonPairList
2 fMonRulesReduce( K ) // produce an rws list from another using KB
3 FMonPairList K;
4 {
5 // Define variables
6 int d;
7 ULong i = 0, len = 0;
8 FMon pl, pr, ql, qr;
```

```

9   FMonPairList L, L3, L2, L1, L0 = fMonPairListNul;
10  Bool eq;
11
12  len = fMonPairListLength( K );
13  L = fMonPairListRemDups( K ); // remove duplicates (which are obviously redundant)
14  len = fMonPairListLength( L );
15
16  i = 0;
17  L2 = fMonPairListCopy( L );
18  L3 = fMonPairListCopy( L ); // Assume no reductions to begin with
19  while ( L2 )
20  {
21    i++;
22    ql = L2 -> lft;
23    qr = L2 -> rt;
24    L2 = L2 -> rest;
25    // L1 is the list without the current element:
26    L1 = fMonPairListAppend( L0, L2 );
27
28    // Reduce left and right hand sides
29    pl = fMonWordReduce( ql, L1 );
30    pr = fMonWordReduce( qr, L1 );
31    eq = fMonEqual( pl, pr );
32
33    // If LHS = RHS, get rid of element...
34    if ( eq == 1 )
35    {
36      L3 = fMonPairListCopy( L1 );
37    }
38
39    // ...else keep element
40    else
41    {
42      L0 = fMonPairListPush( ql, qr, L0 );
43    }
44  } // end while ( L2 )
45  return L3;
46 }

```

We start the algorithm by removing all *duplicate* rules (which are obviously redundant) from our list of rewrite rules on line 13, by using a command taken from the *frmon* library. We then cycle through all the rules left in the list (using a *while loop* started on line 19), reducing the left hand side and the right hand side of each rule in turn on lines 29 and 30. If the reduced left and right hand sides are the **same** (tested on line 31 using the ‘**fMonEqual**’ function), we then **remove** the rule from the list (on lines 34-37) — otherwise we **keep** the rule in the list (on lines 40-43). After we have gone through all the rules in the list and hence finished the while loop (on line 44), we then return the list of rules with redundant entries removed (on line 45) to the piece of code that called the algorithm.

Example 3.2: If $L = \{(D, C), (D, E), (C, E)\}$, then the command

```
R = fMonRulesReduce( L );
```

will assign the list $\{(D, E), (C, E)\}$ to the variable R.

3.2.3: fMonKnuthBendix

We now come to implementing Algorithm 2.10, the Knuth-Bendix critical pairs completion algorithm. The code that implements the algorithm uses the functions described above in Sections 3.2.1 and 3.2.2, and forms a template for the algorithms that we will encounter later.

Recall that given an initial rewrite system R associated with a monoid M, the purpose of the Knuth-Bendix critical pairs completion algorithm is to find the *complete rewrite system* associated with R.

Let us now consider the code that implements Algorithm 2.10, noting that the source code that follows is a simplified version of the full source code that appears in Appendix B.

```

1 FMonPairList
2 fMonKnuthBendix( L ) // Apply the KB completion algorithm
3 FMonPairList L;
4 {
5 // Define local variables
6 ULong leni, lenj, lenf, lenm, added = 0;
7 Short flag;
8 FMon li, ri, lj, rj, lf, rf, lm, rm, pf, sf,
9 pm, sm, wl, w2, af, bf, qf, middle, left, right;
10 FMonPair pair, div;
11 FMonPairList Li, Lj, crit, crws;
12 int i = 0, j = 0, d = 0, passes = 0;
13
14 crws = fMonPairListCopy( L ); // Holds the complete rewrite system computed so far
15 added = 1;
16
17 while ( added > 0 )
18 {
19 passes++;
20 printf("\nPass number %i:\n", passes);
21 crit = fMonPairListCopy( crws );
22 added = 0; // enable the escape
23 Li = fMonPairListCopy( crws );
24
25 // We now go through Li and Lj, looking for overlaps in each pair of LHS's
26 while ( Li )
27 {
28 Lj = fMonPairListCopy( Li );
29 li = Li -> lft;
30 ri = Li -> rt;
31 leni = fMonLength(li);
32
33 while ( Lj )
34 {
35 lj = Lj -> lft;
36 rj = Lj -> rt;
37 lenj = fMonLength(lj);
38
39 // Decide on the fixed and moving elements
40 if ( leni <= lenj ) // i moving, j fixed
41 {
42 lf = lj; rf = rj; lenf = lenj;
43 lm = li; rm = ri; lenm = leni;
44 }
45 else // j moving, i fixed
46 {
47 lf = li; rf = ri; lenf = leni;
48 lm = lj; rm = rj; lenm = lenj;
49 }
50
51 // We now look for overlaps on the left and right
52 for ( i = 1; i <= lenm-1; i++)
53 {
54 // STAGE 1: Moving word overlaps fixed on the left (RED OVERLAP)
55 //
56 // [.....lf.....]
57 // [...lm...] ->
58
59 sm = fMonSuffix( lm, i );
60 pf = fMonPrefix( lf, i );
61
62 if ( fMonEqual( sm, pf ) == 1 ) // if overlap found
63 {
64 ....
65 // Reduce word in two different ways;
66 // If the reductions are different, add onto the 'crit' list
67 ....
68 } // end if ( fMonEqual( sm, pf ) == 1 )
69
70 // STAGE 3: Moving word overlaps fixed on the right (BLUE OVERLAP)
71 //
72 // [.....lf.....]
73 // <- [...lm...]
74
75 pm = fMonPrefix( lm, i );
76 sf = fMonSuffix( lf, i );
77 if ( fMonEqual( pm, sf ) == 1 ) // if overlap found
78 {
79 ....
80 // Reduce word in two different ways;
81 // If the reductions are different, add onto the 'crit' list
82 ....
83 } // end if ( fMonEqual( pm, sf ) == 1 )
84
85 } // end if ( added > 0 )
86

```

```

87     } // end for ( i = 1; i <= lenm-1; i++)
88
89
90     // STAGE 2: Moving word as substring (PURPLE OVERLAP)
91     //
92     //     [.....lf.....]
93     //     [...lm...] ->
94
95     if (fMonEqual(lm, lf) != 1) // if the left hand sides are not equal
96     {
97         for ( j = 0; j <= lenf-lenm; j++ )
98         {
99             middle = fMonSubWord(lf, j+1, j+lenm);
100             if (fMonEqual(middle, lm) == 1) // if overlap found
101             {
102                 ....
103                 // Reduce word;
104                 // If the reductions are different, add onto the 'crit' list
105                 ....
106             } // end if (fMonEqual...)
107         } // end for (j)
108     } // end if (fMonEqual...)
109
110     Lj = Lj -> rest; // cycle through Lj
111 } // end while ( Lj )
112
113     Li = Li -> rest; // cycle through Li
114 } // end while ( Li )
115
116
117 printf("%i critical pairs added; ",added);
118 if ( added > 0 ) // if critical pairs were added
119 {
120     crws = fMonRulesReduce( crit );
121     printf("%i rules in the reduced set.\n", fMonPairListLength(crws));
122 }
123 else
124 {
125     printf("%i rules in the reduced set.\n\n", fMonPairListLength(crws));
126 }
127 } // end while ( added > 0 )
128 printf("Number of passes made = %i.\n", passes );
129 return crws;
130 }

```

We start (as always) by defining all the *local variables* to be used in the algorithm, and then we begin a while loop on line 17 that will keep on executing as long as a new rule is added to the rewrite system during the execution of the body of the while loop. In the while loop itself, we begin by constructing a mechanism (in lines 26-37) in which we are able to loop through **every** pair of rules in the rewrite system as it currently stands.

After some initialisation in lines 39-49 (in which we decide for a given pair of rules which rule's left hand side is the *longest*), from line 51 onwards we look for overlaps with regards to the left hand sides of the two rules that we are considering. Following the discussion in Section 2.3, we know that there are three types of overlap to be considered *computationally*, and in Section 2.3 we colour coded these three types of overlap.

In the source code above, we attempt to find the '*red*' type of overlap discussed in Section 2.3 in lines 54-69, attempt to find the '*blue*' type of overlap in lines 71-85, and attempt to find the '*purple*' type of overlap in lines 90-109. If any of these types of overlap are detected, then any new rules deduced from these overlaps are added to the rewrite system.

After we have analysed each pair of rules for overlaps, a process which ends on line 115, if any new rules were added to the rewrite system during the just completed iteration of the algorithm, then we apply the **fMonRulesReduce** function (see Section 3.2.2) to the current rewrite system to weed out any *redundant* rules from the list (in lines 118-122), and we then cycle back up to line 17 in order to apply the algorithm to the current rewrite system again. Alternatively, if **no** new rewrite rules were added to the rewrite system during the just completed iteration of the algorithm, then the rewrite system is now **complete** and so we may now terminate the algorithm (but only after returning the complete rewrite system to the piece of code that called the algorithm on line 129).

In the above discussion of the source code, I have not talked about the actual processes involved in finding *overlaps* and *new rewrite rules* computationally. To make up for this omission, let us now consider the processes involved in finding the ‘red’ type of overlap (whose pseudo code is on page 12 of this report) and the subsequent deduction of new rules from overlap words of this type. The following code is the ‘full’ version of the code that appears in lines 54-69 of our previous discussion, i.e. the code that appeared previously in lines 54-69 was a truncated version of the following code:

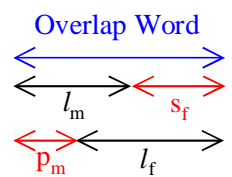
```

1  // STAGE 1: Moving word overlaps fixed on the left
2  //
3  //      [.....lf.....]
4  //  [...lm...] ->
5
6  sm = fMonSuffix( lm, i );
7  pf = fMonPrefix( lf, i );
8
9  if ( fMonEqual( sm, pf ) == 1 ) // if overlap found
10 {
11   pm = fMonPrefix( lm, lenm-i );
12   sf = fMonSuffix( lf, lenf-i );
13
14   // Reduce word in two different ways:
15   w1 = fMonWordReduce( fMonTimes(rm, sf), crws );
16   w2 = fMonWordReduce( fMonTimes(pm, rf), crws );
17
18   // If the reductions are different, add onto list
19   if ( theOrdFun( w1, w2 ) == 1 )
20   {
21     crit = fMonPairListPush( w2, w1, crit );
22     added++;
23   }
24   else if ( theOrdFun( w2, w1 ) == 1 )
25   {
26     crit = fMonPairListPush( w1, w2, crit );
27     added++;
28   }
29 } // end if ( fMonEqual( sm, pf ) == 1 )

```

Recall that for a given pair of rules, in searching for a ‘red’ type of overlap, we are trying to find an overlap between the left hand sides of the rules of the form **suffix(left hand side of rule 1) = prefix(left hand side of rule 2)**. In the above code, this is what is happening in lines 6-9 — we are testing to see if the *prefix* of the left hand side of one rule is equal to the *suffix* of the left hand side of the other rule — for *every possible* suffix and prefix (recall that the above code is in the body of a ‘for’ loop, where the loop runs over the variable ‘i’ which in turn has a maximum value governed by the size of the left hand sides of the rules).

If a match is reported, and so an overlap has been found, we now follow computationally the instructions set out in Algorithm 2.10 regarding what to do when an overlap word is found. In line 15, we reduce the overlap word according to the **first rule** ($l_m \rightarrow r_m$) so that the overlap word reduces to $r_m \times s_f$ ($l_m s_f \rightarrow r_m s_f$). We then reduce the word $r_m s_f$ to normal form according to the current rewrite system in use, so that we obtain a word w_1 . Similarly, in line 16, we reduce the overlap word according to the **second rule** ($l_f \rightarrow r_f$) so that the overlap word reduces to $p_m \times r_f$ ($p_m l_f \rightarrow p_m r_f$). We then reduce the word $p_m r_f$ to normal form according to the current rewrite system in use, so that we obtain a word w_2 .



Now that we have obtained a critical pair of words (w_1, w_2), we can therefore go on (in line 19) to check to see whether it is true that we have $w_2 > w_1$ according to the total order of words in use. If this is the case, then (on line 21) we can add the rule ($w_2 \rightarrow w_1$) to our rewrite system (which is stored as the variable ‘crit’). Similarly, we then go on (in line 24) to check to see whether it is true that we have $w_1 > w_2$ according to the total order of words in use. Again if this is the case, then (on line 26) we can add the rule ($w_1 \rightarrow w_2$) to the rewrite system ‘crit’. Of course, it is evident that if $w_1 = w_2$, then we do nothing (as required by the algorithm).

This completes our discussion regarding what is required to detect ‘red’ overlaps and to subsequently deduce *new rules* from these overlaps. The code for detecting ‘blue’ and ‘purple’ overlaps and then deducing new rules from these overlaps is similar to the above code.

3.2.4: fMonMonoidElements

In order to *implement* Algorithm 2.15, which will (amongst other things) be used in constructing M-actions in later algorithms, the following source code was written. As a reminder, recall that Algorithm 2.15 returns a list of all the elements of some monoid M given a complete rewrite system for M and a list of the generators of M.

Inputs: A complete rewrite system for M; a list of M's generators.

Output: A list of all the *elements* belonging to the monoid M.

```
1  FMonList
2  fMonMonoidElements( gens, crws ) // use complete rws to list elements
3  FMonList gens;
4  FMonPairList crws;
5  {
6  ULong len;
7  Short flag;
8  FMon w, g, wg;
9  FMonList used = fMonListNul, found = fMonListNul, revf, gen2;
10 int i = 0, d = 0, lenu = 0, lenf = 0, inf, inu;
11
12 found = fMonListPush( fMonOne(), found );
13 lenf = 1;
14 while ( lenf > 0 )
15 {
16     revf = fMonListFXRev( found );
17     w = revf -> first;
18     used = fMonListPush( w, used );
19     found = fMonListFXRev( revf -> rest );
20     lenf = lenf-1;
21     lenu = lenu+1;
22     gen2 = fMonListCopy( gens );
23     while ( gen2 )
24     {
25         g = gen2 -> first;
26         gen2 = gen2 -> rest;
27         wg = fMonWordReduce( fMonTimes(w, g), crws );
28         inu = fMonListIsMember( wg, used );
29         inf = fMonListIsMember( wg, found );
30         if ( ( inf == 0 ) && ( inu == 0 ) )
31         {
32             found = fMonListPush( wg, found );
33             lenf = lenf+1;
34         } // end if
35     } // end while ( gen2 )
36 } // end while ( lenf > 0 )
37 return fMonListFXRev( used );
38 } // end FMonMonoidElements
```

The principle that Algorithm 2.15 uses to find all the elements of M is simple: every element of M has to be a *word in the generators of M*. By constructing all possible words in the generators of M modulo the complete rewrite system for M, we can obtain all the monoid elements of M.

In the above source code, we start with the knowledge that the identity word is *always* an element of M, and we therefore place it in a list called '*found*' on line 12 which is a list of elements of M that we find as we go along (this notation is taken directly from Algorithm 2.15). We then start a *while* loop (on line 14) whose purpose is to find *new* monoid elements belonging to M — which we find as follows: for each monoid element $m \in M$ that we found during the *previous* iteration of the while loop (note that if we are executing the while loop for the first time, then we only have the *identity word* to consider), place m in a list called '*used*' (on line 18), and then multiply m on the right by all the monoid generators of M to give a list of words, each of which we reduce on line 27.

If we have not yet encountered a particular reduced word so far (i.e. if it hasn't previously appeared in either the '*found*' list or in the '*used*' list — checked on lines 28-29), then it must be an element of M — and thus we place it in the '*found*' list of elements on line 32. The algorithm terminates when the '*found*' list becomes *empty*, and when this happens, the '*used*' list will now hold *all* of the monoid elements of M, which we return (on line 37) to the piece of code that called the algorithm, as required.

3.2.5: The 'kbe' and 'kba' Programs: Part 1

In order to *use* the fMonKnuthBendix algorithm discussed in Section 3.2.3, we need to be able to give the algorithm some suitable input to process. In order to do this, two programs were created, called 'kbe' and 'kba', that allow the computation of the complete rewrite system associated with a given monoid presentation.

We shall discuss the two programs in further detail in Section 3.2.10 (and there are *two* programs because we shall later define different tasks (concerned with induced monoid actions) for each program to accomplish), but for now let us just say that the programs take *command line parameters* in the form of text files, and giving a single parameter to either of the programs in the form of a text file containing a monoid presentation for a monoid M returns some output including the complete rewrite system associated with the monoid M .

The format of the text file required to specify a monoid presentation for a monoid M can be summarised by the following example concerning the symmetric group $S(3)$.

Example 3.3: The input file associated with the monoid presentation for $S(3)$ shown on page 12 is as follows:

```
A; B; a; b;
A*a ; ;
a*A ; ;
B*b ; ;
b*B ; ;
A^3 ; ;
B^2 ; ;
(A*B)^2 ; ;
```

Notice that the *first line* of the input file specifies the **generators** of the monoid (separated by semicolons); and that the *remaining lines* specify the **relators** of the monoid (which also double up as the initial rewrite system), where a rule of the form $(l \rightarrow r)$ is written as $l; r;$.

The usual mathematical notation when working with a computer is adopted in the input file, such as the use of an *asterisk* to denote multiplication and the use of the 'hat' symbol '^' to denote indices. We shall also by convention use a space to denote the identity word — which, incidentally, is represented as '1' in the computer output and not as 'λ' as we have been discussing.

If we were to run the 'kbe' program with the above presentation for $S(3)$ as input (in the form of the file 's3.in'), then the following would be the output obtained:

```
demo:gareth/kbia> kbe s3.in
Data read in. Now processing...
N is presented as follows:
(C c -> 1)
(c C -> 1)
(D d -> 1)
(d D -> 1)
(C^3 -> 1)
(D^2 -> 1)
(C D C D -> 1)
[7 rule(s)]
```

```

Now computing a complete rewrite system for N....

In fMonKnuthBendix using RemDups:

Pass number 1:
8 critical pairs added; 7 rules in the reduced set.

Pass number 2:
8 critical pairs added; 10 rules in the reduced set.

Pass number 3:
0 critical pairs added; 10 rules in the reduced set.

Number of passes made = 3.

...Complete rewrite system for N computed.

This is the Complete rewrite system for N:
(D c -> C D)
(c^2 -> C)
(c D -> D C)
(D^2 -> 1)
(c C -> 1)
(C c -> 1)
(C^2 -> c)
(D C D -> c)
(C D C -> d)
(d -> D)
[10 rule(s)]

Elements of the monoid M:
1
A
B
a
A B
B A
[6 element(s)]

```

Notice that the elements of $S(3)$ are given at the end of the output — obtained of course with the algorithm discussed in Section 3.2.4.

3.2.6: intFMonListWordReduce

Now that we have seen all the algorithms required to implement the Knuth-Bendix critical pairs completion algorithm *for a free monoid*, we now come to consider the algorithms required to implement Algorithm 2.24, the Knuth-Bendix critical pairs completion algorithm *for induced monoid actions*.

As before, the place to start is to write an algorithm that *reduces* an arbitrary word w as far as it will go using a given rewrite system R . In this context, an arbitrary word w is a pair of the form (x, n) (where x is an element of some monoid M -set X and n is a word in the generators of a monoid N , i.e. $n \in N^*$), and a rewrite system R is a list of rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$ ($x_1, x_2 \in X, n_1, n_2 \in N^*$). We will discuss the algorithm for reducing an arbitrary word (x, n) ($x \in X, n \in N^*$) using a suitable rewrite system after seeing the following simplified version of it, noting first that the algorithm is modified from (and uses) the algorithm in Section 3.2.1.

Inputs: A complete rewrite system for a monoid N (with rules of the form $(l \rightarrow r)$);
A rewrite system R for the set $X \times N^*$ (with rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$);
A word of the form (x, n) to reduce ($x \in X, n \in N^*$).

Output: The normal form of the word (x, n) (with regards to R).

```

1 IntFMonList
2 intFMonListWordReduce( word, rules1, rules2 ) // reduce word using both sets of rules
3 IntFMonList word;
4 FMonPairList rules1; // complete rewrite system for N with rules of the form (l -> r)
5 IntFMonPairList rules2; // rewrite system of the form (x1, n1) -> (x2, n2)
6 {
7 // Define variables
8 IntFMonList back, pass, transfer, new, new2;
9 FMonPairList crws;

```

```

10 IntFMonPairList comparison;
11 FMon origWord, redWord, matchMon, newMon, newmonE;
12 ULong len1, len2;
13 int check = 0, i = 0, j = 0, k = 0, match = 0, match2 = 0, newint = 0;
14 int min = 0;
15
16 // Initialise variables
17 back = intFMonListNul;
18 new = intFMonListNul;
19 crws = fMonPairListCopy( rules1 );
20
21 // Reduce word iteratively
22 transfer = intFMonListCopy( word );
23
24 check = 1; // escape integer
25 while( check != 0 )
26 {
27     check = 0; // to enable the escape!
28     pass = intFMonListCopy( transfer );
29     origWord = pass -> mon;
30     match = pass -> i;
31     i++;
32
33     // Reduce the FMon part of our IntFMon using the
34     // crws passed into the function
35     redWord = fMonWordReduce( origWord, crws );
36
37     new = intFMonListPush(match, redWord, new);
38     transfer = intFMonListCopy( new ); // transfer now holds (match, redWord)
39     new = intFMonListNul;
40
41     comparison = intFMonListCopy( rules2 );
42
43     // Now that we have reduced our word using the Type 1 rules,
44     // we look for simplifications in the Type 2 rules.
45
46     while(comparison) // while there are rules left to compare with
47     {
48         match2 = comparison -> ilft;
49         matchMon = comparison -> mlft;
50
51         if (match == match2) // possible transition (matching integers)
52         {
53             // Look for overlap
54             len1 = fMonLength(redWord);
55             len2 = fMonLength(matchMon);
56
57             if(len2 <= len1) // if list word is <= reduced word
58             {
59                 if (fMonEqual(matchMon, fMonPrefix(redWord, len2)) == 1)
60                 {
61                     // Overlap Found! Now changing (x, pp') to (y, qp')
62                     newint = comparison -> irt; // irt = y
63                     newmonE = comparison -> mrt; // mrt = q
64
65                     if (len1 != len2) // i.e. if p' is not empty
66                     {
67                         newMon = fMonTimes(newmonE, fMonSuffix(redWord, len1-len2));
68                     }
69                     else // i.e. p' is empty
70                     {
71                         newMon = newmonE;
72                     }
73                     check = 1;
74                     new = intFMonListPush(newint, newMon, new);
75
76                     // Escape from loops
77                     transfer = intFMonListCopy( new );
78                     new = intFMonListNul;
79                     comparison = intFMonPairListNul;
80
81                 } // end if (fMonEqual(matchMon, fMonPrefix(redWord, len2)) == 1)
82             } // end if(len2 <= len1)
83             else
84             {
85                 min = 0;
86             }
87             // end if (match == match2)
88             if (check == 0) // only carry on if no match found so far
89             {
90                 comparison = comparison -> rest;
91             }
92         } // end while(comparison)
93     } // end while( check != 0 )
94     return transfer;
95 }

```

The algorithm starts (as usual) by defining all the *variables* to be used in the algorithm. Note that an **IntFMonList** is a variable which stores a (linked) list of expressions of type **IntFMon** (which are variables of the form (x, n)), and that an **IntFMonPairList** is a variable which stores a (linked) list of expressions of type **IntFMonPair** (which are variables of the form $((x_1, n_1), (x_2, n_2))$), used to represent rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$. Note further that we will use the convention that the elements of the monoid M-set X will be stored as *integers* — we will do this in order to simplify the implementation of the algorithm, and the justification for doing this (as discussed earlier) is that we may take *any* representation of the monoid M-set X — it is just a set of elements.

On line 25, we start a *while loop* which will keep on executing until the word (x, n) that we are reducing to normal form did not reduce any further during the previous iteration of the while loop. In the while loop itself, the first thing we do (on line 35) is to reduce the *monoid* part ‘ n ’ of the word (x, n) using the complete rewrite system for the monoid N passed into the algorithm. This leaves us with a word (x, n') to reduce, where n' is the normal form of the word n .

The next stage is to try to reduce the word (x, n') using our collection of type 2 rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$. To do this, for each type 2 rule in the system (we use a while loop started on line 46 to cycle through all the type 2 rules in the system), we look to see if a particular type 2 rule reduces our word (x, n') — for this to be the case, then **two** conditions must be met: first, the integers x and x_1 must be the same, and secondly, n_1 must be a prefix of n' .

We check the **first** condition on line 51 and check the **second** condition on line 59, and if the two conditions are met, then we may reduce our word (x, n') as follows (on lines 61-74): because we now know that the word (x, n') is a word of the form (x_1, n_1n'') , where n'' may or may not be the identity word (checked on line 65), we may therefore reduce the word (x_1, n_1n'') using the type 2 rule $(x_1, n_1) \rightarrow (x_2, n_2)$ to leave us with the word (x_2, n_2n'') .

After the word (x, n) has been reduced as far as it will go using all the type 1 and type 2 rules in the system, on line 94 we then (just before terminating the algorithm) return the reduced form of the word (x, n) to the piece of code that called the algorithm.

3.2.7: intFMonRulesReduce

Recall that in Section 3.2.2 we constructed an algorithm that removed redundant rules from a rewrite system with rules of the form $(l \rightarrow r)$ by removing the rules in the system that had identical *reduced* left and right hand sides when the left and right hand sides were reduced by using all the **other** rewrite rules in the system.

The algorithm in this section implements the above for a rewrite system with rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$. We can modify the algorithm in Section 3.2.2 to accomplish this task, the only major modifications needed being variable changes, e.g. changing FMons to IntFMons. Here is a summary of the important changes that were made to the algorithm in Section 3.2.2:

Old Algorithm (*fMonRulesReduce*)

```
// Remove Duplicates
L = fMonPairListRemDups( K );
```

.....

```
// Reduce left and right hand sides
pl = fMonWordReduce( ql, L1 );
pr = fMonWordReduce( qr, L1 );
```

.....

New Algorithm (*intFMonRulesReduce*)

```
// Remove Duplicates
L = intFMonPairListRemDups( L );
```

.....

```
// Reduce left and right hand sides
pl = intFMonListWordReduce( pl, crwsN, L1, 1 );
r1 = intFMonListWordReduce( r1, crwsN, L1, 1 );
```

.....

```

// If LHS = RHS, get rid of element...
if ( fMonEqual( pl, pr ) == 1 )
{
    L3 = fMonPairListCopy( L1 );
}
// ...else keep element
else
{
    L0 = fMonPairListPush( ql, qr, L0 );
}

// If LHS = RHS, get rid of element....
if ( (fMonEqual( lftM2, rtM2 ) == 1)
      & (lftI2 == rtI2) )
{
    L3 = intFMonPairListCopy( L1 );
}
// ...else keep element
else
{
    L0 = intFMonPairListPush( lftI, lftM,
                              rtI2, rtM2, L0 );
}

```

3.2.8: intFMonKnuthBendix

We have now obtained all of the necessary tools required to implement Algorithm 2.24, the Knuth-Bendix critical pairs completion algorithm for induced monoid actions. Recall that the purpose of the algorithm is to produce a complete rewrite system for the set $X \times N^*$ by using an initial rewrite system for the set $X \times N^*$ and a complete rewrite system for the monoid N (computed using Algorithm 2.10). Let us now consider (a simplified version of) the source code that implements Algorithm 2.24.

Inputs: A complete rewrite system for a monoid N , with rules of the form $(l \rightarrow r)$;
An initial rewrite system for the set $X \times N^*$ (where X is a monoid M -set),
with rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$.

Output: A complete rewrite system for the set $X \times N^*$, with rules of the form
 $(x_1, n_1) \rightarrow (x_2, n_2)$ ($x_1, x_2 \in X, n_1, n_2 \in N$).

```

1  IntFMonPairList
2  intFMonKnuthBendix( crwsN, irws ) // compute crws
3  FMonPairList crwsN;
4  IntFMonPairList irws;
5  {
6  // Define Variables
7  int i, j, k, check, lftIA, lftIB, rtIA, rtIB, count = 0, swapI,
8  redIA, redIB, decide, beginK, endK, executeK, added, il, i2;
9  ULong sizei, sizej, lengthA, lengthB, startSize, endSize, midSize;
10 FMon lftMA = fMonOne, lftMB = fMonOne, rtMA = fMonOne, rtMB = fMonOne,
11 redMA = fMonOne, redMB = fMonOne, m1, m2, swapM = fMonOne;
12 FMonPairList tempN, cycle3;
13 IntFMonPairList tempXW, back, cycle1, cycle2, adder;
14 IntFMonList redA, redB;
15
16 // Check initial list for mistakes
17 irws = intFMonPairListRemDups( intFMonScout( irws ) );
18
19 // Initialise Variables
20 tempXW = intFMonPairListCopy( irws );
21 tempN = fMonPairListCopy( crwsN );
22 back = intFMonPairListCopy( irws );
23 adder = intFMonPairListNul;
24 check = 1;
25
26 while( check != 0 ) // until no more rules added
27 {
28     check = 0; // enable escape
29     added = 0; // reset number of critical pairs added
30     startSize = intFMonPairListLength( tempXW );
31     count++; // used to count the number of iterations
32     printf( "\nIteration %i...\n", count );
33
34     // ***** FIRST STAGE: TYPE 2 WITH TYPE 2 *****
35
36     cycle1 = intFMonPairListCopy( tempXW );
37     sizei = intFMonPairListLength( cycle1 );
38     sizej = sizei;
39     for( i = 1; i <= sizei-1; i++ )
40     {
41         lftIA = cycle1 -> ilft; lftMA = cycle1 -> mlft;
42         rtIA = cycle1 -> irt; rtMA = cycle1 -> mrt;
43
44         // Set up second list - get to correct part of list
45         cycle2 = intFMonPairListCopy( tempXW );
46         for( j = 1; j <= i; j++ )
47         {
48             cycle2 = cycle2 -> rest;
49         }
50
51         for( j = i+1; j <= sizej; j++ )

```

```

53 {
54   lftIB = cycle2 -> ilft;  lftMB = cycle2 -> mlft;
55   rtIB = cycle2 -> irt;   rtMB = cycle2 -> mrt;
56
57   // WE'RE COMPARING (lftIA, lftMA) -> (rtIA, rtMA) with (lftIB, lftMB) -> (rtIB, rtMB)
58   if (lftIA == lftIB) // Check matching integers
59   {
60     // Match! Check if one monoid is a submonoid of the other
61     lengthA = fMonLength(lftMA); lengthB = fMonLength(lftMB);
62
63     if (lengthB < lengthA) // swap elements
64     {
65       swapI = lftIA; lftIA = lftIB; lftIB = swapI;
66       swapI = rtIA; rtIA = rtIB; rtIB = swapI;
67       swapM = lftMA; lftMA = lftMB; lftMB = swapM;
68       swapM = rtMA; rtMA = rtMB; rtMB = swapM;
69       lengthA = fMonLength(lftMA);
70       lengthB = fMonLength(lftMB);
71     }
72
73     if (lengthA != 0)
74     {
75       // Check if lftMA appears in lftMB
76       if (fMonEqual(lftMA, fMonPrefix(lftMB, lengthA)) == 1)
77       {
78         // Match, add onto list if necessary
79         redA = intFMonListNul; redB = intFMonListNul;
80
81         // Reduce word first way
82         redIA = rtIA; redMA = fMonTimes(rtMA, fMonSuffix(lftMB, lengthB-lengthA));
83         // Reduce word second way
84         redIB = rtIB; redMB = rtMB;
85
86         // Simplify reduced words using rewrite system
87         redA = intFMonListPush(redIA, redMA, redA);
88         redA = intFMonListWordReduce( redA, crwsN, tempXW, 1 );
89         redB = intFMonListPush(redIB, redMB, redB);
90         redB = intFMonListWordReduce( redB, crwsN, tempXW, 1 );
91         redIA = redA -> i; redMA = redA -> mon;
92         redIB = redB -> i; redMB = redB -> mon;
93
94         if (intFMonListEqual(redA, redB) != 1) // Now compare words
95         {
96           decide = fMonEqual(redMA, redMB); // See which one is the 'largest'
97
98           // FIRST, decide by the monoids
99           if ( theOrdFun( redMA, redMB ) == 1 )
100          {
101            adder = intFMonPairListNul;
102            adder = intFMonPairListPush(redIB, redMB, redIA, redMA, adder);
103            tempXW = intFMonPairListAppend(tempXW, adder);
104            added++;
105          }
106          else if ( theOrdFun( redMB, redMA ) == 1 )
107          {
108            adder = intFMonPairListNul;
109            adder = intFMonPairListPush(redIA, redMA, redIB, redMB, adder);
110            tempXW = intFMonPairListAppend(tempXW, adder);
111            added++;
112          }
113
114          // SECONDLY, if the monoids are the same, decide by the integers.
115          if (decide == 1)
116          {
117            if (redIA < redIB)
118            {
119              adder = intFMonPairListNul;
120              adder = intFMonPairListPush(redIB, redMB, redIA, redMA, adder);
121              tempXW = intFMonPairListAppend(tempXW, adder);
122              added++;
123            }
124            else if (redIB < redIA)
125            {
126              adder = intFMonPairListNul;
127              adder = intFMonPairListPush(redIA, redMA, redIB, redMB, adder);
128              tempXW = intFMonPairListAppend(tempXW, adder);
129              added++;
130            }
131          } // end if (decide)
132        } // end if (intFMonListEqual...)
133      } // end if (fMonEqual..)
134    } // end if (lengthA != 0)
135  } // end if (IA == IB)
136  cycle2 = cycle2 -> rest;
137 } // end for (j)
138 cycle1 = cycle1 -> rest;

```

```

139     } // end for (i)
140
141     // Tidy Up...
142     tempXW = intFMonRulesReduce( tempXW, crwsN );
143     back = intFMonPairListCopy( tempXW );
144     endSize = intFMonPairListLength( back );
145     midSize = endSize-startSize;
146
147     // ***** SECOND STAGE: TYPE 1 WITH TYPE 2 *****
148
149     cycle1 = intFMonPairListCopy( tempXW ); // with new elements
150     sizei = intFMonPairListLength( cycle1 );
151     for( i = 1; i <= sizei; i++ ) // for each (x, w)
152     {
153         lftIA = cycle1 -> ilft; lftMA = cycle1 -> mlft;
154         rtIA = cycle1 -> irt;  rtMA = cycle1 -> mrt;
155
156         // Set up second list
157         cycle3 = fMonPairListCopy( tempN );
158         sizej = fMonPairListLength( cycle3 );
159
160         for( j = 1; j <= sizej; j++ ) // for each monoid LHS of Type 1 rules
161         {
162             lftMB = cycle3 -> lft; rtMB = cycle3 -> rt;
163
164             // Now look for overlaps.
165             lengthA = fMonLength(lftMA); lengthB = fMonLength(lftMB);
166
167             // Overlap type 1: lftMB inside lftMA
168             if ((lengthB <= lengthA) & (lengthA != 0))
169             {
170                 // Check if lftMB appears somewhere in lftMA
171                 for(k = 1; k <= lengthA-lengthB+1; k++)
172                 {
173                     if(fMonEqual(lftMB, fMonSubWord(lftMA,k,k+lengthB-1)) == 1)
174                     {
175                         // Match found. Now reduce as before
176                         redA = intFMonListNul; redB = intFMonListNul;
177
178                         // Reduce word first way
179                         redIA = rtIA; redMA = rtMA;
180                         // Reduce word second way
181                         redIB = lftIA;
182                         if (k != 1)
183                         {
184                             redMB = fMonTimes(fMonPrefix(lftMA, k-1), rtMB);
185                         }
186                         if (k != lengthA-lengthB+1)
187                         {
188                             redMB = fMonTimes(redMB, fMonSuffix(lftMA, lengthA-lengthB-k+1));
189                         }
190
191                         // Simplify the reduced words using the rewrite system & compare words
192                         // AS CARRIED OUT IN LINES 86-132
193
194                     } // end if fMonEqual...
195                 } // end for (k)
196             } // end if(lengthB <= lengthA)
197
198             // Overlap type 2: lftMB overlaps on right. To begin, set up conditions on loop
199             beginK = 0; endK = lengthA-1; executeK = 0;
200             if (lengthB <= lengthA)
201             {
202                 beginK = 1;
203                 endK = lengthB-1;
204             }
205
206             // Look for overlaps on right hand side
207             for( k = beginK; k <= endK; k++ )
208             {
209                 executeK = 0; // reset switch
210                 if((lengthB <= lengthA) & (lengthA != 0))
211                 {
212                     if(fMonEqual(fMonSuffix(lftMA, k), fMonPrefix(lftMB, k)) == 1)
213                     {
214                         executeK = 1;
215                     }
216                 }
217                 else if (lengthA != 0) // i.e. and lengthB > lengthA
218                 {
219                     if(fMonEqual(fMonSuffix(lftMA, lengthA-k), fMonPrefix(lftMB, lengthA-k)) == 1)
220                     {
221                         executeK = 1;
222                     }
223                 }
224             }

```

```

225     if(executeK == 1)
226     {
227         // Match found. Now reduce as before
228         redA = intFMonListNul; redB = intFMonListNul;
229
230         if (lengthB <= lengthA)
231         {
232             // Reduce word first way
233             redIA = rtIA; redMA = fMonTimes(rtMA, fMonSuffix(lftMB, lengthB-k));
234             // Reduce word second way
235             redIB = lftIA;
236         }
237         else
238         {
239             // Reduce word first way
240             redIA = rtIA; redMA = fMonTimes(rtMA, fMonSuffix(lftMB, lengthB-lengthA+k));
241
242             // Reduce word second way
243             redIB = lftIA; redMB = rtMB;
244             if (k != 0)
245             {
246                 redMB = fMonTimes(fMonPrefix(lftMA, k), redMB);
247             }
248         }
249
250         // Simplify the reduced words using the rewrite system & compare words
251         // AS CARRIED OUT IN LINES 86-132
252
253     } // end if (executeK)
254 } // end for (k)
255 cycle3 = cycle3 -> rest;
256 } // end for (j)
257 cycle1 = cycle1 -> rest;
258 } // end for(i)
259
260 // Get rid of unwanted rules
261 back = intFMonPairListCopy( tempXW );
262 endSize = intFMonPairListLength( back );
263 tempXW = intFMonRulesReduce( tempXW, crwsN );
264 back = intFMonPairListCopy( tempXW );
265
266 // Look for shortcuts
267 back = shortCuts( back );
268
269 endSize = intFMonPairListLength( back );
270 printf("...%i elements in the reduced set\n", endSize);
271
272 // if the size of the list has increased, do the algorithm again
273 if (added > 0)
274 {
275     check = 1;
276 }
277 } // end while
278 return back;
279 } // end IntFMonPairList

```

The first thing to note about the above algorithm is that it *doesn't* implement part (1) of Algorithm 2.24, and the reason for this is that there is no need to implement part (1) of Algorithm 2.24 because we will **only** be using the algorithm *computationally* with initial rewrite systems for the set $X \times N^*$ where all the rules of the form $(x_1, n_1) \rightarrow (x_2, n_2)$ are such that the n_i are **already reduced** to normal form with regards to the complete rewrite system for N (due to the way that the data is processed in the programs that use the computerised version of Algorithm 2.24). The above algorithm does however implement all the *other* parts of Algorithm 2.24, and we shall now look at how it goes about accomplishing this task.

After defining all the variables to be used in the algorithm in lines 6-14, and then checking and initialising some of these variables in lines 16-24, we start the algorithm proper on line 26 by initialising a *while* loop which will keep on executing as long as new rules were added to the system of rewrite rules for the set $X \times N^*$ during the previous iteration of the algorithm.

Inside the while loop, the first task is to implement part (2) of Algorithm 2.24. To do this, we must look for overlaps between the left hand sides of *all the type 2 rules* in the current rewrite system. This process begins on line 39 where, for each type 2 rule in the system, we compare the left hand side of that rule with the left hand sides of all the other type 2 rules in the system (due to the 'for' loop begun on line 52). This means that the code on lines 57-135 executes for every possible (ordered) pairwise combination of left hand sides of type 2 rules.

For a particular pair of rules $A = (lftIA, lftMA) \rightarrow (rtIA, rtMA)$ and $B = (lftIB, lftMB) \rightarrow (rtIB, rtMB)$, the first thing we do in order to look for an overlap is to check to see whether we have $lftIA = lftIB$ or not (done on line 58). If this is **not** the case, then we can go on to compare a *different* pair of rules — but if this **is** the case, then we may continue to look for an overlap, the next condition regarding the presence of an overlap being the requirement that we must have either $lftMA = lftMB$, $lftMA = \text{prefix}(lftMB)$, or $lftMB = \text{prefix}(lftMA)$.

So that we only have to check for **one** kind of prefix, on lines 63-71 we determine which word out of the words $lftMA$ and $lftMB$ is the longest word. We then (on line 76) check to see if the *smaller* word is a prefix of the *larger* word, noting that if the two words have the same length, then line 76 serves to check to see whether it is true that we have $lftMA = lftMB$ or not.

If the test on line 76 is passed, so that we have now *found* an overlap between the left hand sides of two type 2 rules, we now go on to reduce the overlap word using rules A and B to give a pair of words $redA$ and $redB$, where the overlap word is the left hand side of either rule A or rule B, dependent upon which word out of $lftMA$ and $lftMB$ is the longest word. As specified by Algorithm 2.24, we must then simplify this pair of words using the rewrite system for the set $X \times N^*$ at our disposal (on lines 86-92) so that the pair of words $redA$ and $redB$ is now a **critical pair** of words.

The next stage is to decide whether or not our critical pair of words $redA = (redIA, redMA)$ and $redB = (redIB, redMB)$ gives rise to a new rewrite rule for the set $X \times N^*$. In order to do this, we start by following the total order given in Definition 2.23 to check to see (on line 99) whether we have $redMB > redMA$ or not. If it turns out that we do have $redMB > redMA$, then (on lines 101-104) we may add the rule $redB \rightarrow redA$ to our rewrite system. Similarly, if $redMA > redMB$ (checked on line 106), then we may add the rule $redA \rightarrow redB$ to our rewrite system (on lines 108-111), and if $redMA = redMB$, then we may add a new rule to our rewrite system based on which integer is the biggest **integer** out of $redIA$ and $redIB$ (done on lines 114-131), noting that if the integers are the **same**, then we do *not* add a new rule to our rewrite system, as required by the algorithm.

After tidying up the rewrite system we now have after completing stage (2) of Algorithm 2.24 on lines 141-145, we then go on to implement part (3) of Algorithm 2.24, starting on line 147. Here, for every type 2 rule in the system (line 151), we go through every type 1 rule in the complete rewrite system for the monoid N (line 160), and look for overlaps. As we know, there are two different overlaps to look out for (Types ‘D’ and ‘E’ — see page 24), and we have a separate piece of code to detect each type of overlap.

For the overlap type known as ‘Type D’, the task is to find an occurrence of the left hand side of a type 1 rule ($lftMB$) inside the monoid part of the left hand side of a type 2 rule ($lftMA$). We do this on lines 167-196, where the **if loop** started on line 168 checks to see whether $lftMA$ can *fit* inside $lftMB$ (otherwise there is no point in looking for type D overlaps), the **for loop** started on line 171 analyses each potential type D overlap between $lftMA$ and $lftMB$, and the **if loop** started on line 173 checks to see whether a particular potential overlap is a *valid* overlap.

If a valid overlap is found, then we reduce the overlap word $lftMA$ using both the type 1 rule (on line 179) and the type 2 rule (on lines 181-189) to give a pair of words $redA$ and $redB$. We can then go on to obtain a **critical pair** of words corresponding to $redA$ and $redB$ exactly as we did earlier, and we can also use exactly the same method as before to deduce new rewrite rules from this critical pair of words.

For the overlap type known as ‘Type E’, the task is to find an overlap where a prefix of the left hand side of a type 1 rule ($lftMB$) is equal to a suffix of the monoid part of the left hand side of a type 2 rule ($lftMA$). We do this on lines 198-254, where the **for loop** started on line 207 analyses each potential type E overlap between $lftMA$ and $lftMB$, and the **if loops** on lines 210-223 test to see whether each potential overlap is valid.

If a valid overlap is found (and so the variable ‘*executeK*’ has been given the value 1), we then reduce the overlap word using both the type 1 rule (on line 240) and the type 2 rule (on lines 243-247) to give a pair of words *redA* and *redB*. We then go on to obtain a **critical pair** of words corresponding to *redA* and *redB* exactly as we did earlier, and we also use exactly the same method as before to deduce new rewrite rules from this critical pair of words.

The successful completion of all of the above means that we have now completed an iteration of the algorithm. After again tidying up the current rewrite system on lines 260-267 (including removing any *redundant rules* from the system on line 263 and looking for any rule *short cuts* on line 267), we go on to decide whether to start another iteration of the algorithm based upon whether a new rule was added to the rewrite system during the just completed iteration of the algorithm (this check is performed on line 273). If a new rule **was** added to the rewrite system during the just completed iteration of the algorithm, then we cycle back up to line 26 to begin another iteration — otherwise we terminate the algorithm, returning (on line 278) the found complete rewrite system for the set $X \times N^*$ to the piece of code that called the algorithm.

3.2.9: intFMonMonoidElements

In order to *implement* Algorithm 2.27, which returns the list of all the equivalence classes that belong to the induced monoid N-set $Y = X \times N / \langle\langle R \rangle\rangle$, the following code was created. Note that the complete rewrite system we give the algorithm as input is calculated using the algorithm that we have just discussed in Section 3.2.8.

- Inputs:** A complete rewrite system $C_{X \times N^*}$ for a set $X \times N^*$, where X is a monoid M-set with numerical representatives and N^* consists of all words in the generators of a monoid N ;
A complete rewrite system for the monoid N ;
A list N 's generators;
The number of elements in the monoid M-set X .
- Output:** A *list* of all the elements belonging to the set $X \times N / \langle\langle R \rangle\rangle$, where $\langle\langle R \rangle\rangle$ is the equivalence relation generated by the following set of rewrite rules on the set $X \times N$ (where $x \in X$, $m \in M$, $n \in N$ and $f: M \rightarrow N$ is a monoid morphism):
 $(x, f(m)n) \rightarrow (x^m, n)$.

```

1  IntFMonList
2  intFMonMonoidElements( size, gens, crwsN, crwsXW ) // use complete rws to list elements
3  int size;
4  FMonList gens;
5  FMonPairList crwsN;
6  IntFMonPairList crwsXW;
7  {
8      FMon g, monE, monE2, M, M2;
9      FMonList gen2;
10     IntFMonList used = intFMonListNul, found = intFMonListNul, revf,
11         answer = intFMonListNul, L1, L2, L3, L0 = intFMonListNul, check = intFMonListNul;
12     int i = 0, lenu = 0, lenf = 0, inf, inu, intE, intE2, I, I2;
13
14     // Assume that all [x, 1] are elements
15     for(i = 1; i <= size; i++)
16     {
17         found = intFMonListPush( i, fMonOne(), found );
18     }
19
20     // Check that all [x, 1] are indeed elements
21     L2 = intFMonListCopy( found ); // L2 will hold the remaining elements
22     L3 = intFMonListCopy( found ); // L3 will hold the returned list
23                                     // (To begin with, we assume that all elements are valid)
24     while ( L2 ) // while there are elements to analyse
25     {
26         I = L2 -> i;
27         M = L2 -> mon;
28         check = intFMonListPush( I, M, check );
29         L2 = L2 -> rest; // Cycle through L2

```

```

30     L1 = intFMonListAppend( L0, L2 ); // L1 is the list without the current entry:
31
32     // Reduce elements
33     check = intFMonListWordReduce( check, crwsN, crwsXW, 1 );
34     I2 = check -> i;
35     M2 = check -> mon;
36
37     if (intFMonListIsMember( I2, M2, L1 ) == 1)
38     {
39         // 'element' can be reduced to another element - get rid!!
40         L3 = intFMonListCopy( L1 );
41     }
42     else
43     {
44         // keep element
45         L0 = intFMonListPush( I, M, L0 );
46     }
47 } // end while
48
49 // finished checking - now set found to be checked elements
50 found = intFMonListCopy( L3 );
51
52 lenf = intFMonListLength( found );
53 while ( lenf > 0 )
54 {
55     revf = intFMonListFXRev( found );
56     intE = revf -> i;
57     monE = revf -> mon;
58     used = intFMonListPush( intE, monE, used );
59     found = intFMonListFXRev( revf -> rest );
60     lenf = lenf-1;
61     lenu = lenu+1;
62     gen2 = fMonListCopy( gens );
63
64     while ( gen2 )
65     {
66         g = gen2 -> first;
67         gen2 = gen2 -> rest;
68         answer = intFMonListPush( intE, fMonTimes( monE, g ), answer );
69         answer = intFMonListWordReduce( answer, crwsN, crwsXW, 1 );
70         intE2 = answer -> i;
71         monE2 = answer -> mon;
72         inu = intFMonListIsMember( intE2, monE2, used );
73         inf = intFMonListIsMember( intE2, monE2, found );
74         if ( ( inf == 0 ) && ( inu == 0 ) )
75         {
76             found = intFMonListPush( answer -> i, answer -> mon, found );
77             lenf = lenf+1;
78         } // end if
79         answer = intFMonListNul;
80     } // end while ( gen2 )
81 } // end while ( lenf > 0 )
82 return intFMonListFXRev( used );
83 } // end intFMonMonoidElements

```

After defining all the variables to be used in the algorithm on lines 8-12, including the important ‘*used*’ and ‘*found*’ lists (see Algorithm 2.27 for their meanings), we start implementing Algorithm 2.27 on lines 14-18 by assuming that all expressions of the form $[x, \lambda_N]$ (for all $x \in X$) are elements of the set Y , and thus we place them in the ‘*found*’ list on line 17.

The next step is to check that all the elements that we have just placed in the *found* list are **valid** elements of the set Y , and we do this (on line 33) by reducing each word (x, λ_N) separately using the complete rewrite systems ‘*crwsN*’ and ‘*crwsXW*’ passed into the algorithm. If a particular word (x, λ_N) has a *normal form* of the form (y, λ_N) , where $y \in X$ and $y \neq x$, then we *get rid* of the equivalence class $[x, \lambda_N]$ from the *found* list (on line 40) — otherwise we *keep* the equivalence class $[x, \lambda_N]$ in the *found* list (on line 45).

Embarking on the iterative stage of the algorithm, we set up a while loop on line 53 so that the code on lines 55-79 will keep on executing until the *found* list becomes empty. In the while loop itself, the first thing that we do is to take the first equivalence class $[x, n]$ ($x \in X, n \in N$) from the *found* list and place it in the *used* list (on line 58). Notice that this is the part of the algorithm that ensures that the *used* list will always contain all of the elements of the set Y at the end of the algorithm.

In order to cycle through each generator g of N in turn, we set up another while loop on line 64 so that we may (i) **act** on the equivalence class $[x, n]$ on the right by g to give an equivalence class represented by the word ‘*answer*’ (on line 68); (ii) **reduce** (on line 69) the word ‘*answer*’ using the complete rewrite systems for the monoid N and for the set $X \times N^*$ passed into the algorithm; and (iii) place (the reduced form of) the word ‘*answer*’ in the *found* list as an equivalence class (in line 76) if it is true that the word ‘*answer*’ is not the representative of an equivalence class in either the *found* list or in the *used* list (checked on lines 72-74).

Note that the code on lines 64-80 (described in the above paragraph) implements stages (b), (c) and (d) of the iterative part of Algorithm 2.27 (as specified on page 28). Recalling that stage (a) was implemented on line 58, we can therefore say that we have now completed the implementation of the iterative part of the algorithm.

When this iterative part of the algorithm (which spans lines 53-81 of the code) finishes, the whole algorithm finishes, and the *used* list will now hold all of the elements of the induced monoid N -set Y . As this is what the output of the algorithm is required to be, we therefore return the *used* list (on line 82) to the piece of code that called the algorithm.

3.2.10: The ‘kbe’ and ‘kba’ Programs: Part 2

Recall that in Section 2.10 we discussed two different applications of Algorithms 2.24 and 2.27. The first application was to use the algorithms to compute the elements of the induced monoid N -set Y given presentations for M and N , the monoid morphism f in terms of its effect on the generators of M , and the assumption that the monoid M -set X is made up of the elements of M . The second application was to use the algorithms to compute the elements of the induced monoid N -set Y given just the presentation for N , the monoid morphism f in terms of its effect on the generators of M , and the effect of the action of all the generators of M on all the elements of the monoid M -set X .

In order to implement the two applications described above, two programs were created, called ‘*kbe*’ and ‘*kba*’. Let us now discuss how the two programs achieve their goals.

Let us first consider the ‘*kbe*’ program, where the ‘*kb*’ stands for ‘*Knuth-Bendix*’ and the ‘*e*’ stands for ‘*elements*’ — as in we assume that the monoid M -set X is made up of the elements of M . Given presentations for M and N (using the same file format as discussed in Section 3.2.5), plus a text file listing the images under f of the generators of M , our task now is to compute the elements of the induced monoid N -set Y .

The first thing to do after reading in all of the data is to compute a complete rewrite system for the monoid M using the *fMonKnuthBendix* algorithm (which was discussed in Section 3.2.3):

```
// Compute crws for M
printf("\nNow computing a complete rewrite system for M....\n");
crwsM = fMonKnuthBendix( rwsM );
printf("\n...Complete rewrite system for M computed.\n");
```

Once we have computed a complete rewrite system for M , we can then use it to compute the elements of the monoid M (using the *fMonMonoidElements* algorithm discussed in Section 3.2.4):

```
// Compute the elements of M
elts = fMonMonoidElements( M, crwsM );
```

Knowing now how many elements make up the monoid M (we can use the *fMonListLength* function to find this out), we therefore also know how many elements make up the monoid M -set X , as we assume that the monoid M -set X is made up of the elements of M . This then enables us to move on to the next step, which is to construct an initial rewrite system for the set $X \times N^*$.

In Example 2.22, we found out that if we are given the monoid morphism f in terms of its effect on the generators of M , then a valid initial rewrite system for the set $X \times N^*$ is the set consisting of all rewrite rules of the form $(x, f(m)) \rightarrow (x^m, \lambda_N)$, where x is any member of the monoid M -set X , $f(m)$ is the image of any generator m of the monoid M , and x^m is the action of any generator m of the monoid M on any member x of the monoid M -set X .

Now because we have assumed that the monoid M -set X is made up of the elements of M , then the expression x^m is simply the monoid product $x \times m$, which we can reduce using our complete rewrite system for M to give another (possibly different) element of M :

```
answer = fMonWordReduce( fMonTimes(eltsElement, build), crwsM );
```

We now have all the information required in order to build up an initial rewrite system for the set $X \times N^*$ with rewrite rules of the form $(x, f(m)) \rightarrow (x^m, \lambda_N)$, i.e. we know what $|X|$ is, we have been given all the images $f(m)$, and we can calculate all the expressions of type x^m .

```
for(i = 1; i <= nOfGen; i++) // for each generator image
{
    ....
    genElement = fMonWordReduce( genElement, crwsN ); // reduce each f(m) using the crws for N
    for(j = 1; j <= lengthElts; j++) // for each element of X
    {
        ....
        rws2 = intFMonPairListPush( j, genElement, position, empty, rws2 );
        // push ((x, f(m)) -> (x^m, 1) )
    }
}
```

Notice that in the above code we reduce each image $f(m)$ using a complete rewrite system for the monoid N (which we can compute in the same way that we found a complete rewrite system for the monoid M). It follows that all the rewrite rules in the initial rewrite system will be of the form $(x_1, n_1) \rightarrow (x_2, \lambda_N)$, with $x_1, x_2 \in X$, and $n_1 \in N$ — and this is why we didn't implement part (1) of Algorithm 2.24 in our algorithm *intFMonKnuthBendix* discussed in Section 3.2.8 — part (1) of Algorithm 2.24 is effectively implemented here.

Now that we have a valid initial rewrite system for the set $X \times N^*$, we may ask the *intFMonKnuthBendix* algorithm to give us its associated complete rewrite system. Assuming that this process terminates (i.e. assuming that the associated complete rewrite system for the set $X \times N^*$ is finite), we may then go on to ask the *intFMonMonoidElements* algorithm to give us the elements of the induced monoid N -set Y , giving the *intFMonMonoidElements* algorithm as input the complete rewrite system for the set $X \times N^*$ that we have just computed using the *intFMonKnuthBendix* algorithm:

```
// Now compute a crws for elements of the form (x, n)
printf("\nNow computing a complete rewrite system for elements of the form (x, n):\n");
crwsXW = intFMonKnuthBendix( crwsN, rws2 );
printf("\n...Complete rewrite system computed for elements of the form (x, n).\n");

....

// Compute the elements for the induced set
printf("\n-----ELEMENTS-----\n");
eltsIW = intFMonMonoidElements( lengthElts, N, crwsN, crwsXW );
intFMonListDisplay( eltsIW );
```

To demonstrate how the *kbe* program works *in practice*, consider that we ask it to compute the elements of the induced monoid N -set Y for the example that we considered throughout Chapter 2 (recall that we considered the monoid morphism $f: S(3) \rightarrow S(4)$ defined by $f(\lambda_{S(3)}) = \lambda_{S(4)}$, $f(A) = D^2$, $f(B) = C^3DC^2$, $f(a) = d^2$ and $f(b) = c^2dc^3$). The three input files that are required by the program in order to begin its work on this example are as follows (see over):

Monoid Presentation for S(3)
(in file **s3.in**):

```
A; B; a; b;
A*a ; ;
a*A ; ;
B*b ; ;
b*B ; ;
A^3 ; ;
B^2 ; ;
(A*B)^2 ; ;
```

Monoid Presentation for S(4)
(in file **s4.in**):

```
C; D; c; d;
C*c ; ;
c*C ; ;
D*d ; ;
d*D ; ;
C^4 ; ;
D^3 ; ;
(C*D)^2 ; ;
```

Images of the M generators
(in file **edata_s3s4.in**):

```
D^2; C^3*D*C^2; d^2; c^2*d*c^3;
```

Notice that the images above must be given in the order that the generators are specified in the monoid presentation for S(3) on the left, so that the above list is really $f(A); f(B); f(a); f(b)$;

The command that we type in at a terminal to run the *kbe* program on this example is

```
demo:gareth/kbia>kbe s3.in s4.in data.in,
```

and the output that we obtain is as follows:

```
Data read in. Now processing...
Generating set for monoid M =
A
B
a
b
[4 generator(s)]
M is presented as follows:
(A a -> 1)
(a A -> 1)
(B b -> 1)
(b B -> 1)
(A^3 -> 1)
(B^2 -> 1)
(A B A B -> 1)
[7 rule(s)]
Now computing a complete rewrite system for M....
In fMonKnuthBendix using RemDups:
Pass number 1:
8 critical pairs added; 7 rules in the reduced set.
Pass number 2:
8 critical pairs added; 10 rules in the reduced set.
Pass number 3:
0 critical pairs added; 10 rules in the reduced set.
Number of passes made = 3.
...Complete rewrite system for M computed.
This is the Complete rewrite system for M:
(B a -> A B)
(a^2 -> A)
(a B -> B A)
(B^2 -> 1)
(a A -> 1)
(A a -> 1)
(A^2 -> a)
(B A B -> a)
(A B A -> b)
(b -> B)
[10 rule(s)]
Elements of the monoid M:
1
A
B
a
A B
B A
[6 element(s)]

Continuing to process the data...
Generating set for monoid N =
C
D
c
d
[4 generator(s)]
N is presented as follows:
(C c -> 1)
(c C -> 1)
(D d -> 1)
(d D -> 1)
(C^4 -> 1)
(D^3 -> 1)
(C D C D -> 1)
[7 rule(s)]
Now computing a complete rewrite system for N....
In fMonKnuthBendix using RemDups:
Pass number 1:
8 critical pairs added; 8 rules in the reduced set.
Pass number 2:
12 critical pairs added; 16 rules in the reduced set.
Pass number 3:
5 critical pairs added; 18 rules in the reduced set.
Pass number 4:
5 critical pairs added; 21 rules in the reduced set.
Pass number 5:
0 critical pairs added; 21 rules in the reduced set.
Number of passes made = 5.
...Complete rewrite system for N computed.
This is the complete rewrite system for N:
(c D c D -> D C^2 d)
(D c D c -> C D c D)
(D C^2 d C -> C D c D c)
(D C^2 D -> c D c)
(d C d -> D c D)
(D C d -> c D)
(d C D -> D c)
(d C^2 -> C D c)
(c D C -> C^2 d)
(d c -> C D)
(c^2 -> C^2)
(c d -> D C)
(d^2 -> D)
(d D -> 1)
(D d -> 1)
(c C -> 1)
(C c -> 1)
(C^3 -> c)
(D C D -> c)
(C D C -> d)
(D^2 -> d)
[21 rule(s)]
```

Continuing to process the data...

The number of generators in M is 4.

Images of M generators:

D^2
C^3 D C^2
d^2
c^2 d c^3
[4 image(s)]

Type 2 rules for the initial rewrite system:

(1, d) -> (2, 1)
(2, d) -> (4, 1)
(3, d) -> (6, 1)
(4, d) -> (1, 1)
(5, d) -> (3, 1)
(6, d) -> (5, 1)
(1, C^2 d C) -> (3, 1)
(2, C^2 d C) -> (5, 1)
(3, C^2 d C) -> (1, 1)
(4, C^2 d C) -> (6, 1)
(5, C^2 d C) -> (2, 1)
(6, C^2 d C) -> (4, 1)
(1, D) -> (4, 1)
(2, D) -> (1, 1)
(3, D) -> (5, 1)
(4, D) -> (2, 1)
(5, D) -> (6, 1)
(6, D) -> (3, 1)
(1, C^2 d C) -> (3, 1)
(2, C^2 d C) -> (5, 1)
(3, C^2 d C) -> (1, 1)
(4, C^2 d C) -> (6, 1)
(5, C^2 d C) -> (2, 1)
(6, C^2 d C) -> (4, 1)
[24 rule(s)]

Now computing a complete rewrite system for elements of the form (x, n):

Checking the initial rewrite system...
...Initial rewrite system checked.

Iteration 1...

...18 elements to begin with
...0 elements added of Type 2 vs. Type 2
...39 elements added of Type 1 vs. Type 2
...42 elements in the reduced set

Iteration 2...

...42 elements to begin with
...0 elements added of Type 2 vs. Type 2
...0 elements added of Type 1 vs. Type 2
...42 elements in the reduced set

...Complete rewrite system computed for elements of the form (x, n).

The following is the Complete Rewrite System:

-----TYPE 1 RULES-----

(c D c D -> D C^2 d)
(D c D c -> C D c D)
(D C^2 d C -> C D c D c)
(D C^2 D -> c D c)
(d C d -> D c D)
(D C d -> c D)
(d C D -> D c)
(d C^2 -> C D c)
(c D C -> C^2 d)
(d c -> C D)
(c^2 -> C^2)
(c d -> D C)
(d^2 -> D)
(d D -> 1)
(D d -> 1)
(c C -> 1)
(C c -> 1)
(C^3 -> c)
(D C D -> c)
(C D C -> d)
(D^2 -> d)
[21 rule(s)]

-----TYPE 2 RULES-----

(3, C d) -> (4, C^2)
(6, C^2 d) -> (4, c)
(6, C d) -> (2, C^2)
(5, C^2 d) -> (2, c)
(6, C^2 D) -> (2, C)
(2, C d) -> (6, C^2)
(4, C^2 d) -> (6, c)
(3, C^2 D) -> (4, C)
(5, C d) -> (1, C^2)
(3, C^2 d) -> (1, c)
(5, C^2 D) -> (1, C)
(1, d) -> (2, 1)
(2, d) -> (4, 1)
(3, d) -> (6, 1)
(4, d) -> (1, 1)
(5, d) -> (3, 1)
(6, d) -> (5, 1)
(1, D) -> (4, 1)
(2, D) -> (1, 1)
(3, D) -> (5, 1)
(4, D) -> (2, 1)
(5, D) -> (6, 1)
(6, D) -> (3, 1)
(4, c D) -> (6, C^2)
(2, C D) -> (4, c)
(1, C D) -> (2, c)
(1, c D) -> (3, C^2)
(4, C D) -> (1, c)
(5, c D) -> (2, C^2)
(6, C D) -> (5, c)
(3, C D) -> (6, c)
(2, c D) -> (5, C^2)
(6, c D) -> (4, C^2)
(5, C D) -> (3, c)
(3, c D) -> (1, C^2)
(4, C^2 D) -> (3, C)
(1, C^2 d) -> (3, c)
(4, C d) -> (3, C^2)
(1, C^2 D) -> (5, C)
(2, C^2 d) -> (5, c)
(1, C d) -> (5, C^2)
(2, C^2 D) -> (6, C)
[42 rule(s)]

-----ELEMENTS-----

[1, 1]
[2, 1]
[3, 1]
[4, 1]
[5, 1]
[6, 1]
[1, C]
[1, c]
[2, C]
[2, c]
[3, C]
[3, c]
[4, C]
[4, c]
[5, C]
[5, c]
[6, C]
[6, c]
[1, C^2]
[5, C^2]
[3, C^2]
[2, C^2]
[6, C^2]
[4, C^2]
[24 element(s)]

Notice that the above output *matches up exactly* with what was calculated previously in Chapter 2 regarding the complete rewrite system for S(3), the elements of S(3), the complete rewrite system for the induced S(4) action, and the elements of the induced monoid N-set Y.

Let us now turn our attention to the second program, ‘*kba*’, where the ‘*kb*’ stands for ‘*Knuth-Bendix*’ as before and the ‘*a*’ stands for *action* — as in we must give the program as input the effect of the *action* of all the generators of *M* on all the elements of some monoid *M*-set *X*. The other inputs that we must give the program are the presentation for the monoid *N* and the monoid morphism *f* in terms of its effect on the generators of *M* — and our task (once again) is to compute the elements of the induced monoid *N*-set *Y*.

The only place in which the *kba* program *differs* to the *kbe* program is in how it manipulates the input data to obtain an initial rewrite system for the set $X \times N^*$. All other aspects of the program, such as computing the complete rewrite system for the set $X \times N^*$ and subsequently computing the elements of the induced monoid *N*-set *Y*, goes through *exactly* as in the *kbe* program.

So, once we have read in all of the data and computed a complete rewrite system for the monoid *N* as in the *kbe* program, how do we go about constructing an initial rewrite system for the set $X \times N^*$? Well, recalling that we are given the effect of the monoid morphism *f* on each generator of *M* (i.e. we are given all expressions of type $f(m)$), as before the set of rules of the form $(x, f(m)) \rightarrow (x^m, \lambda_N)$ will suffice to build up a valid initial rewrite system for the set $X \times N^*$, where *x* is any member of the monoid *M*-set *X*, $f(m)$ is the image of any generator *m* of the monoid *M*, and x^m is the action of any generator *m* of the monoid *M* on any member *x* of the monoid *M*-set *X*.

The key to constructing an initial rewrite system now lies in the fact that we are *given* the effect of the action of all the generators of *M* on all the elements of the monoid *M*-set *X*. Because of this, we are therefore effectively given all expressions of type x^m (and can therefore deduce the size of the set *X*), and thus our only task is to use the data given to us to build up an initial rewrite system for the set $X \times N^*$, which we do as follows:

```
for(i = 1; i <= nOfGen; i++) // for each generator m of M
{
  build = transferGen -> first;
  build = fMonWordReduce(build, crwsN); // reduce f(m) using the crws for N
  for(j = 1; j <= rowLength; j++) // for all x in X
  {
    // Build up the Pair List rules
    rws2 = intFMonPairListPush(j, build, transferNum -> i, empty, rws2);
    //           push(x, f(m) -> (x^n,          1)
    transferNum = transferNum -> rest;
  }
  transferGen = transferGen -> rest;
}
```

Once the above process of building up a valid initial rewrite system for the set $X \times N^*$ has finished (a rewrite system which again has rules of the form $(x_1, n_1) \rightarrow (x_2, \lambda_N)$, where n_1 is always normalised), we may then proceed as in the *kbe* program.

Let us now consider that we want to use the *kba* program to calculate all the elements of the induced monoid *N*-set *Y* for our now familiar example regarding the monoid morphism $f: S(3) \rightarrow S(4)$ and its associated *M* generator images. This time as input, we must give the program a presentation for *S*(4) and a special data file containing all the other information that the program requires to function, a file whose structure can be summarised as follows:

- Line 1:** The number of generators in *M* (used to simplify the processing of the data);
- Line 2:** The images of the generators of *M* under the monoid morphism *f* (in the order that the generators are given in the presentation file for *M*);
- Remaining Lines:** The effect of the action of all the generators of *M* on all the (*numbered*) elements of the monoid *M*-set *X*, where each line represents the effect of a *particular* generator of *M* on all the elements of the monoid *M*-set *X* — and the lines are ordered so that the first line in this section corresponds to the first image on line 2, the second line in this section corresponds to the second image on line 2, etc.

Here are the *data files* associated with this example:

Monoid Presentation for $S(4)$
(in file **s4.in**):

C; D; c; d;
C*c ; ;
c*C ; ;
D*d ; ;
d*D ; ;
C^4 ; ;
D^3 ; ;
(C*D)^2 ; ;

Other Information, formatted as specified on page 54
(in file **adata_s3s4.in**):

4,
D^2; C^3*D*C^2; d^2; c^2*d*c^3;
2, 4, 6, 1, 3, 5,
3, 5, 1, 6, 2, 4,
4, 1, 5, 2, 6, 3,
3, 5, 1, 6, 2, 4,

Before we consider the *computer output* that we obtain when we apply the *kba* program to this example, let us first consider the origin of the four rows of numbers in the above ‘*other information*’ file. Recall that in Example 2.22 we constructed transformations to represent each of the $S(3)$ generator actions, summarised as follows:

Generator Action	Transformation	Generator Action	Transformation
<i>A-action</i>	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 4 & 6 & 1 & 3 & 5 \end{pmatrix}$	<i>a-action</i>	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 5 & 2 & 6 & 3 \end{pmatrix}$
<i>B-action</i>	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 1 & 6 & 2 & 4 \end{pmatrix}$	<i>b-action</i>	$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 5 & 1 & 6 & 2 & 4 \end{pmatrix}$

We now see that the bottom rows of the transformations correspond to the four rows of numbers in the ‘*other information*’ file, and this corresponds with what the four rows of numbers are *supposed to represent*, i.e. the effect of the action of all the generators of M (which are A , B , a and b in this case) on all the elements of the monoid M -set X (which are the six elements of $S(3)$ in this case). Notice that we must always give the *numerical* transformation representation of the generator actions to the program *kba*, as it can only deal with the numerical representatives of the elements of the monoid M -set X .

The command that we type in at a terminal to run the *kba* program on this example is

```
demo:gareth/kbia>kba s4.in adata_s3s4.in,
```

and the output that we obtain is *identical* to the output that we obtained by using the *kbe* program on the same example previously, except for the fact that the calculations regarding obtaining $S(3)$ ’s complete rewrite system and elements are not shown at the beginning of the output for the *kba* program, the reason being that we do not give the *kba* program **all** the information regarding $S(3)$ — which, incidentally, is the advantage of the *kba* program. It follows that the output that we obtain by using the *kba* program on our example starts as follows:

```
Data read in. Now processing...
```

```
Generating set for monoid N =
```

```
C
D
c
d
[4 generator(s)]
```

```
N is presented as follows:
```

```
(C c -> 1)
(c C -> 1)
(D d -> 1)
(d D -> 1)
```

```
... AS BEFORE WHEN USING THE KBE PROGRAM
```

Chapter 4: Analysis of the Programs

4.1: Using the Programs

Now that we have constructed programs that allow the computation of elements and complete rewrite systems associated with free monoids and induced monoid actions, we can now go on to use the programs to carry out the above tasks, tasks which would otherwise be slow, cumbersome and highly prone to error if done by hand due to their repetitive nature.

As an example of how we may use the programs, let us again consider our by now familiar example where we consider the monoid morphism f between $S(3)$ and $S(4)$, $f: S(3) \rightarrow S(4)$, where the monoid $S(3)$ -set X is made up of the elements of $S(3)$, and $S(3)$ and $S(4)$ are presented as follows:

$$\begin{aligned} S(3) &= \langle A, a, B, b \mid Aa, aA, Bb, bB, A^3, B^2, ABAB \rangle, \\ S(4) &= \langle C, c, D, d \mid Cc, cC, Dd, dD, C^4, D^3, CDCD \rangle. \end{aligned}$$

This time, let us consider what the monoid morphism f can be defined as, remembering that we must define the effect of the monoid morphism on each individual generator of $S(3)$.

By elementary group theory, the kernel of the monoid morphism f must be a normal subgroup of $S(3)$, and so is either the identity group I , the cyclic group $C(3)$, or the symmetric group $S(3)$ itself (as these are the only normal subgroups of $S(3)$). By the first isomorphism theorem, we deduce that the image of f is $S(3)/I = S(3)$ if $\ker(f) = I$; that $\text{Im}(f) = S(3)/C(3) = C(2)$ if $\ker(f) = C(3)$; and that $\text{Im}(f) = S(3)/S(3) = I$ if $\ker(f) = S(3)$.

Taking the case $\text{Ker}(f) = I$ (and so $\text{Im}(f) = S(3)$) first, we note that because $S(4)$ contains exactly four distinct subgroups isomorphic to $S(3)$, and because each of these subgroups has six automorphisms, then there are precisely 24 choices for f in this instance. One such choice for f is the one that we have been considering throughout this report, i.e. $f(\lambda_{S(3)}) = \lambda_{S(4)}$, $f(A) = D^2$, $f(B) = C^3DC^2$, $f(a) = d^2$ and $f(b) = c^2dc^3$.

In order to verify that $\text{Ker}(f) = I$ in this instance, we used the fact that the six elements of $S(3)$ are $\lambda_{S(3)}$, A , B , a , AB and BA in order to deduce that the images of the six elements of $S(3)$ under the above monoid morphism f are $\lambda_{S(4)}$, D^2 , C^3DC^2 , d^2 , $D^2C^3DC^2$ and $C^3DC^2D^2$ respectively. Reducing these images using the complete rewrite system for $S(4)$ (shown on page 25) gives us a list of reduced images $\lambda_{S(4)}$, d , c^2dC , D , C^2Dc and C^2DcD , and because all but one of these reduced images are non-identity elements of $S(4)$, then we must conclude that $\ker(f) = I$ as required.

Further, in order to verify that $\text{Im}(f) = S(3)$ in this instance, we note that as elements of $S(3)$ map onto elements in $S(4)$ with the same order (e.g. $A \in S(3)$ with order 3 in $S(3)$ maps onto $d \in S(4)$ with order 3 in $S(4)$), and as it can be shown that the set of reduced images $\{\lambda_{S(4)}, d, c^2dC, D, C^2Dc, C^2DcD\}$ forms a subgroup of $S(4)$ isomorphic to $S(3)$, then f must be an isomorphism between $S(3)$ and the subgroup $\{\lambda_{S(4)}, d, c^2dC, D, C^2Dc, C^2DcD\}$ of $S(4)$, so that $\text{Im}(f) = S(3)$ as required.

Now we have already applied the *kbe* program to the above example in Section 3.2.10, and we found out there that the induced monoid N -set Y was made up of the following 24 equivalence classes:

- | | | | |
|---------------------------|---------------|---------------|-----------------|
| (1) $[1, \lambda_{S(4)}]$ | (7) $[1, C]$ | (13) $[1, c]$ | (19) $[1, C^2]$ |
| (2) $[2, \lambda_{S(4)}]$ | (8) $[2, C]$ | (14) $[2, c]$ | (20) $[2, C^2]$ |
| (3) $[3, \lambda_{S(4)}]$ | (9) $[3, C]$ | (15) $[3, c]$ | (21) $[3, C^2]$ |
| (4) $[4, \lambda_{S(4)}]$ | (10) $[4, C]$ | (16) $[4, c]$ | (22) $[4, C^2]$ |
| (5) $[5, \lambda_{S(4)}]$ | (11) $[5, C]$ | (17) $[5, c]$ | (23) $[5, C^2]$ |
| (6) $[6, \lambda_{S(4)}]$ | (12) $[6, C]$ | (18) $[6, c]$ | (24) $[6, C^2]$ |

Now in our example, because $X = \{1, 2, 3, 4, 5, 6\}$ corresponds to $S(3) = \{\lambda_{S(3)}, A, B, a, AB, BA\}$, and because $S(4) = \{\lambda_{S(4)}, C, D, c, d, C^2, CD, Cd, DC, Dc, cD, dC, C^2D, c^2d, CDc, CdC, DC^2, DcD, cDc, C^2Dc, C^2dC, CDcD, DC^2d, C^2DcD\}$, then it follows that the set $X \times S(4)$ has $6 \times 24 = 144$ elements, each of which belongs to one of the 24 equivalence classes shown on the previous page. In order to see which equivalence class contains an arbitrary element from the set $X \times S(4)$, we can use the command

```
kbe s3.in s4.in edata_s3s4.in word.in
```

in a suitable terminal, where `word.in` contains a representation of an arbitrary element from the set $X \times S(4)$. The following is a summary of which elements of the set $X \times S(4)$ belong to each equivalence class of the set Y , obtained by repeatedly using the above command on each element of the set $X \times S(4)$.

Equivalence Class:	Members of the Equivalence Class:
[1, $\lambda_{S(4)}$]	(1, $\lambda_{S(4)}$), (2, D), (4, d), (5, C^2Dc), (3, C^2dC), (6, C^2DcD)
[2, $\lambda_{S(4)}$]	(2, $\lambda_{S(4)}$), (4, D), (1, d), (6, C^2Dc), (5, C^2dC), (3, C^2DcD)
[3, $\lambda_{S(4)}$]	(3, $\lambda_{S(4)}$), (6, D), (5, d), (4, C^2Dc), (1, C^2dC), (2, C^2DcD)
[4, $\lambda_{S(4)}$]	(4, $\lambda_{S(4)}$), (1, D), (2, d), (3, C^2Dc), (6, C^2dC), (5, C^2DcD)
[5, $\lambda_{S(4)}$]	(5, $\lambda_{S(4)}$), (3, D), (6, d), (1, C^2Dc), (2, C^2dC), (4, C^2DcD)
[6, $\lambda_{S(4)}$]	(6, $\lambda_{S(4)}$), (5, D), (3, d), (2, C^2Dc), (4, C^2dC), (1, C^2DcD)
[1, C]	(1, C), (2, DC), (4, dC), (5, C^2D), (3, cDc), (6, CDcD),
[2, C]	(2, C), (4, DC), (1, dC), (6, C^2D), (5, cDc), (3, CDcD)
[3, C]	(3, C), (6, DC), (5, dC), (4, C^2D), (1, cDc), (2, CDcD)
[4, C]	(4, C), (1, DC), (2, dC), (3, C^2D), (6, cDc), (5, CDcD)
[5, C]	(5, C), (3, DC), (6, dC), (1, C^2D), (2, cDc), (4, CDcD)
[6, C]	(6, C), (5, DC), (3, dC), (2, C^2D), (4, cDc), (1, CDcD)
[1, c]	(1, c), (4, CD), (2, Dc), (3, c^2d), (5, CdC), (6, DC^2d)
[2, c]	(2, c), (1, CD), (4, Dc), (5, c^2d), (6, CdC), (3, DC^2d)
[3, c]	(3, c), (5, CD), (6, Dc), (1, c^2d), (4, CdC), (2, DC^2d)
[4, c]	(4, c), (2, CD), (1, Dc), (6, c^2d), (3, CdC), (5, DC^2d)
[5, c]	(5, c), (6, CD), (3, Dc), (2, c^2d), (1, CdC), (4, DC^2d)
[6, c]	(6, c), (3, CD), (5, Dc), (4, c^2d), (2, CdC), (1, DC^2d)
[1, C^2]	(1, C^2), (5, Cd), (3, cD), (4, CDc), (2, DC^2), (6, DcD)
[2, C^2]	(2, C^2), (6, Cd), (5, cD), (1, CDc), (4, DC^2), (3, DcD)
[3, C^2]	(3, C^2), (4, Cd), (1, cD), (5, CDc), (6, DC^2), (2, DcD)
[4, C^2]	(4, C^2), (3, Cd), (6, cD), (2, CDc), (1, DC^2), (5, DcD)
[5, C^2]	(5, C^2), (1, Cd), (2, cD), (6, CDc), (3, DC^2), (4, DcD)
[6, C^2]	(6, C^2), (2, Cd), (4, cD), (3, CDc), (5, DC^2), (1, DcD)

As you can see, each equivalence class contains *six* members from the set $X \times S(4)$.

Let us now consider the case where $\ker(f) = C(3)$ by defining the monoid morphism f as follows: $f(\lambda_{S(3)}) = \lambda_{S(4)}$, $f(A) = \lambda_{S(4)}$, $f(B) = C^2$, $f(a) = \lambda_{S(4)}$ and $f(b) = c^2$. In order to verify that $\text{Ker}(f) = C(3)$ in this instance, we note that as the images under f of the six elements of $S(3) = \{\lambda_{S(3)}, A, B, a, AB, BA\}$ are $\lambda_{S(4)}$, $\lambda_{S(4)}$, C^2 , $\lambda_{S(4)}$, C^2 and C^2 respectively, and as these images *do not reduce any further* using the complete rewrite system for $S(4)$, then it follows that we must conclude that $\ker(f) = \{\lambda_{S(3)}, A, a\}$, a subgroup of $S(4)$ which is isomorphic to $C(3)$, thus giving us the required conclusion. Further, as C^2 has order 2 in $S(4)$ (and is therefore self-inverse), it is a trivial task to show that $\text{Im}(f) = C(2)$ for the chosen monoid morphism f .

As an aside, note that because $S(4)$ has *nine* elements of order 2, then it follows that there are nine choices for f in order to obtain $\ker(f) = C(3)$ and $\text{Im}(f) = C(2)$ — just define $f(\lambda_{S(3)}) = \lambda_{S(4)}$, $f(A) = \lambda_{S(4)}$, $f(B) = \alpha$, $f(a) = \lambda_{S(4)}$ and $f(b) = \alpha$, making sure that α is an element from $S(4)$ of **order 2**.

Applying the *kbe* program to the example on the bottom of the previous page yields the following complete rewrite system and equivalence classes:

Complete Rewrite System for the set $X \times S(4)^*$:

- | | | | |
|-----|-------------------------------------------------------|------|-------------------------------------------------------|
| (1) | $(3, DC) \rightarrow (1, Cd)$ | (6) | $(1, C^2) \rightarrow (3, \lambda_{S(4)})$ |
| (2) | $(1, DC) \rightarrow (3, Cd)$ | (7) | $(5, \lambda_{S(4)}) \rightarrow (3, \lambda_{S(4)})$ |
| (3) | $(3, c) \rightarrow (1, C)$ | (8) | $(2, \lambda_{S(4)}) \rightarrow (1, \lambda_{S(4)})$ |
| (4) | $(1, c) \rightarrow (3, C)$ | (9) | $(4, \lambda_{S(4)}) \rightarrow (1, \lambda_{S(4)})$ |
| (5) | $(6, \lambda_{S(4)}) \rightarrow (3, \lambda_{S(4)})$ | (10) | $(3, C^2) \rightarrow (1, \lambda_{S(4)})$ |

Equivalence Class:

Members of the Equivalence Class:

- | | |
|------------------------|---------------------------------------------------------------------------------------------------------------|
| [1, $\lambda_{S(4)}$] | (1, $\lambda_{S(4)}$), (2, $\lambda_{S(4)}$), (4, $\lambda_{S(4)}$), (3, C^2), (5, C^2), (6, C) |
| [3, $\lambda_{S(4)}$] | (3, $\lambda_{S(4)}$), (5, $\lambda_{S(4)}$), (6, $\lambda_{S(4)}$), (1, C^2), (2, C^2), (4, C^2) |
| [1, C] | (1, C), (2, C), (4, C), (3, c), (5, c), (6, c) |
| [3, C] | (3, C), (5, C), (6, C), (1, c), (2, c), (4, c) |
| [1, D] | (1, D), (2, D), (4, D), (3, C^2D), (5, C^2D), (6, C^2D) |
| [3, D] | (3, D), (5, D), (6, D), (1, C^2D), (2, C^2D), (4, C^2D) |
| [1, d] | (1, d), (2, d), (4, d), (3, c^2d), (5, c^2d), (6, c^2d) |
| [3, d] | (3, d), (5, d), (6, d), (1, c^2d), (2, c^2d), (4, c^2d) |
| [1, CD] | (1, CD), (2, CD), (4, CD), (3, cD), (5, cD), (6, cD) |
| [3, CD] | (3, CD), (5, CD), (6, CD), (1, cD), (2, cD), (4, cD) |
| [1, Cd] | (1, Cd), (2, Cd), (4, Cd), (3, DC), (5, DC), (6, DC) |
| [3, Cd] | (3, Cd), (5, Cd), (6, Cd), (1, DC), (2, DC), (4, DC) |
| [1, Dc] | (1, Dc), (2, Dc), (4, Dc), (3, C^2Dc), (5, C^2Dc), (6, C^2Dc) |
| [3, Dc] | (3, Dc), (5, Dc), (6, Dc), (1, C^2Dc), (2, C^2Dc), (4, C^2Dc) |
| [1, dC] | (1, dC), (2, dC), (4, dC), (3, C^2dC), (5, C^2dC), (6, C^2dC) |
| [3, dC] | (3, dC), (5, dC), (6, dC), (1, C^2dC), (2, C^2dC), (4, C^2dC) |
| [1, CDc] | (1, CDc), (2, CDc), (4, CDc), (3, cDc), (5, cDc), (6, cDc) |
| [3, CDc] | (3, CDc), (5, CDc), (6, CDc), (1, cDc), (2, cDc), (4, cDc) |
| [1, CdC] | (1, CdC), (2, CdC), (4, CdC), (3, DC^2), (5, DC^2), (6, DC^2) |
| [3, CdC] | (3, CdC), (5, CdC), (6, CdC), (1, DC^2), (2, DC^2), (4, DC^2) |
| [1, DcD] | (1, DcD), (2, DcD), (4, DcD), (3, C^2DcD), (5, C^2DcD), (6, C^2DcD) |
| [3, DcD] | (3, DcD), (5, DcD), (6, DcD), (1, C^2DcD), (2, C^2DcD), (4, C^2DcD) |
| [1, CDcD] | (1, CDcD), (2, CDcD), (4, CDcD), (3, DC^2d), (5, DC^2d), (6, DC^2d) |
| [3, CDcD] | (3, CDcD), (5, CDcD), (6, CDcD), (1, DC^2d), (2, DC^2d), (4, DC^2d) |

As before, we have 24 equivalence classes, each of which contains *six* members from the set $X \times S(4)$ — but notice this time that the equivalence classes are different and are full of interesting patterns.

To finish, let us consider the case where $\ker(f) = S(3)$. In this situation, there is only one way to define the monoid morphism f , namely $f(\lambda_{S(3)}) = \lambda_{S(4)}$, $f(A) = \lambda_{S(4)}$, $f(B) = \lambda_{S(4)}$, $f(a) = \lambda_{S(4)}$ and $f(b) = \lambda_{S(4)}$. It follows easily that $\text{Im}(f) = I$, and applying the *kbe* program to this situation yields the following complete rewrite system and equivalence classes:

Complete Rewrite System for the set $X \times S(4)^*$:

- | | | | |
|-----|-------------------------------------------------------|-----|-------------------------------------------------------|
| (1) | $(2, \lambda_{S(4)}) \rightarrow (1, \lambda_{S(4)})$ | (4) | $(5, \lambda_{S(4)}) \rightarrow (1, \lambda_{S(4)})$ |
| (2) | $(3, \lambda_{S(4)}) \rightarrow (1, \lambda_{S(4)})$ | (5) | $(6, \lambda_{S(4)}) \rightarrow (1, \lambda_{S(4)})$ |
| (3) | $(4, \lambda_{S(4)}) \rightarrow (1, \lambda_{S(4)})$ | | |

Equivalence Class:**Members of the Equivalence Class:**

[1, $\lambda_{S(4)}$]	(1, $\lambda_{S(4)}$), (2, $\lambda_{S(4)}$), (3, $\lambda_{S(4)}$), (4, $\lambda_{S(4)}$), (5, $\lambda_{S(4)}$), (6, $\lambda_{S(4)}$)
[1, C]	(1, C), (2, C), (3, C), (4, C), (5, C), (6, C)
[1, D]	(1, D), (2, D), (3, D), (4, D), (5, D), (6, D)
[1, c]	(1, c), (2, c), (3, c), (4, c), (5, c), (6, c)
[1, d]	(1, d), (2, d), (3, d), (4, d), (5, d), (6, d)
[1, C ²]	(1, C ²), (2, C ²), (3, C ²), (4, C ²), (5, C ²), (6, C ²)
[1, CD]	(1, CD), (2, CD), (3, CD), (4, CD), (5, CD), (6, CD)
[1, Cd]	(1, Cd), (2, Cd), (3, Cd), (4, Cd), (5, Cd), (6, Cd)
[1, DC]	(1, DC), (2, DC), (3, DC), (4, DC), (5, DC), (6, DC)
[1, Dc]	(1, Dc), (2, Dc), (3, Dc), (4, Dc), (5, Dc), (6, Dc)
[1, cD]	(1, cD), (2, cD), (3, cD), (4, cD), (5, cD), (6, cD)
[1, dC]	(1, dC), (2, dC), (3, dC), (4, dC), (5, dC), (6, dC)
[1, C ² D]	(1, C ² D), (2, C ² D), (3, C ² D), (4, C ² D), (5, C ² D), (6, C ² D)
[1, C ² d]	(1, C ² d), (2, C ² d), (3, C ² d), (4, C ² d), (5, C ² d), (6, C ² d)
[1, CDc]	(1, CDc), (2, CDc), (3, CDc), (4, CDc), (5, CDc), (6, CDc)
[1, CdC]	(1, CdC), (2, CdC), (3, CdC), (4, CdC), (5, CdC), (6, CdC)
[1, DC ²]	(1, DC ²), (2, DC ²), (3, DC ²), (4, DC ²), (5, DC ²), (6, DC ²)
[1, DcD]	(1, DcD), (2, DcD), (3, DcD), (4, DcD), (5, DcD), (6, DcD)
[1, cDc]	(1, cDc), (2, cDc), (3, cDc), (4, cDc), (5, cDc), (6, cDc)
[1, C ² Dc]	(1, C ² Dc), (2, C ² Dc), (3, C ² Dc), (4, C ² Dc), (5, C ² Dc), (6, C ² Dc)
[1, C ² dC]	(1, C ² dC), (2, C ² dC), (3, C ² dC), (4, C ² dC), (5, C ² dC), (6, C ² dC)
[1, CDcD]	(1, CDcD), (2, CDcD), (3, CDcD), (4, CDcD), (5, CDcD), (6, CDcD)
[1, DC ² d]	(1, DC ² d), (2, DC ² d), (3, DC ² d), (4, DC ² d), (5, DC ² d), (6, DC ² d)
[1, C ² DcD]	(1, C ² DcD), (2, C ² DcD), (3, C ² DcD), (4, C ² DcD), (5, C ² DcD), (6, C ² DcD)

Yet again every one of the above 24 equivalence classes has six members, and this time each equivalence class [1, n] ($n \in S(4)$) has members (x, n) , where $x \in \{1, 2, 3, 4, 5, 6\}$.

In summary, for the example where we have a monoid morphism f from $S(3)$ to $S(4)$, and where the $S(3)$ -set X is made up of the elements of $S(3)$, we have found that there is a total of 34 different ways of defining f — and we have looked at three of these different ways of defining f , where each of the 3 f s that we looked at had different *kernels*.

The information that we have obtained by looking at our three different ways of defining f has given us a heap of information to analyse in further detail, and this really is the main advantage and purpose of the computer programs created in this project — they allow the generation of data to analyse which would otherwise take an age to compute by hand, thus allowing the mathematician to concentrate on analysing the data instead of on generating the data.

4.2: Comparisons

The intention when starting this project was first to construct programs in order to apply the Knuth-Bendix methods to free monoids and to induced monoid actions, and then to compare the efficiency of our programs with other methods of implementation by comparing the processor time taken and total memory used to compute certain '*benchmark*' examples.

However, on completion of the programs, it was found that there weren't many other programs or algorithms available for direct comparison with — indeed, in spite of the relative wealth of mathematical algorithms available in packages such as Maple and Mathematica, only one package was obtained containing a program suitable for direct comparison with, that package being the KBMAG package.

KBMAG stands for "*Knuth-Bendix in Monoids, and Automatic Groups*", and the package is written by Derek Holt of the University of Warwick to be used either as a 'stand-alone' package or as a share package in the third version of the GAP system (GAP = Groups, Algorithms and Programming). The program in KBMAG that we were interested in was the `kbprog` program, the reason being that `kbprog` allows the computation of complete rewrite systems associated with finitely presented monoids. Further, as the program can be used with the *length-lex* ordering, it follows that it carries out exactly the same task as our programs *kba* and *kbe* do when they are asked to find complete rewrite systems associated with finitely presented monoids.

To begin the comparison, the two programs `kbprog` and `kbe` were asked to compute the complete rewrite system associated with the alternating group $A(4)$, which was presented as follows:

$$A(4) = \langle A, a, B, b \mid Aa, aA, Bb, bB, A^3, B^3, ABAB \rangle.$$

The two programs had no trouble in computing this example, and here are the respective inputs and outputs for the two programs:

KBE

Input File:

```
A; B; a; b;
A*a; ;
a*A; ;
B*b; ;
b*B; ;
A^3; ;
B^3; ;
(A*B)^2; ;
```

Command:

```
demo:gareth/kbia> kbe a4.in
```

Output Obtained (Complete Rewrite System):

```
This is the Complete rewrite system for M:
(B a B -> A b A)
(b A b -> B a B)
(a B a -> B a B)
(B A b -> a B)
(b A B -> B a)
(A B a -> b A)
(a B A -> A b)
(b a -> A B)
(a^2 -> A)
(a b -> B A)
(b^2 -> B)
(b B -> 1)
(B b -> 1)
(a A -> 1)
(A a -> 1)
(A^2 -> a)
(B A B -> a)
(A B A -> b)
(B^2 -> b)
[19 rule(s)]
```

Time Taken:

10 milliseconds

KBPROG

Input File:

```
#Alternating group A(4)
_RWS := rec(
  isRWS := true,
  ordering := "shortlex",
  generatorOrder := [A,B,a,b],
  inverses := [a,b,A,B],
  equations := [ [A^3,IdWord], [B^3,IdWord],
                 [(A*B)^2,IdWord] ]
);
```

Command:

```
demo:kbmag/test> kbprog a4 -v
```

Output Obtained (Complete Rewrite System):

```
_RWS := rec(
  isRWS := true,
  isConfluent := true,
  maxeqns := 19,
  generatorOrder := [A,B,a,b],
  ordering := "shortlex",
  inverses := [a,b,A,B],
  equations := [
    [A*a,IdWord],
    [B*b,IdWord],
    [a*A,IdWord],
    [b*B,IdWord],
    [A^2,a],
    [B^2,b],
    [A*B*A,b],
    [a^2,A],
    [b^2,B],
    [b*a,A*B],
    [a*b,B*A],
    [B*A*B,a],
    [A*B*a,b*A],
    [a*B*A,A*b],
    [b*A*B,B*a],
    [B*A*b,a*B],
    [a*B*a,A*b*A],
    [b*A*b,A*b*A],
    [B*a*B,A*b*A]
  ]
);
```

Time Taken:

12 milliseconds

As you can see, the time taken to compute the complete rewrite system for $A(4)$ is negligible for both programs, so that no conclusions can be drawn about which program is the most efficient in this example. This was also the case when we tested both programs with *other* simple examples such as $S(3)$ or $C(4)$, say — so in order to test the programs with a more difficult example, we asked the programs to compute the complete rewrite system associated with the alternating group $A(6)$, presented as follows:

$$A(6) = \langle A, a, B, b \mid Aa, aA, Bb, bB, A^2, B^4, (AB)^5, (AB^2)^5 \rangle.$$

This time, the inputs and outputs were as follows:

KBE

Input File:

```
A; B; a; b;
A*a; ;
a*A; ;
B*b; ;
b*B; ;
A^2; ;
B^4; ;
(A*B)^5; ;
(A*(B^2))^5; ;
```

Command:

```
demo:gareth/kbia> kbe a6.in
```

Output Obtained (Complete Rewrite System):

No output obtained –
the program did not terminate.

Time Taken:

Unknown — Program Halted after 1 hour

KBPROG

Input File:

```
#Alternating group A_6
_RWS := rec(
  isRWS := true,
  ordering := "shortlex",
  generatorOrder := [A,B,a,b],
  inverses := [a,b,A,B],
  equations := [ [A^2, IdWord], [B^4, IdWord],
                 [(A*B)^5, IdWord], [(A*B*B)^5, IdWord] ]
);
```

Command:

```
demo:kbmag/test> kbprog a6 -v
```

Output Obtained (Complete Rewrite System):

```
_RWS := rec(
  isRWS := true,
  isConfluent := true,
  maxeqns := 69,
  generatorOrder := [A,B,a,b],
  ordering := "shortlex",
  inverses := [a,b,A,B],
  equations := [
    [B*b, IdWord],
    [b*B, IdWord],
    [A^2, IdWord],
    [B^3, b],
    ...
    [b*A*B*A*B^2*A*B*A*b*A*B*A,
     A*B*A*B^2*A*b*A*B^2*A*B^2],
    [b*A*B^2*A*b*A*B^2*A*B*A*b,
     A*B*A*B*A*b*A*b*A*B*A*b*A]
  ]
);
```

Time Taken:

35 milliseconds

In this example, the `kbprog` program again terminates very quickly, but the `kbe` program does not terminate (or at least takes more than an hour to terminate, which is unacceptable when compared to the time that the `kbprog` program takes to compute the complete rewrite system for the above presentation for the alternating group $A(6)$). The reason for this becomes clear when you look at some of the other output provided by the `kbprog` program:

```
demo:kbmag/test> kbprog a6 -v
#69 eqns; total len: lhs = 664, 609; 212 states; 0 secs.
#No new eqns for some time - testing for confluence
#System is confluent.
#Halting with 69 equations.
#Exit status is 0
demo:kbmag/test>
```

It seems as though that during the Knuth-Bendix critical pairs completion algorithm, the `kbprog` program tests the rewrite system for *local confluence* when it fails to find a new rewrite rule after a specified length of time, recalling (from Definition 2.13) that a rewrite system is defined to be *locally confluent* if every time that an arbitrary word u reduces to words v and w using **single rules** (i.e. we have $u \rightarrow v$ and $u \rightarrow w$), then v and w reduce to a *common* term.

It can be shown that *local confluence* is a **sufficient** condition for *confluence*, so that if it can be shown that a rewrite system is locally confluent, then that rewrite system is automatically **confluent** — and so the Knuth-Bendix critical pairs completion algorithm can terminate with that rewrite system as output (it is the complete rewrite system that we want to obtain).

This means that the `kbprog` program can terminate quickly if it can prove the local confluence of its rewrite system, whereas we do not possess this short-cut and have to follow the Knuth-Bendix critical pairs completion algorithm to its natural conclusion — a task which will undoubtedly account for the long time that the `kbe` program takes to process this example due to the large size of the rewrite system involved (after an hour, the `kbe` program records that the rewrite system that it is looking at has at least 414 rules, so that there are at least ${}^{414}C_2$ comparisons of left hand sides of rules to be made — a task which will involve a lot of computation!)

We must therefore conclude that for the above example, it is the `kbprog` program that is the most efficient, but we note that it achieves its results by applying algorithms that we have not yet implemented in our programs (see the next section for more discussion on this). Who knows which program would be the most efficient if our `kbe` program had a local confluence test — would we then have (with other examples) the proof that the *frmon* library enables more efficient implementation of certain mathematical algorithms than any other method?

Whatever the answer to the above question, we must conclude that our programs *are* the most efficient at computing complete rewrite systems for induced monoid actions, simply due to the fact that there are *no other similar programs or algorithms* to compare any results with!

4.3: Improvements

When considering any major program or algorithm, improvements can always be made, and the '*kba*' and '*kbe*' programs in this project are by no means exceptions to this rule. Let us now consider a few improvements that could be made to our programs, improvements that could be implemented if given enough extra time or the relevant theory.

- (1) In the previous section, we saw that the `kbprog` program in the KBMAG package tested a rewrite system for local confluence during the execution of the Knuth-Bendix critical pairs completion algorithm if no new rewrite rules were found after a specified length of time. Our first potential improvement would therefore be to try to implement (in the `fMonKnuthBendix` algorithm) a similar test for our programs. A little investigation yields that there is a suitable algorithm for testing local confluence in Section 2.3 of [5], and more time would allow us to implement this algorithm in our code.
- (2) In Section 4.1, we considered examples in which we placed every member of the set $X \times S(4)$ in its corresponding equivalence class in the induced monoid N -set Y . In order to do this, we had to consider each member of the set $X \times S(4)$ individually, typing in a command in the terminal for *every* member of the set. Even though this process is still quicker than performing the same calculations by hand, it is also repetitive and therefore wasteful, in that the equivalence classes of Y have to be recalculated every time that we want to find out which equivalence class of Y contains a particular member of the set $X \times S(4)$.

A better solution would be to construct a third program, called 'classify', say, that took the output from the *kba* or *kbe* programs regarding the complete rewrite systems for the sets N and $X \times N^*$ plus the equivalence classes of the induced monoid N -set Y , and then placed each member of the set $X \times N$ in the correct equivalence class *automatically*, returning the kind of output that we presented in Section 4.1 (a list of the members of each equivalence class).

In order to implement the above, first we would need to build functions that could output data such as complete rewrite systems and lists of equivalence classes to disk, and then we would need to build the `classify` program itself in order to take the data on disk as input and produce a list of all the members of each equivalence class using this data, a task which would be implemented by repeating the same kind of reduction that we perform at the end of the *kba* and *kbe* programs (see lines 180-195 of page B32 of Appendix 2, for example).

Taking things a stage further, we could extend the usability of the 'reduce' program by allowing the user to see interactively which equivalence class of Y contains an arbitrary word from the set $X \times N^*$, or we could allow the user to pick any word from the set N^* to reduce using the complete rewrite system for the monoid N , and so on. The creation of a menu system would probably be the best way to proceed in this situation, so that the user could pick an option from the menu and enter input interactively, instead of having to place any input in one or more input files. The menu system could look as follows:

```
***REDUCE PROGRAM***
```

Choose an option:

1. List the Complete Rewrite System for the monoid N
2. List the Complete Rewrite System for the set $X \times N^*$
3. List the Equivalence Classes of the induced monoid N -set Y
4. Reduce an arbitrary word from the set N^*
5. Reduce an arbitrary word from the set $X \times N^*$
6. Classify an arbitrary word from the set $X \times N^*$
7. List all the members of the equivalence classes of Y

- (3) In either of our programs, once we have found all the equivalence classes that make up the induced monoid N -set Y , an extra task that could then be performed would be to find the *transformation representations corresponding to the action of the generators of N on all the equivalence classes of Y* .

For example, if Y was the set of equivalence classes $\{[1, 1], [1, C], [3, 1], [3, C]\}$, if N was generated by the generator 'C' and had a single relation $C^2 \rightarrow \lambda_N$, and if the action of a generator n' of N on an equivalence class $[x, n]$ of Y (written as $[x, n]^{n'}$) was defined to be (as it has been defined previously) the equivalence class containing the word (x, nn') , then the transformation representation of the action of the generator C on the equivalence classes of Y would be (numbering the elements of Y as $1 = [1, 1]$, $2 = [1, C]$, $3 = [3, 1]$ and $4 = [3, C]$) the transformation

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix},$$

because $\{[1, 1], [1, C], [3, 1], [3, C]\}^C = \{[1, 1]^C, [1, C]^C, [3, 1]^C, [3, C]^C\}$
 $= \{[1, C], [1, C^2], [3, C], [3, C^2]\} = \{[1, C], [1, 1], [3, C], [3, 1]\}.$

Let us now consider the *pseudo code* that could implement the above ideas, noting that the algorithm in the pseudo code would be implemented in the `rwsa.c` and `rwse.c` source code files, and then used at the end of the `kba.c` and `kbe.c` source code files.

Pseudo code to find the transformation representations corresponding to the action of the generators of N on all the equivalence classes of the set Y :

Inputs: The *equivalence classes* of the induced monoid N -set Y ;
 A *complete rewrite system* for the set $X \times N^*$;
 A list of the *generators* of the monoid N .

Output: *Transformation representations* corresponding to the action of each of the generators of N on all the equivalence classes of the induced monoid N -set Y , where each transformation has the form

$$\begin{pmatrix} 1 & 2 & 3 & \dots & p \\ t(1) & t(2) & t(3) & & t(p) \end{pmatrix}, \text{ where } p = |Y|.$$

Algorithm:

```

intFMonListPerm(Y, crwsXN, genN)
{
    Number the elements of Y as 1, ..., p
    (assuming that p = |Y|)

    For each generator g of N (for each element of genN):
    {
        For i goes from 1 to p (for each element of Y):
        {
            Let t(i) be the number corresponding to the
            equivalence class E, where E is obtained
            by reducing the word
            (Yi)g = (xi, ni)g = (xi, nig)
            using the complete rewrite system for the
            set X×N*, crwsXN
        }
        Let Tg be the list {1, t(1), 2, t(2), ..., p, t(p)},
        a list which represents the transformation
        representation for the action of the generator g of
        N on all the equivalence classes of the set Y.
    }
    Return all of the different Tg's to the piece of code
    that called this algorithm.
}

```

Conclusions

In this project, we have succeeded in constructing programs that allow the computation of complete rewrite systems associated with free monoids and induced monoid actions by using variations on the Knuth-Bendix critical pairs completion algorithm. The programs can currently be accessed by first logging in to the math2 server and then going to the `/mw3/usr/lib/freemon2/gareth/kbia` directory, and the two programs that can be found at this location perform the following tasks:

- Both programs allow the computation of a complete rewrite system for a monoid M (using the *usual Knuth-Bendix critical pairs completion algorithm*) by typing in either `kba M.in` or `kbe M.in` in a suitable Unix terminal, where `M.in` is an input file containing a presentation for the monoid M .
- To obtain a complete rewrite system for an induced monoid action (using a *modified version* of the Knuth-Bendix critical pairs completion algorithm), the `kba` program is used when you can give a presentation for the monoid N , can give the effect of the action of each of the generators of M on each of the elements of the monoid M -set X , and can give the monoid morphism f in terms of its effect on each generator of M . A typical command in this instance is `kba N.in data.in`.
- Again to obtain a complete rewrite system for an induced monoid action (using a *modified version* of the Knuth-Bendix critical pairs completion algorithm), the `kbe` program is used when you know that the monoid M -set X is the set of elements of the monoid M , know that the M -action x^m ($x \in X, m \in M$) is the monoid product $x \times m$, and know that you can give presentations for M and N plus the monoid morphism f in terms of its effect on each generator of M . A typical command in this instance is `kbe M.in N.in data.in`.

In this report, we have discussed the theory behind what makes the programs work, discussed how the programs were implemented using the C programming language and the *frmon* library of functions, and discussed improvements that can be made to the programs as they stand now.

I have enjoyed participating in this project, especially in the implementation and testing of the algorithms in the C programming language. At this stage, I would like to thank Professor Larry Lambe for providing us with the *frmon* library of functions so that the project could start, Professor Ronnie Brown for guiding us through the theory behind the algorithms, but most of all Dr. Christopher Wensley for all the advice that he gave throughout the year. This report would not have been possible without his support.

Gareth Evans 2001-2002

e-mail address: mau402@bangor.ac.uk

Bibliography

Books

- [1] BAADER, F. & NIPKOW, T. (1998) Term Rewriting and All That, 1st ed. Cambridge University Press
- [2] BARNARD, T. & NEILL, H. (1996) Teach Yourself Mathematical Groups, 1st ed. Hodder & Stoughton
- [3] BECKER, T. & WEISPFENNING, V. (1993) Gröbner Bases, 1st ed. Springer-Verlag
- [4] BROOKS, D. R. (1998) C Programming: The Essentials for Engineers and Scientists, 1st ed. Springer-Verlag
- [5] BOOK, R. V. & OTTO, F (1993) String-Rewriting Systems, 1st ed. Springer-Verlag
- [6] COX, D. & LITTLE, J. & O'SHEA, D. (1998) Using Algebraic Geometry, 1st ed. Springer-Verlag
- [7] EPSTEIN, D. B. A. (1992) Word Processing in Groups, 1st ed. Jones and Bartlett
- [8] HOWE, J. M. (1995) Fundamentals of Semigroup Theory, 1st ed. Oxford University Press
- [9] HUMPHREYS, J. F. (1996) A Course in Group Theory, 1st ed. Oxford University Press
- [10] MAC LANE, S. (1998) Categories for the Working Mathematician, 2nd ed. Springer-Verlag
- [11] NEUMANN, P. M. & STOY G. A. & THOMPSON E.C. (1994) Groups and Geometry, 1st ed. Oxford University Press
- [12] HUMPHREYS, J. F & PREST, M. Y. (1989) Numbers, Groups and Codes, 1st ed. Cambridge University Press

Research Papers

- [A] BROWN, R. & HEYWORTH, A.
Using Rewriting Systems to Compute Left Kan Extensions and Induced Actions of Categories
J. Symbolic Computation (2000) 29, 5-31
- [B] HEYWORTH, A. & WENSLEY, C. D.
Logged Rewriting and Identities Among Relators
Groups St. Andrews 2001: LMS Lecture Notes Series (to appear)
Cambridge University Press

Miscellaneous

Notes from the G2M31 Abstract Algebra and G3M04 Combinatorial Group Theory courses