University of Liège
Faculty of Applied Sciences

Master Thesis

Automatic demonstration of mathematical conjectures.

Author : Floriane Magera

Supervisor : Prof. Bernard Boigelot

Master thesis submitted for the degree of

MSc in Computer Science and Engineering

Academic year 2015-2016

Contents

Chapter 1

Introduction

Mathematicians have always looked for automating proof techniques. This project was popular at the end of the XIXth century, with the Hilbert program. David Hilbert thought that every mathematical conjecture could be decided mechanically, as the result of a computation that could be carried out without using any intuition [Hil05].

That belief was found to be invalid in 1931 by Kurt Gödel with his theorems of incompleteness [Göd31]. He showed that all sufficiently complex formal systems can prove some formulae but there will always be some true formulae that can not be proved. This means that there does not exist an algorithm able to decide any arithmetic formula, if this arithmetic is expressive enough. It is the case for the first order arithmetic, defined by Peano's axioms. There are thus formulae expressed in this arithmetic that can not be proved true or false.

There still exists some decidable formal systems, but these are therefore necessarily less expressive : Presburger arithmetic is one of them[Sta84]. Presburger arithmetic is the first order theory of the natural numbers with addition. Theoretically, one can develop a system that is able to decide automatically the truth of any formula in this arithmetic. For Presburger arithmetic, automatic theorem provers have been developed : ACL2[Ray10], E[Sch02], LASH[Boi], etc.. The formulae expressed in this arithmetic can be translated in automata, but one could represent things with automata that could not be expressed in this logic : automata are more expressive. So one could imagine a more complex logic that would reach the level of expressivity of automata. An example of such a logic is the monadic second order theory. The monadic second order theory is the Presburger arithetic extended with the notion of tables : infinite words that can be indexed. The logic operates on natural types as Presburger arithmetic, yet there is a restriction on the numbers acting as indexes : the addition operation is not allowed on them. This system is decidable and the tool Mona is a theorem prover for this logic[Hen+95].

A particular logic has been introduced for reasoning about automatic sequences. An automatic sequence $\mathbf{x} = (a(n))_{n\geq 0}$ can be generated by a deterministic finite automaton that takes as input a natural number n and outputs $a(n)$ which is the n$^{\text{th}}$ term of the sequence. Properties of these sequences defined with quantifiers, logical operators, integer variables, addition, indexing on the sequence, and comparison operators are decidable. An example of a formula in this logic is the following :

$$\exists i \forall j (j < n) \Rightarrow (T[i+j] = T[i+n-1-j])$$

This formula can be understood as : "For which n does the Thue-Morse word have a palindrome of length n?". The Thue-Morse word is an automatic sequence : the n$^t h$ term of the sequence is defined by the parity of the number of ones in the binary representation of n, if there is an even number of '1', then $T[n] = 0$ otherwise there is an odd number of '1' and $T[n] = 1$. For example, $T[5] = 0$ as the binary representation of 5 is 101, and thus the number of '1' is even. Some examples of the kind of properties proofs are showed in the papers [Du+14], [HS12], [GSS12], [RNP13] and [DPR16]. This work is focusing on a tool, Walnut, developed for automating deciding procedures for this logic. The output of Walnut for the formula given as an example above is an automaton that accepts all the words n that are possible lengths.

In its current form, Walnut works with natural numbers, commonly represented in the binary numeration system. In a more general system, numbers can also be represented differently. A definition of numeration system is given in the user manual of Walnut [Mou16]:

"A number system S is a 3-tuple (Σ_S, R_S, val_S) of alphabet $\Sigma_S \supseteq \{0,1\}$, language $R_S \subset \Sigma_S^*$ of valid representations containing 0^* and at least one of 0^*1 or 10^* and decoding function $val_S : R_S \to \mathbb{N}$ that assigns integers to every word in R_S. The decoding function has the following additional properties :

- $val_S(z) = 0$ iff $z \in 0^*$
- $val_S(1) = 1$

- For all $w \in R_S$, either $zw \in R_S$ and $val_S(zw) = val_S(w)$ or $wz \in R_S$ and $val_S(wz) = val_S(w)$ for all $z \in 0^*$. The former is called an msd (most significant digit first) number system and the latter is called an lsd (least significant digit first) number system.
- For all natural numbers n, there is a representation $w \in R_S$ such that $val_S(w) = n$.

The addition relation $+_S \subset R_S^3$ is defined such that $(x, y, z) \in +_S$ if and only if x, y, z are of the same length and $val_S(z) = val_S(x) + val_S(y)$. The equality relation $=_S \subset R_S^2$ is defined such that $(x, y) \in =_S$ if and only if x and y are of the same length and $val_S(x) = val_S(y)$. The less than relation is defined as $<_S \subset R_S^2$ for which $(x, y) \in <_S$ if and only if x and y are of the same length and $val_S(x) < val_S(y)$.

Number systems for which the automata for representations, addition, equality, and less-than exist, and equality is the same as word equality, i.e., $(x, y) \in =_S$ if and only if $x = y$, are exactly the type of number systems one can define and use in Walnut. Note that the alphabet of a number system is restricted to finite subsets of $\mathbb{Z}$ due to the same restriction on automata in Walnut."

All the natural numeration systems are built-in, for example one can decide to express a formula in base 3 such that $Rep_3(9) = 100,\ Rep_3(4) = 11, ...$ A user can use a new numeration system if it satisfies the previous condition and if the addition automaton is provided explicitly, i.e. if the user inputs an automaton accepting the language $+_S$. The contribution of this master thesis is to automate fully this step, by creating a method to generate the addition automaton of new numeration systems, namely the Fibonacci numeration system.

1.1 Description of the master thesis

The reader not familiar with numeration systems is invited to read the Chapter 2 first.

The goal of this work is to develop an algorithm for generating the addition automaton for some Pisot numeration systems. Pisot numeration systems are linear : there is a linear recurrence relation on the values of the sequence of naturals constituting them. A subset of the Pisot numeration systems is composed of the Fibonacci numeration systems. The work will focus on these numeration systems, as a starting point for further generalization. These numeration systems are defined by a Fibonacci polynomial i.e polynomials of the following form :

$$X_m = X_{m-1} + ... + X_0$$

from this equation, a numeration system U can be created. It is defined by a strictly increasing sequence of positive integers (U_n) with $U_0 = 1$, U_1,... U_{m-1} arbitrarily chosen . Then for every $j \geq 0$, $U_{m+j} = U_{m+j-1} + ... + U_j$. Given the degree of the polynomial (which will will be denoted by m along the paper) and the first values of the basis $\{U_0, ...U_{m-1}\}$, the aim is to construct the addition automaton in that base. The numeration system defined by the Fibonacci polynomial of order m will be denoted by F_m along the work.

The work relies on the assumption that the alphabet is defined by $\mathcal{A} = [0, 1]$. The first values of the base must be chosen in a way such that every integer is representable and thus that the numeration system is complete. Along this work, we assume that it is properly the case.

The generation of the addition automaton may be divided in several steps :

- **Addition bit by bit** which will be discussed in Section 3.1.
- **Extended normalization** : The result of the previous step is a number on the initial alphabet doubled. The number must be transformed such that it falls back to the initial alphabet. This problem is studied in Section 3.2 and in Chapter 4. In the solution proposed, there is a supplementary sub problem, the post normalization : it is the problem of extended normalization restricted to the end of the words. This problem is addressed in Section 5.1.
- **Normalization** : the latter number is transformed in normal form. A solution is provided for this problem in Section 5.2.

The core problem of the work is the extended normalization, the transformation of a word on $[0, 2]^*$ to $[0, 1]^*$ representing the same value. This problem has already been studied in earlier work [Fro99]. This approach will be presented, as it was partly followed. This work is presented in Chapter 3.2, then another solution to this problem is brought up in Chapter 4.

Another objective of the work was also to implement the solution. The implementation of the extended normalization is discussed in Chapter 6, some parts of Frougny's approach were also implemented.

The illustrations provided will be mostly examples in the F_2 numeration system : the Fibonacci numeration system of order 2 with the characteristic polynomial $X_{j+2} = X_{j+1} + X_j$. This choice is logical as it is the simplest Fibonacci numeration system and the number of states in the different automata stay low even in the final result.

Chapter 2

Numeration systems

In this chapter, the definitions and concepts needed to understand the work are presented. The properties of Fibonacci numeration systems are presented, some reminders about automata as well. Most of the properties and definitions given are taken from the book [Rig14]. First some definitions about numeration systems are given, then the notions of representation are introduced.

2.1 Positional numeration system

A positional numeration system is defined by an increasing sequence of integers $\boldsymbol{U} = (U_n)_{n\geq 0}$ such that every word $c_j ... c_0$ is the representation of the integer

$$\sum_{i=0}^{j} c_i U_i$$

The weight of the coefficient c_i depends thus on its position in the word. With this definition, we define also the representation of an integer in a numeration system as an encoding of its value in this system.

We will denote by the $rep_U(n)$ the function which gives the normal representation of the natural number n in the numeration system. The normal representation will be defined in Definition 2.5.1. Then we define also the $val_U(w)$ function, which given a word w returns the integer value represented by that word in the numeration system U.

Numeration systems are basically notations for representing numbers. Here we focus on natural numbers.

Numeration systems can be divided in several subsets which have interesting properties. The Fibonacci numeration systems are part of the Pisot numeration systems and the linear numeration systems which are discussed later.

2.2 Linear numeration system

A numeration system **U** is linear iff its sequence satisfies a linear recurrence relation :

$$U_{n+k} = a_{k-1}U_{n+k-1} + ... + a_0U_n$$

$\forall n \geq 0, a_0...a_{k-1} \in \mathbb{Z}$. From this definition, it is clear that Fibonacci numeration systems are linear.

2.3 Pisot numeration system

2.3.1 Pisot number

Definition 2.3.1 *A real number $\alpha > 1$ is a Pisot number if it is an algebraic integer(which means that is the root of a polynomial) whose conjugates have modulus less than one [Pis46].*

Pisot numeration systems are particular linear numeration systems : their characteristic polynomial of recurrence is the minimal polynomial of a Pisot number $\alpha > 1$. They have interesting properties :

- The normal representation of the naturals $rep_U(\mathbb{N})$ is a regular language.
- The mapping between a representation of a natural number n and its normal representation is also a regular language. Indeed a deterministic finite automaton which accepts the language

  $$\{(w, rep_U(n)) \mid val_U(w) = n\}$$

  exists for the defined alphabet. It is the automaton generated in the Section 5.2.
- The addition in the numeration system is also a regular language. Thus the goal of this work is to build an automaton which accepts the language

  $$\{(x, y, z) \in \mathbb{N}^3 \mid x + y = z\}$$

Fibonacci numeration systems are a subset of Pisot numeration systems. The characteristic linear recurrence relation of the Fibonacci numeration system of order k is of the form

$$U_{n+k} = U_{n+k-1} + ... + U_n, \ \forall n \geq 0$$

From the properties given for Pisot numeration systems, one knows that there will be a way to build a deterministic finite automaton for addition.

2.4 Some useful definition

Definition 2.4.1 *Let $w = w_0..w_{n-1}$ be a finite word of length n. Let i, j be such that $0 \leq i \leq j < n$. The word $w_i..w_j$ is a factor of w of length $j - i + 1$*

We define here the notion of ordering that we will need in the next definition.

Definition 2.4.2 *For two words u, v belonging to the same alphabet and without any prefix of the form 0^n, u is said to be genealogically smaller than v if either $|u| < |v|$ or if $|u| = |v|$, $u = paq$ and $v = pbl$ with $a < b$.*

Definition 2.4.3 *The lexicographical order is also defined : given two words u and v on the same alphabet, u is said to be lexicographically smaller than v, $u <_l exv$ if u is a proper prefix of v or if $u = paq$ and $v = pbl$ with $a < b$.*

2.5 Normal representation

The representation of a number in a numeration system is not always unique. That is why a normal representation is needed.

Definition 2.5.1 *Let $\boldsymbol{U} = (U_n)_{n \geq 0}$ be an increasing sequence of positive integers. U_0 is equal to 1 in order to be able to represent every integer. The normal representation of a natural number n $rep_U(n)$ is the genealogically greatest U-representation of n.*

If a word $d_r...d_0$ is a U-representation of n then

$$n = \sum_{i=0}^{r} d_i U_i$$

and the word $d_r...d_0$ is genealogically less or equal to the normal U-representation $rep_U(n)$.

To give an example, the representation of an integer in the Fibonacci numeration system which is defined by the sequence $\{1, 2, 3, 5, 8, 13, ...\}$ is not always unique. Indeed,

$$val_{F_2}(11) = val_{F_2}(100) = 3$$

And $rep_{F_2}(3) = 100$ as this representation is longer.

The following section presents a way to find the normal representation of an integer expressed in a positional numeration system.

2.6 Greedy representation of an integer

Given a numeration system $U = (U_n)_{n\geq 0}$ defined by an increasing sequence of positive integers and a positive integer ℓ, the normal representation $rep_U(\ell)$ is computed with the following algorithm. First, a value k is searched such that $U_k \leq \ell < U_{k+1}$, the result of the Euclidean division of ℓ by U_k is noted c_k thus $\ell = c_k U_k + r_k$ where r_k is the remainder. This procedure is repeated with the remainder until a sum of the following form is obtained :

$$\ell = \sum_{j=0}^{k} c_j U_j$$

Note that the values of c_j are not restricted here. In the context of this work, c_j should to be restricted to a certain alphabet. It depends on the numeration base used, as the first values of the base can be chosen, the user should provide values that allow every integer to be representable on our chosen alphabet $\mathcal{A} = [0, 1]$. As will be seen in Section 2.7, there is an interval in which each value can be chosen.

As said before, the first values of a Fibonacci numeration system can be chosen. However, as the alphabet allowed is $[0, 1]$ and as every natural number should be representable, these values are to be chosen carefully. The restrictions on these values are mentioned in the next section.

2.7 Completeness

A numeration system defined by F_m and an alphabet $\mathcal{A} = [0, 1]$ is said to be complete if every positive integer is representable. According to C. Frougny [Fro88], one way to check that a system is complete is to show that

$$U_0 + ... + U_k \geq U_{k+1} - 1$$

for every $k \geq 0$. Thanks to this condition, we will find out in which interval the first values of the base can be chosen.

- if $k \geq m$, the condition is trivially satisfied by the linear recurrence equation of Fibonacci numeration systems.
- if $k < m$, one shows that $U_1 = 2 : U_1 > U_0$ and $U_0 \geq U_1 - 1$ knowing that $U_0 = 1$,
$$1 < U_1 \leq 2$$
.

  then if $m > 2$, the condition becomes $U_0 + U_1 \geq U_2 - 1$, which gives
$$2 < U_2 \leq 4$$
. One can prove by induction that
$$U_k \leq 2^k$$
by the induction hypothesis, $U_i \leq 2^i$, $0 < i < k$:
$$2^0 + ... + 2^{k-1} \geq U_k - 1$$
$\sum_{i=0}^{k-1} 2^i$ is a geometric sequence of reason q=2. The sum of the n+1 elements of a geometric sequence (c_n) is given by
$$c_0 + ... + c_n = c_0 \frac{1 - q^{n+1}}{1 - q}$$
$$\sum_{i=0}^{k-1} 2^i = \frac{1 - 2^k}{1 - 2} = 2^k - 1$$
and trivially,
$$U_k \leq 2^k$$

The final condition is $U_{i-1} < U_i \leq 2^i$ with $1 < i < m$, so the user has to choose the first values of the base in this interval.

2.8 Automata and transducers

The notion of automaton and transducer are reminded here.

2.8.1 Automaton

Definition 2.8.1 *A deterministic finite automaton, or DFA for short, over an alphabet B is given by a 5-tuple $A = (Q, q_0, B, \delta, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times B \to Q$ is the transition function and $F \subseteq Q$ is the set of final states.*

The map δ can be extended to $Q \times B^*$ by setting $\delta(q, \epsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$ for all $q \in Q$, $a \in B$ and $w \in B^*$. Automata can be used to represent set of words, one says that an automata accepts a language $\mathcal{L}$.

Definition 2.8.2 *If the language L is accepted by the automaton A, then*

$$L = \{w \in B^* | \delta(q_0, w) \in F\}$$

. If $w \in L$ then the word w is accepted by the automaton A.

The aim of this work is thus to construct an automaton that will accept the language

$$\{(rep_{F_k}(x), rep_{F_k}(y), rep_{F_k}(z)) \mid x + y = z,\ (x, y, z) \in \mathbb{N}^3\}$$

for the Fibonacci numeration system of order k.

The notion of transducer will also be used as a more natural way of thinking about the addition automaton is a transducer with two input tapes and one output for the result of the addition. Some kind of transducers can be converted in DFA and conversely. These transducers are presented in the next section.

2.8.2 Transducer

A transducer is a 6-tuple $T = \{Q, A, B, \delta, q_0, \lambda\}$ where Q is a finite set of states, A and B are finite alphabets, usually called the input and the output alphabet respectively, $q_0 \in Q$ is the initial state. The transition function is $\delta \subset Q \times A \times B^* \times Q$. The output function is $\lambda : Q \to B^*$.
Such a transducer accepts a language

$$\{(f, g) \in A^* \times B^* \mid g = hw\}$$

, where (f, h) is the label of a path going from the initial state to one of the terminal states q, and $w = \lambda(q)$. A transition $\tau = (p, u, v, q)$ with $p, q \in Q$, $u \in A^*$ and $v \in B^*$, is noted $p \xrightarrow{u/v} q$.

A transducer is said to be length-preserving if the language accepted satisfies

$$\forall (f, g) \in L(T), |f| = |g|$$

A subclass of length-preserving transducers are the synchronous tranduscers. A transducer is said to be synchronous if for all its transitions $p \xrightarrow{u/v} q$, $|u| = |v| = 1$ [DN09]. This class of transducers can be translated in DFA, each transition of the automaton is just a tuple (u, v). There is no need for an output function λ for a synchronous transducer.

2.8.3 Composition of two synchronous transducers

Here the theoretical composition is explained.

Given two transducer $\mathcal{T}_1$ and $\mathcal{T}_2$, the goal is to compose them such that the output of $\mathcal{T}_1$ will be the input of $\mathcal{T}_2$. Formalized as automatons, the alphabets of $\mathcal{T}_1$ and $\mathcal{T}_2$ are defined:

$$A_1 = \Sigma_1 \times \Sigma_2$$

$$A_2 = \Sigma_2 \times \Sigma_3$$

It is important to note that the alphabet of the output of $\mathcal{T}_1$ and the alphabet of the input of $\mathcal{T}_2$ are equal. Otherwise the composition will fail. The second transducer can not take as input something that is not in its alphabet.

The first step is to expand the alphabet of both transducers. The alphabets are modified in the following way :

$$A_1' = \Sigma_1 \times \Sigma_2 \times \Sigma_3$$

$$A_2' = \Sigma_1 \times \Sigma_2 \times \Sigma_3$$

This can be obtained by modifying each transition of the transducers : each transition is transformed in n new transitions with $|\Sigma| = n$, with the same start and end, just the labels are expanded with the n^{th} value of Σ. Another way to do that is to make the Cartesian product of the automaton with a new automaton that accepts all words on the desired alphabet.

The next step is make the intersection of the two automata. A new automaton $\mathcal{I}$ is obtained. Its alphabet is :

$$A_{\mathcal{I}} = \Sigma_1 \times \Sigma_2 \times \Sigma_3$$

The second component is useless, as it is a temporary. the last step is thus by a projection on that second composant, giving as result

$$\mathcal{I}' : \Sigma_1 \times \Sigma_3$$

That was the theoretical explanation of the composition. The implementation is not as simple, a lot of different composition operations are performed. Indeed all the main sub-problems explained : addition bit by bit, extended normalization, post normalization and finally normalization will all be composed to obtain the addition automaton. Thus a more general composition is needed ; it should work also with automatons with 3 tapes for examples. The implementation is discussed in Chapter 6.

In this work, one will mostly deal with transducers. Note that transducers can be considered as automata with tuples on their transitions, only if the transducer is synchronous.

Chapter 3

State of the art : Frougny's algorithm

As explained in the introduction, the generation of the addition automaton can be divided in three sub tasks.

1. Bit by bit addition,
2. Extended normalization : this step transforms a word on $[0,2]^*$ into another word on $[0,1]^*$ encoding the same value,
3. Normalization : here it means convert each word in a new word encoding the same value, without having any 1^m factors.

First the problem of bit by bit addition is solved.

3.1 Addition bit by bit

The first step of the addition process is the easiest. It is the addition bit by bit, one simply adds two inputs, without wondering about the output alphabet. As one can see on Figure 3.1, the created automaton A accepts the language

$$\{(x,y,z)|x+y=z,x,y\in\mathcal{A}\}$$

The automaton is composed of only one state.

A way to go further is to require the inputs x and y to be in their normal form. It is obtained by creating an automaton which only accepts words in their normal form. The normal form of words in Fibonacci numeration system of degree m are the words which do not contain the factor 1^m.

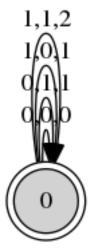

Figure 3.1: Simplest transducer for addition bit by bit for F_2

The automaton N accepting the normalized words is thus composed of m states, all being accepting. Every '1' in the word leads to the next state, and every '0' goes back to the initial state. The automaton N_2 is represented in Figure 3.2.

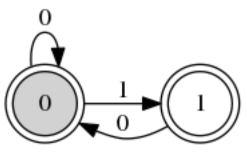

Figure 3.2: Automaton accepting normalized words in F_2

In order to ensure that x and y are normalized, the Cartesian product of the automaton N with itself is performed, and then it is composed with the simple addition automaton A. The final result is showed on Figure 3.4. The automaton takes as input two normalized words and outputs a word which is on the doubled alphabet and which is not in its normal form. The extended normalization that has been explained brings back the output word to the correct alphabet. The next section focuses on the normalization of the word.

The step following addition bit by bit is the extended normalization. It is the main focus of this work. the extended normalization happens after the raw bit by bit addition of two numbers in their normal representation developed here. The problem is that the normal form has not been conserved for the output. Indeed, the alphabet of the output is now doubled and moreover, the addition of two numbers in their normal form can lead to a representation that is no longer normal as one can see in the following simple example in F_2: $01 + 10 = 11$.

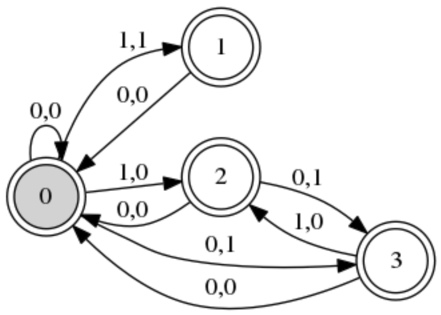

Figure 3.3: Automaton reading two normalized words in F_2

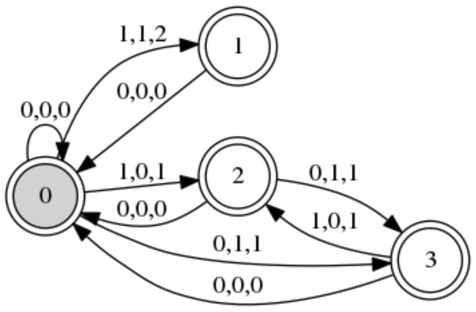

Figure 3.4: Final addition transducer for normalized input in F_2

For these reasons, first the output's alphabet must be reduced, and then it should be put in its normal form. Here, this chapter focus on the reduction of the alphabet, which is called the extended normalization. We present a method proposed by C. Frougny [Fro99] to solve this problem in this chapter. Some mechanisms presented will be reused in the normalization part.

The problem of extended normalization consists in answering the question : knowing the Fibonacci polynomial of the numeration system of interest, how can a word on $[0,2]^*$ be transformed to another word on $[0,1]^*$ encoding the same number? For example, in F_2, it is easy to see that the following transformation is correct, but the point is to develop an algorithm to perform such a transformation automatically and efficiently.

$$020011 \rightarrow 101001$$

The discovery of how to perform extended normalization in an efficient way was the core problem of the work, and is still the part that could need further improvement to obtain an efficient generation of the addition automaton for Fibonacci polynomials of high degree.

3.2 Frougny's algorithm

The main source of this work is an article of Christiane Frougny which describes the procedure to obtain a transducer that is capable of decreasing the alphabet of an incoming word by one when applied m times. The method will be explained for our specific case. The transducer reads words from left to right, applying sequentially rules that will be explained here.

The whole approach relies on rules that are to be applied to transform a word. These rules are essential and will be used further, whilst the rest of the procedure is not used in the final implementation.

3.3 Rules

In order to transform a word into a word encoding the same value but containing a smaller number of '1' or '2', the characteristic polynomial of the numeration system can be used. Two types of rules can be derived from it : reductions and unfoldings.

3.3.1 Reductions

These are performed by using the polynomial of the numeration system on factors of size $m+1$. The left-most letter is thus increased by one and all the others are decreased by one. It could be interesting to apply it several times. The set of possible reductions is described by :

$$\{0(i_1+1)...(i_{m-1}+1)l \to 1(i_1)...(i_{m-1})(l-1) \mid 1 \le l \le 2,\ 0 \le i_1, ..i_{m-1} \le 2\} \tag{3.1}$$

For example, the next set represents all the reduction rules for the F_2 numeration system.

$$R = \{011 \to 100, 012 \to 101, 021 \to 110, 022 \to 111, 031 \to 120, 032 \to 121\}$$

In practice, the last rule of R is not applied. It is better to reduce as much as possible so the transformation used is $032 \to 210$. On one side, this leads to a factor on the same alphabet and moreover it produces a zero, which is necessary to apply more rules. For example, one could face a word $w = 03221$ with the theoretical rule it could be transformed in that way :

$$03221 \to 12121$$

And then no other rule is applicable. If it is transformed in the other way, it becomes :

$$03221 \to 21021$$

After that, the last factor can be reduced too by a new rule application.

$$03211 \to 21110$$

As the final result. The words are not normalized yet, but the second result is closer to be, as there is only one '2' remaining.

Reductions are also used for the last step of normalization.

3.3.2 Unfoldings

These consist in developing a factor with the polynomial and then reducing a factor to the left of the developed one. The unfoldings take the following form :

$$\begin{array}{r} \{\ 0(i_1+1)...(i_{j-1}+1)(i_j+1)\nu_{j+1}...\nu_{j+m}\ \to \\ 1(i_1)...(i_{j-1})(i_j-1)\nu_{j+1}...\nu_m(\nu_{m+1}+1)...(\nu_{m+j}+1)\ \} \end{array} \tag{3.2}$$

with the following conditions on the variables :

$$1 \le j \le m-1$$
$$0 \le i_1, ..i_{m-1} \le 2$$
$$\nu_{j+1}...\nu_{j+m} \in [0,2]$$
$$\nu_{j+1}...\nu_m <_{lex} 1^{m-j-1}1$$

Where $<_{lex}$ denotes the lexicographic ordering.

For example, here are the sets of unfoldings of F_2 :

$$Un = \{0200 \to 1001, 0201 \to 1002, 0202 \to 1003, 0300 \to 1101, 0301 \to 1102, 0302 \to 1103\}$$

Unfoldings are needed specifically in the case of an extended alphabet, where words of the type 02^n can occur. These words can not be reduced with a simple reduction. Note also that unfolding can extend the alphabet further to $[0, ..3]$.

Now the two sets of rules will be used to transform each factor matching the left member of a rule. The next step is to create a transducer which will read a word from left to right, and will transform the word using the predefined rules.

3.4 Creating the transducer

The goal of the method is to construct a transducer which will apply the rules on its input in order to perform extended normalization. The construction of this transducer is described in this section.

3.4.1 Generate the state space

The state space of the transducer is directly extracted from the rules. The states represent the current factor being considered at a certain point when waiting for a matching with a rule. The set of states is then the set of *strict* prefixes of the left member of the rules. The initial state is ε.

3.4.2 Generate transitions

From each state q, each possible input x is considered. Let $w = qx$. The word w from which the factor y has been removed is denoted by $w \backslash y$. There are three cases :

- There is no rule matching w. There are then two possibilities :
  - w matches an other state q' ($w = q'$): then nothing is written on the output tape and the next state for input x is q'.
  - a suffix of w matches an other state q'. If there are several matching suffixes, the longest one is chosen. The remaining part $w \backslash q'$ is written on the output tape and the next state for an input x is q'.
- There is a *reduction* rule matching w. If $w \to w'$ and w' can not be reduced anymore (otherwise reduction rules are again applied), w' is considered. There are again two cases. The last letter of w is named ℓ:
  - ℓ belongs to the reduced alphabet, so $\ell = 1$ (ℓ can not be '0' to allow a reduction) in this case. Then $w' \backslash (\ell - 1)$ is written on the output tape and the next state is $\ell - 1$. In this case, the next state is always '0'.

– ℓ does not belong to the reduced alphabet, as ℓ is the last digit, it can only be equal to 2. From the sturcture of unfoldings, there will always be at most $(m-1)$ digits '3' created, so the last digit can not be a '3'. Then the form of w' must be considered : $(h+1)(i_1)...(i_{m-1})1$, let k be the greatest integer such that $i_k = 0$ and $i_{k+1}, ...i_{m-1} \geq 1$. The output for this transition is everything before i_k : $(h+1)i_1..i_{k-1}$. Moreover the next state is $0i_{k+1}..i_{m-1}1$. If there does not exist such a k, then it means that one more reduction can be applied. After applying it, this step can be applied.

- An *unfolding* rule is applicable. If $w \to w'$, w' is considered. The longest strict prefix of w', n is searched such that n belongs to the reduced alphabet and such that the suffix $w' \backslash n$ begins with '0'. The prefix n is written on the output tape. This decomposition exists because of the requirement $\nu_{j+1}...\nu_m <_{lex} 1^{m-j-1}1$ in Equation 3.2 on unfoldings. The next state is $w' \backslash n$.

Find the output value for each state

Frougny's article does not give the choice of the first base values. The values of the m first values must necessarily be powers of two : $U_0 = 2^0, ...U_{m-1} = 2^{m-1}$. This part of her method is explained to be complete, even if it was actually never used as we wanted to allow other values for the m first values. A new approach is developed in Section 5.1.1. This part defines the output function of the transducer.

The output function $\omega(q)$ is defined for each type of state. There are several cases :

- If the state $q \in [0, 1]^*$ then the output satisfies $\omega(q) = q$.

- Otherwise, q must be normalized, but by definition of the states there are no rule that can be applied. Let $q = hq_1...q_\ell$ with $1 \leq \ell \leq 2m-2$. The approach proposed is to continue the base with $U_{-1} = 1, U_{-2} = 0, ...U_{-m} = 0$. The label of q is suffixed with m zeros : $q = hq_1...q_\ell, 0^m$ If $\ell \leq m-1$, an unfolding rule can be applied.

$$q \to (h+1)i_1..i_{j-1}(i_j - 1)q_{j+1}...q_\ell, 0^{m-\ell}1^j$$

By definition of the $U_{-2}, ...U_{-m}$ and as $\ell \leq m-1$, the output value is equal to

$$(h+1)i_1..i_{j-1}(i_j - 1)q_{j+1}...q_\ell$$

if $m \leq \ell \leq j + m - 1$ The same method as previously is applied : q is suffixed with zeroes and one unfolding is also applied, it is possible to generate '1' at indexes greater than ℓ. As $U_{-2} = ... = U_{-m} = 0$, the only problem is a '1' located at the index $\ell + 1$. The work does not specify exactly how to behave in this case, so an idea of resolution is given. If $q_\ell < 2$, q_ℓ is increased by 1. Otherwise a second unfolding can be applied with $q_{\ell+2}...q\ell + m = 0$. As a result, q_{ell} will be reduced, and if $q_{\ell+1}$ is still different from zero, it can be added to q_ℓ.

Final result

The resulting transducer for F_2 is showed on Figure 3.5.

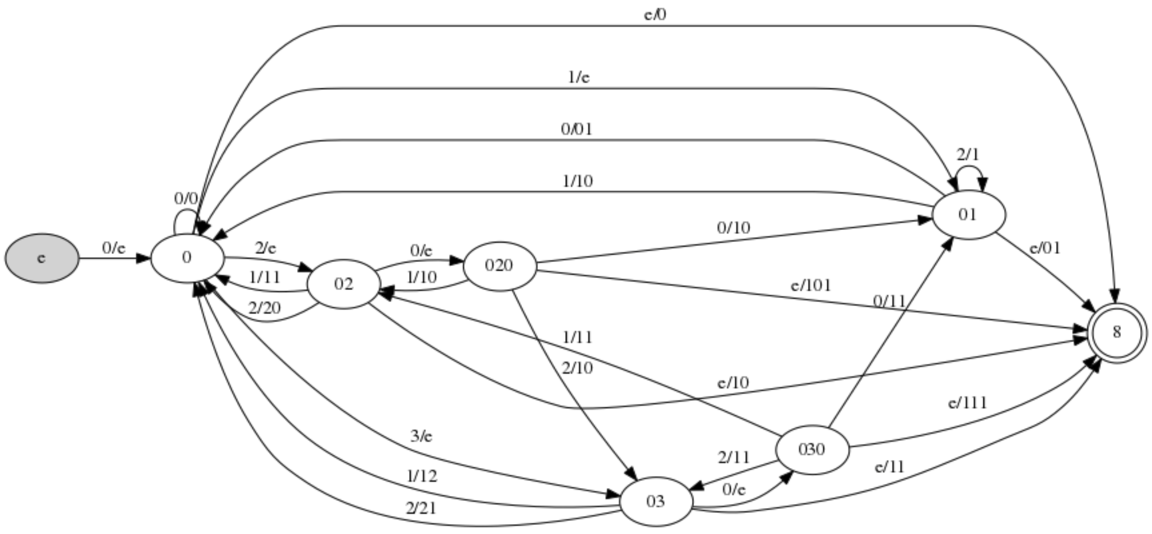

Figure 3.5: Transducer obtained with Frougny's method

Now the next step is to transform this transducer in an automaton as we want to obtain the addition automaton eventually. However this transducer can not be translated in a deterministic finite automaton because of the shape of its transitions. The transitions generated vary in length which would not be a problem if the length of the input and output were always equal. The input length is always equal to one or zero, and the output's length vary between $[0, 2]$. In a DFA, there is no output nor input, the transition labels would be translated in a tuple, it won't be possible to know where the input stops and where the output starts. The transducer could be translated in an automaton if the output and input were of the same length. This would also mean that the output function of the transducer would always output ϵ, so it would not be needed to specify it.

In the next Chapter, a new approach is developed to construct a synchronous transducer i.e a length preserving transducer. The aim is to keep as much similarities with Frougny's method, to keep the same interesting properties : i.e. the guarantee of a reduced alphabet in m passes through the transducer that will be developed.

In order to keep the same properties, we try to keep the same effect and strategy : the transducer generated will apply rules, and another interesting property to maintain is that after applying a rule on a factor of the word, if there is a suffix of the transformed word that could match another rule, it will be kept it in memory, allowing overlapping rule applications.

The method proposed by Frougny was partly implemented, some parts are used in the final version. The implementation will be discussed in Chapter 6. First the theoretical explanations about the new methods considered is presented in the next chapter.

Chapter 4

Efficient extended normalization

Our aim is to obtain a synchronous transducer having the same effect as the one obtained in Section 3.2. It would be a transducer transforming a word thanks to the rules described, in a sequential way and such that the nesting of rules applications would be possible. In that way, the same efficiency could be obtained: the extended normalization would be performed in m applications of the transducer. If the alphabet was to be extended, it would not be the case anymore as m applications reduce by one the alphabet.

One solution would have been to transform the transducer to make it synchronous, but another solution would be to try to make the algorithmic simpler and to rely on simple operations on automaton. An approach relying on simple automata and operations on them is chosen.

First the problem is defined and formalized, then a first approach is developed and finally a final one is explained.

4.1 Problem

The aim is to mimic with a synchronous transducer the behaviour of the transducer obtained in Section 3.2. First, its effect can be formalized. The transducer applies rules, in a sequential way on words, from left to right. The application of one rule is considered as a basic action. This action can be repeated an unbounded number of times, but it is not a simple loop : as rule applications may be nested, the previous result of a transformation can be fed as input to the next one. But there should be a possibility of not applying any rule for a while. Whatever the solution to mimic the asynchronous transducer, the first step is to develop a basic automaton that applies one of the rules.

4.2 Base automaton

As there are two types of rules reductions and unfoldings, an automaton for each type will be generated. Each automaton will transform words matching the left member of its rule.

4.2.1 Reduction automaton

The reduction automaton is the simplest. the aim is to generate an synchronous transducer that reads the left member of a rule and outputs the right one. The number of rules for a numeration system grows quite fast, so an approach that reasons on the structure of the reduction rules is preferred. In order to build it in an efficient way, one can generalize the form of reductions. The set of reductions for F_2 modified for maximal reduction is considered:

$$R = \{011 \to 100, 012 \to 101, 021 \to 110, 022 \to 200, 031 \to 120, 032 \to 210\}$$

This set can be divided in two subsets :

$$R_1 = \{011 \to 100, 012 \to 101, 021 \to 110, 031 \to 120\}$$

and

$$R_2 = \{022 \to 200, 032 \to 210\}$$

These subsets correspond to the number of times one can apply a reduction on the factor. There will be one subset for each possible number of reductions, here there will always be at most two sets, because it is not possible to create factors 3^m. The factors containing '3' are generated by unfoldings which can produce at most 3^{m-1}. A first idea is thus to divide the state space of the transducer according to the subset of the rule which will applied : from the initial state, there will be a transition "0/1" to a new state s_1 and another transition "0/2" to another state s_2. The rules from R_1 will be continued from s_1 and the ones from R_2 from s_2.

Then there is another observation to be made : for each rule member of a set, there is always a digit that justifies the number of reductions. For example : if one considers the rule $021 \to 110$, the '1' in the left member is the restricting digit. It is important to generate complete rules. Without this observation, incomplete rules could generated, for example :

$$022 \to 111, 022 \to 200$$

The first rule is incomplete, it is not wrong : the transformation is correct, but it would lead to a less compact automaton, and moreover, it would bring non-determinism.

Starting from the states previously introduced s_i, there will be m transitions following (as reductions are always $m+1$ digits long). Along these transitions, the input will be decreased by the number of reductions i. the last observation impacts the input values considered at each new transition.

Further details about the exact implementation of the automaton and pseudo-code are presented in Chapter 6. The automaton resulting is shown on Figure 4.1

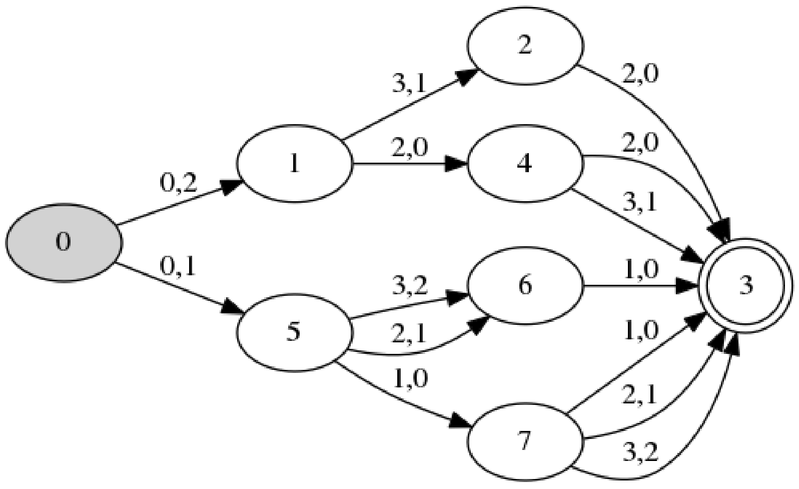

Figure 4.1: Transducer applying one reduction rule

4.2.2 Unfolding automaton

Now the unfolding automaton should be generated. This time, the F_3 numeration system is used as an example. Indeed the unfolding rules don't all have the same size, unlike reduction rules whose length is always equal to $m+1$. The different lengths are not noticeable for the F_2 base, because the length interval of an unfolding is $[m+2, 2m]$. So for F_2 the size of unfolding rules is fixed to 4, whilst for F_3 it is $[5, 6]$.

The approach is similar to the one developed for reductions. The generation of the automaton is based on observations on the unfolding rules. Some of the rules for F_3 are recalled :

$$U_3 = \{033022 \rightarrow 121033, 023011 \rightarrow 111022, 03102 \rightarrow 11103, 02010 \rightarrow 10011\}$$

While reductions could be divided according to the number of reductions possible, here unfoldings can be partitioned according to their size :

$$U_3^5 = \{03102 \rightarrow 11103, 02010 \rightarrow 10011\}$$

and

$$U_3^6 = \{033022 \rightarrow 121033, 023011 \rightarrow 111022\}$$

Similarly to the reduction automaton, the state space will again be divided according to the length of the rule : from the initial state, there will be a transition "0/1" to a state s_1 and s_2, here this transition is the same introducing non-determinism. The number of second states depends on the order of the Fibonacci numeration system considered.

Then the second observation is that there is a middle factor that remains unchanged in every unfolding. The length of this factor is defined by the rule's length. In unfoldings, the j^{th} term ≥ 2 is expanded, then the first m terms are reduced. So there will be $m - j$ terms unchanged between the indexes $j + 1$ and m. After these, the j following digits read are increased. All these observations are also easily noticeable in the definition of unfoldings in Equation 3.2.

There is one last thing to remark : the unchanged factor should be strictly smaller than the factor 1^{m-j} in the lexicographical order. It should be remembered whether an zero has already been written the middle factor when generating it. This will be dealt with in a similar way than the restricting digit in the reduction case.

Each second state s_i will correspond to a value for j, in the interval $[1, m-1]$, so the number of transitions generated after the second state depends on it. The idea of the rest of the generation of the automaton is then to keep a counter of the number of transitions generated while constructing the rules of a certain length. The counter indicates the action to perform : decreasing by one or two input values, generating the middle factor and finally increasing by one the input. It follows quite naturally the definition of the unfolding given in the previous chapter.

Further details about the exact implementation of the automaton will be discussed in Chapter 6. The resulting automaton is shown on Figure 4.2. For once, the example is from F_3. Note that the initial values of the base do not matter here, they are used in the step of post normalization, which is described in Section 5.1.

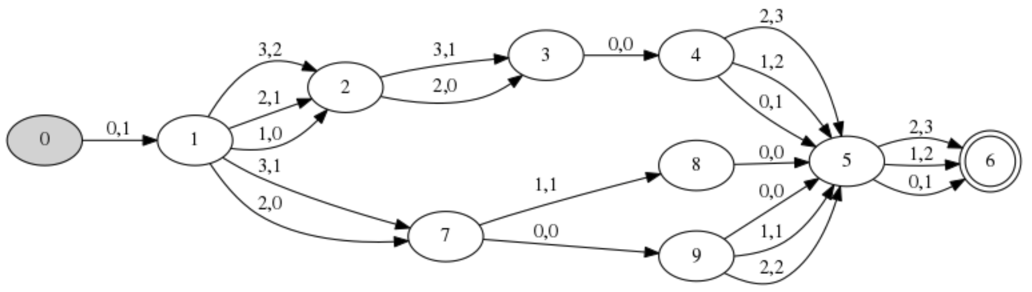

Figure 4.2: Transducer applying one unfolding rule

4.2.3 Creation of the base automaton

Now that two transducers applying reductions and unfoldings have been created, the last step to create the base automaton is to make the union of them in order to obtain an automaton that will apply one rule. It is important to notice that the final automaton will accept and transform words of length $\ell \in [m+1, 2m]$. The result of the union is shown in Figure 4.3. Once again, the result for the Tribonacci (F_3) numeration system is presented.

Now the problem is to figure out how to use this simple automaton in order to perform extended normalization. Some details that should also be considered are the handling of factors that do not exactly match the left member of a rule, the loop to add in order to allow the sequential application of rules. The main concern is still how is it possible to obtain the nesting of the rules ? The next section presents the first idea that was followed, but which was proven unsuccessful.

4.3 Strategy one

4.3.1 Aim

In order to visualize the effect wanted, a schema is presented on Figure 4.4 The arrow in bold represents a word or a sub-word. The rectangles

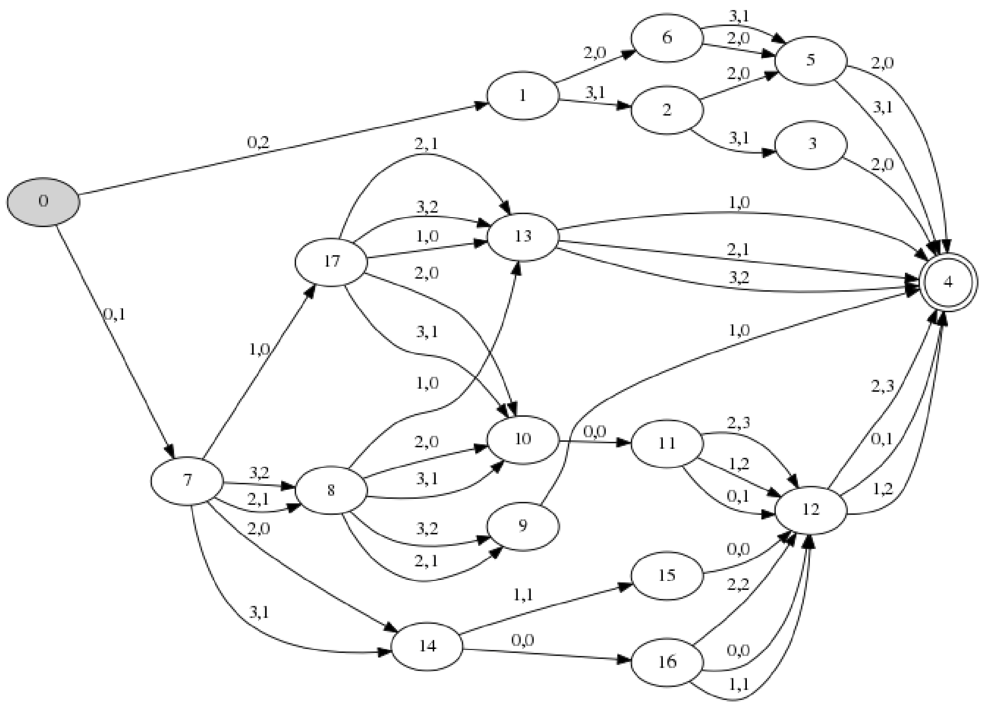

Figure 4.3: Transducer applying one rule in F_3

represent a transformation of a factor by the application of one rule of the base automaton. Rules are applied sequentially on the word, some part of the previous word transformed can be reused. There might also be some factors unchanged in the word.

The base automaton corresponds to the effect of a box. The aim is to find out how to modify it in order to obtain the correct result. The nesting of the rules implies that the ending of modified factors will be used as input to the next transformation. First this operation has to be defined, which basically is

$$\mathcal{C} = \mathcal{A}(\mathcal{B}())$$

The application of an automaton $\mathcal{A}$ to the language accepted by another automaton $\mathcal{B}$. Thus the operation of composition is used, it has been defined in Chapter 2.

The composition operation is a mean to nest rules : if the base automaton is composed with itself delayed of one position, the resulting automaton will be able to nest the two rules. The following section focuses on the automation of this nesting.

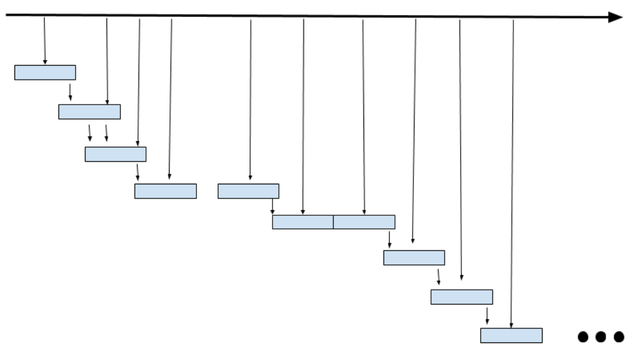

Figure 4.4: Effect on a word

4.3.2 Composition with delayed versions

The principle of this strategy is to use composition of the base automaton with itself delayed. Several approaches were tried to obtain a satisfying result. In any case, a factor could stay unchanged and the rules could several times as well. The first step is thus to add an identity part to the automaton, and make it a loop. The loop is generated by adding a transition labeled with ε from each accepting state to the initial one. Then the automaton is determinized. The addition is then added by adding transitions from the initial state to itself, labeled by “i/i”, for each $i \in [0, 3]$. One can see the resulting automaton for F_2 on Figure 4.5.

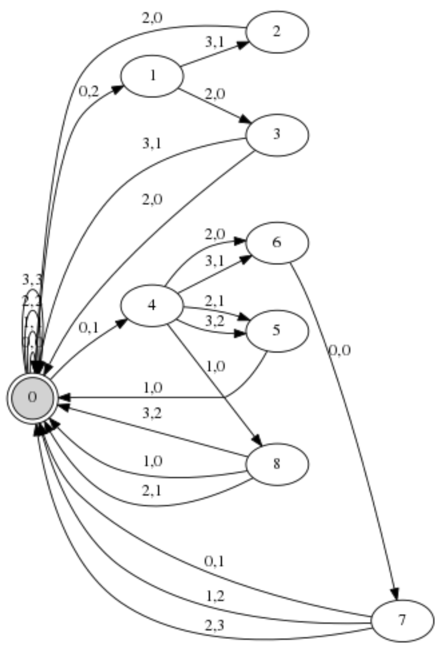

Figure 4.5: Base automaton with identity and loop for F_2

Approach 1

The first idea is thus to compose the previous automaton with itself delayed. If the visualization of the effect aimed is presented in the same representation as in Figure 4.4 : On the figure 4.6, there are 3 delayed versions of the base

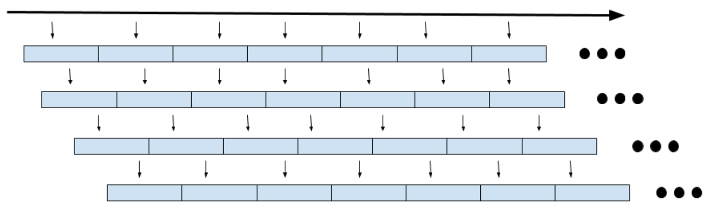

Figure 4.6: Effect of the first approach

automaton modified as in Figure 4.5. Each horizontal block of rectangles is an application of the base automaton. It is composed with itself delayed by one position, then with itself delayed with two positions, etc. The nesting of the rules is generated artificially by these compositions.

In order to cover all the nestings possible, the length of the rules is considered : $|rules| \in [m+1, 2m]$. The nesting of a rule is only possible if in the previous transformation, a '0' was created as all rules begin with '0'. But since the length of every application vary, there is no point in defining which delays are relevant except for the beginning of the word. The naive approach is to compose the automaton with its 2m-1 delayed versions which covers all possible nestings.

Problems The transducer that can be obtained with this strategy has not exactly the same effect as the transducer obtained in Section 3.2. Indeed, the order in which the transformations are made is different, it is not sequential anymore. It is not certain that the properties showed by Frougny are still holding here, namely the guarantee that any word is normalized in m applications of the transducer. It is also difficult to assess the relevance of the composition with delayed versions as the blocks vary in size from $[0, 2m]$. This approach might be very inefficient because if there is a word without any zero like $w = 0022222$ the loop in the base automaton is useless. Indeed a "real" nesting is needed to be able to reduce the word, as there is a composition with $2m-1$ automata, if there are more than $(2m-1)m$ factors strictly greater than 1^m i.e. for F_2 '22'/'21'/'12', the word won't be on the good alphabet. An example is shown in F_2 :

$$0012121212212212 \rightarrow 01011212212212$$

$$01011212212212 \rightarrow 01011212212212$$

$$01011212212212 \rightarrow 01100212212212$$

$$01100212212212 \rightarrow 01101102212212$$

The result obtained after the first pass through the transducer is thus 01101102212212. Note that there is a composition that was useless, the automaton delayed once could not work because it read "101" as a first factor, which does not match any rule. Then the second application of such a transducer would give :

$$01101102212212 \rightarrow 10010020012212$$

$$10010020012212 \rightarrow 10010100112212$$

$$10010100112212 \rightarrow 10010101002212$$

$$10010101002212 \rightarrow 10010101020012$$

As a result, the word is indeed not in the desired form. So the approach is not valid, and a new method should be investigated.

This method was still implemented, and even if it had worked, it would not have been usable due to a prohibitive computation time. Indeed, the composition is a rather heavy operation. In the case of F_2, three compositions of the base automaton is performed. The result of the intersection of two automaton $A_1 = \{\Sigma_1, Q_1, \delta_1, i_1, F_1\}$ and $A_2 = \{\Sigma_2, Q_2, \delta_2, i_2, F_2\}$ is a new automaton which has at most $Q_1 \times Q_2$ states. So as a result of the composition, one might obtain an automaton with thousands of states, whilst the final addition automaton in that case has only 15 states... One can certainly hope for a better efficiency.

For these reasons, this approach is not relevant, since the program generating the transducer for extended normalization would not terminate in reasonable time.

Approach 2

The problem with the last procedure is that the nesting is not performed as it should be. The order in which the rules are applied should be strictly sequential. In the previous approach, if a nesting was possible, it would be done at a following application of the delayed base automaton and there might be some transformations further in the word performed first. This is something that will be avoided in the following approaches. If a nesting is possible, it should be used directly. The next idea is then to compose the base automaton, but the version without the loop. This approach consists in composing the basic automaton applying one rule with itself delayed once then twice,etc in fact this would yield the the effect searched, but the number of compositions would be unbounded and would only depend on the length of the input word. It would be ideal to find an automation for this in order to have a fixed number of compositions. If one considers a rule application, all the nestings involving this rule are covered by composing it with its 2m-1 delayed versions. A first transducer can be obtained with this composition, then a loop is added to cover all the possible lengths. The effect is represented in blue on Figure 4.7.

One block composed of 2m sequential applications acts on factors of length $4m - 1$. The problem is that there is no nesting between two blocks. With the current approach, we consider only the nestings possible with the rule application on the first line. The last rule applied in a block will not have its possible nesting considered. As a result, there is a cut in the rule nesting. Another idea is to compose this first automaton to a second one which is basically itself delayed $2m$ times. Now the nesting is guaranteed on factors

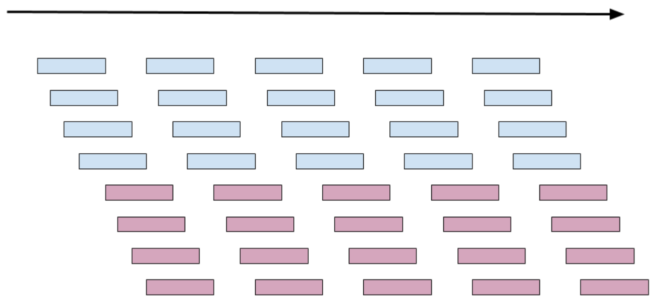

Figure 4.7: Effect of the second approach

of length $8m - 1$. But the problem remains, there is always a last rule application composed from which no nesting is considered.

Results This version without loop has been developed, and it was tested on words of limited length $[0, 4m - 1]$ for the blue version and $[0, 8m - 1]$ for the purple one. It resulted that the automaton resulting accepts a language that includes the language of the addition on these limited words. But this result is lost on longer words, because with the loop, there will always be a cut in the sequence of overlappings as explained above. This solution does not work either.

Even if it had worked, the number of compositions would still have been a heavy problem. A block automaton might have about 60000 states, thus the composition of the automaton which effect is in blue with the one in purple is really long, again for the simplest case which should result in an automaton of 15 states.

4.3.3 Consequence

The idea of composition is unsuccessful to provide the same effect as Frougny's with those two approaches. Moreover one can observe that a solution that heavily relies on composing transducers can be computationally costly. In the next section, a new way is developed to avoid these heavy compositions, such that the final automaton performing extended normalization has as much states as the base automaton seen on Figure 4.5.

4.4 Strategy two

The approach used here was already used in an intuitive way for normalization, here it is generalized. The new method explained does not use composition, it modifies the base automaton in order to allow the overlapping of the rules.

4.4.1 New method to nest rule applications

First, we observe that if there exists an overlapping, there will be an output factor that will be used as an input later. So there is a path going to the initial state and another path going out of the initial state such that the output of the incoming path and the input of the outgoing path both match. The idea is then to bypass the initial state. A new transition is created from the first state of the incoming path to the end state of the outgoing path, labeled with the input of the incoming path and the output of the outgoing path.

As an example, we can observe the following automaton on Figure 4.8. There is a path from state 3 to the initial state 1, labeled with "0/1" and there is another path from the initial state to the state 2, labeled with "1/0". The output of the first path is equal to the input of the second path, so a merging of these transitions is possible. One creates thus a new transition (in bold and red) going from state 3 to state 2, labeled with "0/0" as the ones have been consumed.

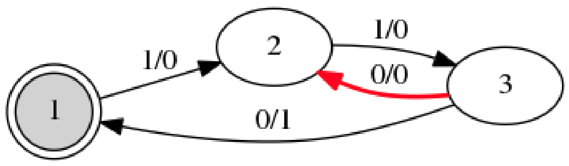

Figure 4.8: Example of merging

From the form of the rules, it is certain that a zero is needed as the output of the first transition of the incoming path and one can also deduce that there is no point in looking for a path longer than m transitions.

1. For reductions, the length of the factor which is decreased and thus which contains zeros is equal to m.

2. For unfoldings, the m last digits contain the middle factor which contains also a zero.

So first all the interesting paths should be found, then the next step is to compare the output of incoming paths with the input of the outgoing ones, and for those who match, create the transition bypassing the initial state. Note that these transitions are just added, there is no deletion of the initial paths. Thus non-determinism is brought again. But as there is one correct way to bring the word to the initial alphabet, the non-determinism will be discarded at the next step, when the output is normalized.

4.4.2 Results

The automaton obtained with the new operation on the base automaton with a loop is shown on Figure 4.9. The result is really satisfying as the computation time is significantly decreased. And the effect is the one aimed : rule applications can now be nested and the transformation is sequential from left to right. Thus the properties of the Frougny's transducer will certainly be holding in this case too.

The part to handle the case of factor ending a word which do not match a rule and that are still containing '2' or '3' is still missing. This problem, called post normalization, will be studied in the next chapter. The full automaton performing extended normalization is thus the composition of the automaton created in this chapter and the next one. The resulting automaton will then need to be composed $(m-1)$ times with itself as explained in the paper [Fro99].

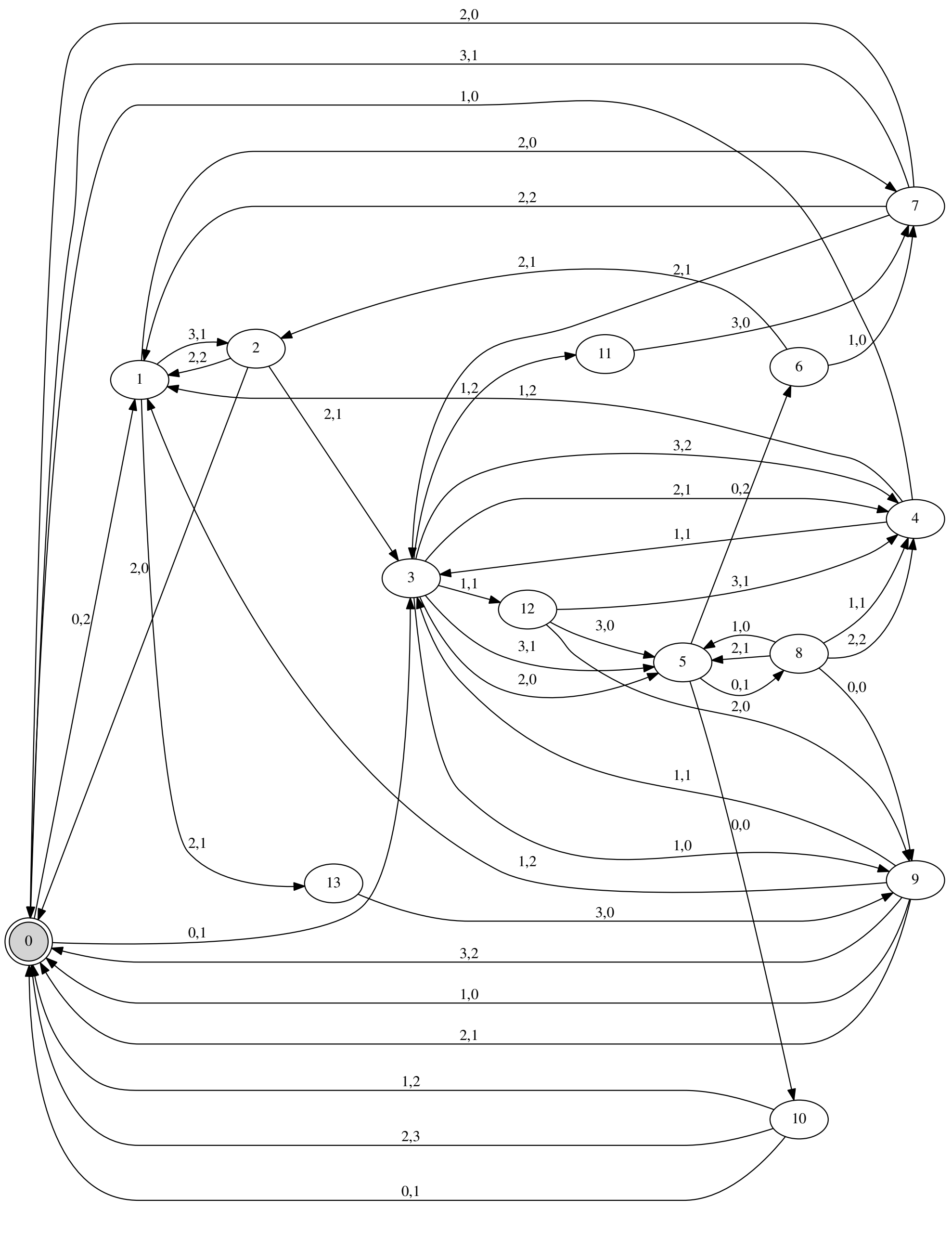

Figure 4.9: Nested rules

Chapter 5

Post normalization operations

The core problem of this work has been solved in the previous chapter. This chapter develops the final steps needed to create the addition automaton.

- First the extended normalization is not complete, as said earlier, some transformation may still need to be applied to the last digits of number encodings. The first section of this chapter focuses on the solution to this problem, the post normalization.
- Finally the automaton performing normalization is created.

Eventually, the composition of all the automatons generated is explained, the result of this last step is the addition automaton.

5.1 Post normalization

Post normalization is needed because the input alphabet might be increased during the extended normalization. The extended normalization takes as input words on the doubled alphabet $[0, 2]$. But the extended normalization uses unfoldings rules which can produce words on $[0, 3]$. For example the rule in F_2: $0202 \to 1003$. The aim is to obtain a reduced alphabet, so if a word finishes by a factor matching the left member of the previous rule, a further transformation must be applied in order to transform the last '3'. The output of the extended normalization is on $[0, 2]$, as the transducer is applied m times, some factors containing '2' could remain. The number of '2' is of course reduced after the first transformation, but it is possible to have still a few remaining occurrences left.

This part differs from Frougny's approach by not forcing $U_0...U_{m-1}$ to be powers of 2. This method was developed when trying to adapt the transducer described in Section 3.2. The first method starts from the states defined there, keep in mind that these states are the remainder of the factor being treated. If one reach the end of the word in a certain state, the label of the state is the current remainder. If the label is two digits long, the output should be two digits long too, preferably on the final output alphabet $\mathcal{A} = [0, 1]$.

First the initial approach developed for Frougny's transducer is explained and then its adaptation to its final use is presented.

5.1.1 Initial output fonction

For each state q, the output value is defined. If q belongs to the reduced alphabet, then the output value is q.

Otherwise, q has to be normalized knowing the $m-1$ first numbers of the numeration system $U_{m-1}..U_0$.

The goal is to normalize q or if it is not possible, to ease the use of a rule after. For example : if the final state is labelled with "004" in F_2, it is preferred to transform it to "012" than "020", in that way, the use of a reduction rule is allowed directly at the next step. So a little twist to the greedy algorithm shown in the Chapter 2 is applied. Knowing the first values of the base, these are only used to normalize the subword $q_{m-1}..q_0$.

Expansion The value $i = q_{m-1}..q_0$ of the subword is computed , the greatest value representable on $m-1$ digits whilst respecting the alphabet aimed is denoted by s : $s = \sum_{k=0}^{m-1} U_k$. If $i > s$, the Euclidean division is used : $i = ns + r$. Each value q_j of the subword will be initialized to n. Then r will be normalized using the common greedy algorithm, and the resulting word will be added.

Then the full word q is considered, with its ending subword transformed. If a rule can now be applied, it is applied and if the ending subword is still not normalized, the expansion step can be performed again and so on until either no rule is applicable, or until the word is on the final alphabet.

5.1.2 Why is this also working?

First the form of the states that are normalized is considered. States are basically suffixes of left members of rules. States that are to be normalized computed from reductions are in the form :

$$S = \{0(i_1+1)...(i_k+1) \mid 0 \leq i_1, ..i_{m-1} \leq 2, k \leq m-1\}$$

with at least one of the $i_k \geq 2$ As $|s| \leq m$ there is no rule applicable for this case. The assumption of a complete system on the chosen alphabet is used here. If the first values are correctly chosen, then there will be a way to normalize s or at least to transform the first zero, which would lead to a decrease of the output alphabet, or to the possibility to apply a rule later.

Now the set of states derived from unfoldings are considered :

$$S' = \{\ 0(i_1+1)...(i_{j-1}+1)(i_j+1)\nu_{j+1}...\nu_{j+m-1}\ \} \tag{5.1}$$

with $1 \leq j \leq m-1$, $0 \leq i_1, ..i_{m-1} \leq 2$,$\nu_{j+1}...\nu_{j+m-1} \in [0,2]$ and $\nu_{j+1}...\nu_m <_{lex} 1^{m-j-1}1$. The structure of the states guarantees that there is at least one zero in the ending of the state, and also $i_j + 1 \geq 2$ belongs to the ending of the word. If one considers the number represented on the m last letters of the label of the state :

Either it is greater than $\sum_{k=0}^{m-1} U_k$, then the twisted version of the greedy algorithm can be used to ensure that there is no more zero in that part of the word, thus allowing a reduction to be used later.

Or if that number is smaller than $\sum_{k=0}^{m-1} U_k$, from the assumption of the completeness of the numeration system on the chosen alphabet, it is certain that the number will be normalizable with the greedy algorithm.

It is also sufficient to say that a normalization rule can be applied later. This is the case when there is a factor $w \geq 1^m$. So when it will go through the next pass of in the transducer, if there is an ending factor $0w$, a rule is applied directly, or if there is an ending factor $0vw$ with $v \geq 1^\ell, 1 \leq \ell$

- If $\ell \geq m$, then as the strategy is to reduce as much possible, at least one zero will be created in the factor v : one will then consider the subword starting from the right-most zero created in the factor v: $0v'w$ if $|v'| \geq m$ the same operation is repeated until a zero is created such that the new remainder $|v'| < m$ and then w will indeed be reduced at the next rule application.

- If $\ell < m$ then w is directly reduced. There may be several sequential application of rules.

5.1.3 Adaptation to the new approach

As explained at the end of the Chapter 3.2, the transducer can not be translated in a DFA because it is not synchronous. The method created for this step is still used in the final version. As states represent all possible remainders left to treat, they are used to treat the ending of the word. Every ending of a word which can not be reduced corresponds thus to one of the states defined by Frougny. The approach chosen is then to create a synchronous transducer, which reads the state and output its output value.

The concept behind this transducer is really simple : for each state q not in $[0, 1]$, its label "ℓ" and its output value "o" are read synchronously digit by digit. Then for each $i < |q|$ a new transition is created with labels "ℓ_i, o_i" and this new transition is directed towards a freshly created state, from which the next transition $i + 1$ will be generated if $i < |q| - 1$, if it is not the case,the last state is only made accepting. The automaton should be able to read words longer than $2m$, so the initial state should be accepting and there is a self-loop reading and outputting "j, j" with $j \in [0, 2]$

The resulting automaton is then minimized to obtain a more compact automaton. The result is a automaton performing extended normalization in the special case of words or factors smaller than $2m$. It must be composed with the extended normalization automaton presented earlier to form the complete extended normalization automaton.

5.2 Normalization

As explained in [Fro99], normalization can be obtained through one pass in two transducers. These transducers just apply the transformation

$$\rho = \{01^m \to 10^m\}$$

repeated as many times as possible on a word. The transformation is applied in a sequential way, from right to left and from left to right, which gives the two transducers. The right to left transducer is noted $\mathcal{R}$ and the left to right transducer is noted $\mathcal{L}$.

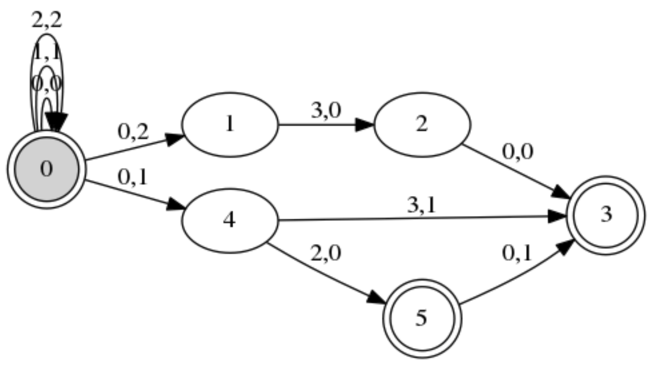

Figure 5.1: F_2's transducer for post normalization

Note that normalization is applied to words on the final alphabet and should produce words on that alphabet. So if there is a word in F_2 $w = 1111$, one should add one '0' to allow the normalization $01111 \to 10100$

For each of these transducers, there is a worst case that would need a number y of applications of the transducer alone, y depending only on the input word.

Worst case for a transducer $\mathcal{R}$

The worst case is a word of the form 01^{mn}. There are n reductions possible, but the transducer can only apply the last one, some example of execution of the transducer would give on a word $w = 0111111$ represented in F_2 :

$$0111111 \to 1001111$$

$$1001111 \to 1010011$$

$$1010011 \to 1010100$$

Three passes in the transducer give the final result. Note that here $n = 3$, so there will be n passes to obtain the word properly normalized. This result shows that this transducer is not fitted for normalization.

Worst case for a transducer $\mathcal{L}$

Similarly, the worst case for the left to right transducer is a word in the form : $0(1^{m-1}0)^n 1^m$. At each application, the transducer can only transform again the last part of the read word. An example of some executions on the word $w = 0101011$:

$$0101011 \rightarrow 0101100$$
$$0101100 \rightarrow 0110000$$
$$0110000 \rightarrow 1000000$$

Once again, one can observe that the transducer should be applied to the word n+1 times. There is again no limit on the number of passes needed to normalize the word.

It is also worth noticing that the worst case of each transducer is perfectly handled by the other one. Which gives an insight on why they should be applied both on a word to obtain the best result.

The theoretical statements claim that the transducers have thus to be composed in any order. In the development, a limitation was found : the transducer which is applied last must be slightly transformed. This is caused by the fact that the final transducer should prevent the acceptation of 1^m factors in the result. So as a result, the last transducer should be modified to prevent that, but the first one should not.

5.2.1 Construction of the transducers

As stated before, the aim is to achieve : two transducers applying sequentially the reduction rule $\{011 \rightarrow 100\}$. A start is thus a transducer applying the rule. The two transducers are showed on Figures 5.2 and 5.3 for F_2.

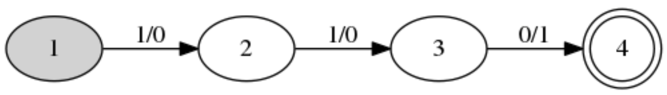

Figure 5.2: Simplest transducer normalizing once from right to left

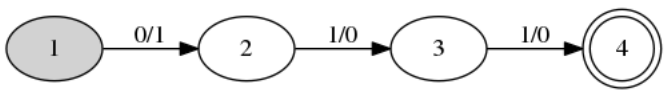

Figure 5.3: Simplest transducer normalizing once from left to right

Then it should be possible to repeat this transformation. A first step is to merge states 4 and 1 on Figures 5.2 and 5.3, i.e to make the initial state the accepting one. That way the transformation can be applied several times. Now cases like

$$011011 \to 100100$$

are correctly handled. But it would also be interesting to allow nested transformations :

$$01011 \to 1000$$

$$01111 \to 10100$$

the first transformation is nested for the transducer$\mathcal{R}$ and the second one is for the transducer $\mathcal{L}$.

In the case of the transducer $\mathcal{R}$, the '1' created can be consumed in a further transformation. Similarly, for the transducer $\mathcal{L}$, the last '0' created can become a '1' in the next transformation. The incoming and the outgoing transitions from the initial and final state are merged such that a new transition going from the penultimate state to the second one is created. The labels must also be merged : for the transducer $\mathcal{R}$ the labels "0/1" and "1/0" give the fresh label "0/0" and for the $\mathcal{L}$ one : "1/0" and "0/1" give "1/1". The merging is possible only because the output of the incoming transition and the input of the outgoing one are equal. The result can be seen in the F_2 case on Figures 5.4 and 5.5. The transitions resulting of the merging are in red and in bold.

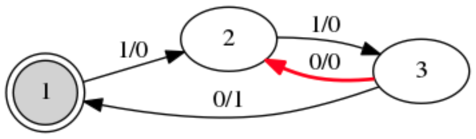

Figure 5.4: Transducer normalizing from right to left

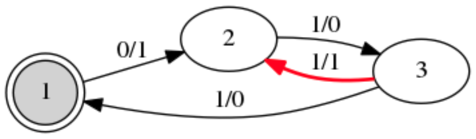

Figure 5.5: Transducer normalizing from left to right

The next step is to reverse the transducer $\mathcal{R}$, as the addition transducer will then be left sequential, it will read words from left to right. Then it would be interesting to accept factors that are on the proper alphabet, but that do not need to be normalized. The result of these operations is shown on Figures 5.6 and 5.7.

For the right transducer, it is important to reverse it first, then to add the possibility to accept factors that do not need normalization. There are two reasons for that :

- The states added for that part should be accepting, which would turn into m initial states after reversion.
- The transitions are returned after reversion, so as the idea was to lead to an identity state on '1', and to return to the initial state on '0', there would always be a "0/0" transition to make the transition "1/1" reachable.

Note that on the transducer $\mathcal{L}_2$, the generation of "11" factors was allowed. The transducer $\mathcal{L}$ is composed with the $\mathcal{R}$ one, so these factors will be eliminated further.

Finally the resulting transducer is shown on Figure 5.8.

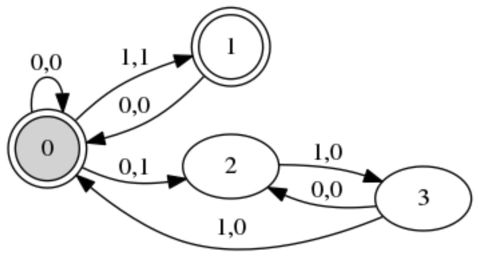

Figure 5.6: Transducer normalizing from right to left

5.3 Addition automaton

All the building blocks needed to generate the addition automaton have been detailed. In this section, the final assembling is explained in order to generate the final addition automaton.

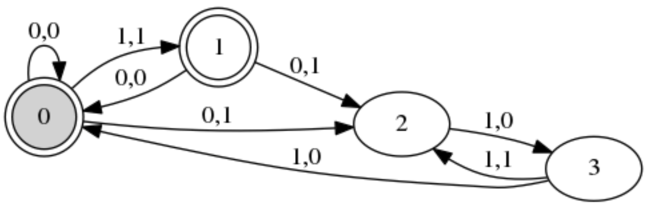

Figure 5.7: Transducer normalizing from left to right

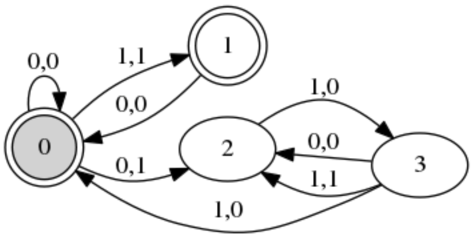

Figure 5.8: Transducer performing normalization in F_2

First as explained earlier, the complete extended normalization automaton is obtained by computing the composition of the automaton defined in Chapter 4 and the automaton defined in Section 5.1. As Frougny states, the resulting automaton should be then composed with itself m times perform the extended normalization.

The final step is to compose the automaton of addition bit by bit with the automaton resulting from the previous manipulation, and after the result of this composition is composed with the normalization automaton.

Chapter 6

Implementation

In this chapter, some implementation details are discussed. The results of the Chapters 4 and 5 were implemented using some parts of the Frougny's method. First we present the implementation of these parts of Frougny's method, then the rest of the implementation is discussed.

The implementation was made in C, useful for its low level efficiency which was really needed for the heavy computations on automata, and also it allowed to use the LASH library [Boi]. We used the core package in order to create automata and to operate on them such as intersection, product, union, minimization etc.

6.1 Frougny's algorithm

All the data needed to create the transducer has been generated : the rules, the states and the transitions. Only the transitions are not needed in the final implementation of the work. The code generated follows closely the algorithm explained in the Chapter 3.2 so the implementation is explained with a high level idea, to present generally the structure of the code created for this part.

Pattern

In order to represent labels, states, rules, etc.. One simple way to represent arrays was needed. It is the purpose of the structure pattern, which is an array of integers dynamically allocated, stored with its size.

A lot of different operations should be provided to operate on them :

- All operations related to prefixes and suffixes, to retrieve all the prefixes of a pattern in order to create the state space for example, all the suffixes, or to retrieve one suffix or one prefix.

- Operations to get the minimum and the maximum of a pattern, useful for post normalization.

- An operation to increment the pattern, which is used for generating the rules.

Rules

A rule is a new type which contains two patterns, a pattern for the left member and one for the right member. Rules are stored in an hashtable, the hashing function takes as input the left member of the rule, in order to retrieve easily the possible transformations for a certain pattern. In that way, when there is a factor that needs to be transformed, feeding it to the hashing function gives the possible transformations.

States

A state is characterized by its label and its output value. Both are represented by pattern instances. States are also saved in a hashtable, ordered by label. States are reused in the post normalization part.

Transitions

A transition is represented by a starting state and a destination state, an input which is always only a digit, and an output which could be longer than one digit. The states and the output are thus represented by patterns, for the states the pattern is the label of the state.

HashTable

A simple hashtable was implemented, its hashing function operates on patterns which is useful as patterns are used in every data structure. The hashtable is implemented with separate chaining, so each element of a table is a linked list.

Generation of the transducer

1. First the rules are implemented by following strictly their definition given by equation 3.2 and 3.1, except that as explained earlier for re-

ductions, if there is a possibility to reduce more a factor than by one, it is done.

2. While the rules are being generated, the state space is also build: from each left member of a rule, all the possible prefixes are extracted and if there is one that is not already the label of an existing state, then a new state with that label is added. The output value of each state is computed later.
3. Then for each state, each possible input is considered and the correct transition for that input is created according to the method described.
4. Finally, the output value for each state is computed with the method described in Section 5.1.

All the code for these steps is located in `pattern.c`, `genData.c` and `hashtable_r.c` .

6.2 Implementation of the new approach

The implementation of the main steps of the final method described in Chapter 4 is discussed here.

- First the implementation of the base automaton is explained.
- After that, the implementation of composition is discussed,
- Finally the implementation of the new operation allowing the nesting of the rules is explained.

6.2.1 Base automaton

As explained in its theoretical description, the base automaton is actually the union of two automata. The construction of those is worth explaining. First the implementations of the reduction automaton is explained, and then the unfolding is discussed.

Reduction automaton

The implemented algorithm is recursive, following from observations made earlier.

In the gen_red function, the algorithm creates the initial states and its outgoing transitions going to new states corresponding to the R_1 and R_2 sets. Then the rest of the automaton is build by calls to the recursive function red_rec. Note that the algorithm would also work in the case of a larger alphabet.

```
automaton get_red(int alph_max, int m)
{
    automaton a = new_automaton();
    int init = auto_add_new_i_state(a);
    for(int i = 1; i < 2*alph_max, i++)
    {
        int new_state = auto_add_state(a);
        auto_add_transition(a, init, new_state, "i/0");
        red_rec(a, new_state, i , false, 2*alph_max, m);
    }
    return a;
}
```

Listing 1: Algorithm to generate the reduction automaton

The red_rec function takes as arguments :

- a : the automaton
- curr_state :the current state from which the automaton is currently being build,
- to_red : the number of reductions that are to be apply, here it is 1 or 2, but it could be more with an alphabet expansion
- is_restricted : a boolean to remind whether a restricting digit has been created earlier
- alph_max : the upper bound of the doubled alphabet
- tr_left : the number of transition still needing to be build

The algorithm is not too complicated : if there is no more transitions to create, nothing is done. Otherwise there are two cases :

- Either a transition with a restricting digit ahs already been created, then transitions reading number from to_red to alph_max+1 and outputting the read value minus to_red are created.

- Either there is no restricting digit yes, so it can be created now or later if this is not the last transition that must be created.
  - If the restricting transition is created, just one transition labeled with `to_red` as input and zero as output is added, then the recursive call takes place with `true` as the new value of `is_restricted`.
  - Otherwise it can be left for later and the behaviour is the same as previously in the first case considered, creating many transitions to another new state.

```
void red_rec(automaton a, int curr_state, int to_red, bool is_restricted,
                                    int alph_max, int tr_left)
{
    if(tr_left == 0)
        return;

    int new_state = auto_add_state(a);
    if(!is_restricted)
    {
        auto_add_transition(a, curr_state, new_state, "to_red /0");
        red_rec(a,new_state, to_red, true, alph_max, tr_left -1);

        if(tr_left != 1 )
        {
            int new_state2 = auto_add_state}(a);
            for(int i = to_red+1; i <= alph_max + 1; i++)
                auto_add_transition(a, curr_state, new_state2,
                                                        "i/i-to_red");

            red_rec(a, new_state2, to_red, false, alph_max, tr_left-1);
    }
    else
    {
        for(int i = to_red; i <= alph_max  + 1; i++)
            auto_add_transition(a, curr_state, new_state,"i/i-to_red");

        red_rec(a, new_state, to_red, is_restricted, alph_max, tr_left-1);
    }
    if(tr_left == 1)
        auto_mark_accepting(a, new_state);


}
```

Listing 2: Recursive algorithm to construct the reduction automaton

The theoretical definition of the rules that was implemented in the first version was not used. Note that if one wanted to expand the alphabet, the algorithms would not have to change. The aim was to find an algorithm to generate this automaton in an efficient way. Another way to do it could be to generate all the reduction rules and then add each of them as a sequence of transitions and states to the automaton, and then to trust the minimizing algorithm provided by LASH to obtain the same result. This solution was not implemented, as the number of rules grows fast, the option to save memory and computing time was preferred.

6.2.2 Unfolding automaton

The algorithm used to generate the unfolding automaton is again a recursive one.

The first function is simple : the two first states (the initial one and all possible second states) are created, the first transition is always the same, then for each length possible, there is a recursive call to construct the rest of the automaton. Again, there is a fixed number of transitions to generate in the recursive function. A rule application just being a path from the initial state to an accepting one. In our case, `alph_max` is always equal to 1.

```
automaton get_un(int alph_max, int m)
{
    automaton a = new_automaton();
    int init = auto_add_new_i_state}(a);
    int second = auto_add_new_state}(a);
    auto_add_transition(a, init, second, "0/1");
    for (int j = 1; j < m ; j++)
        un_rec(a, second, j,1, 2*alph_max, m,false);

    return a;
}
```

Listing 3: Algorithm to generate unfolding automaton

The second function is quite longer. The arguments are quickly described:

- a : the automaton
- state : the index of the state the function is working on
- j : the value j, the index of the digit which will be reduced by 2.
- curr_tr : the number of transitions already created
- alph_max : the upper bound of the doubled alphabet
- m : the degree of the Fibonacci polynomial characterizing the numeration system

- restr: a Boolean to know whether the middle factor (which is left unchanged) has been limited, it already satisfies the condition $<_{lex} 1^{m-j}$

The next action depends on the number of transitions already created. Knowing that there are a total of $m + j + 1$ transitions to be generated. If they are all generated, the recursive call is ended, otherwise, if the last transition is being generated and thus the last state as well, the last one is marked as accepting. Then from the current position in this path, there are four parts to be distinguished in every rule application :

1. If $curr\_tr < j$, then all the different transitions possible are created such that each of them read an input number between $[1, 3]$ and outputs this number decremented. All transitions go from the current state to the new one.

2. If $curr\_tr = j$, then the input number will be decreased by 2. A new state is created, and new transition are created with labels reading as inputs numbers $\in [2, 3]$ and outputting the value minus 2.

3. If $j < curr\_tr \leq m$, then the current path is in the middle factor that will stay unchanged. There is a condition to fulfill. Either it is already done, and then `restr` is true, and the transitions' label read any input in $[0, 2]$ and output the same value.

   Otherwise, the condition can be satisfied now by adding a transition labelled by "0/0".If it is not the last term of the middle factor, the choice to satisfy the condition later can be made. Then a second new state is created and a transition towards it is added with label "1/1".

4. If $m < curr\_tr \leq j+m$, then any input is read in the doubled alphabet and the output value is the input value incremented.

```
void un_rec(automaton a, int state, int j,int curr_tr,int alph_max,
                                                 int m, bool restr)
{
    if(curr_tr == m+j+1)
        return;

    int new_state = auto_add_state}(a);

    if(curr_tr == order + j)
        auto_mark_accepting(a, new_state);

    if(curr_tr == j)
    {
        for( int i = 2; i <= alph_max+1; i++)
            auto_add_transition(a, state, new_state, "i/i-2");

        un_rec(a, new_state, j, curr_tr+1, alph_max, m, false);
    }
    else if(curr_tr < j)
    {
        for(int i = 1; i <= alph_max+1; i++)
            auto_add_transition(a, state, new_state, "i/i-1");

        un_rec(a, new_state, j, curr_tr+1, alph_max, m, false);
    }
    else if(curr_tr > j && curr_tr <= m)
    {
        if(restr)
        {
            for(int i = 0; i <= alph_max; i++)
                auto_add_transition(a, state, new_state, "i/i");

            un_rec(a, new_state, j, curr_tr+1, alph_max, m, restr);
        }
        else
        {
            auto_add_transition(a, state, new_state, "0/0");
            un_rec(a, new_state, j, curr_tr+1, alph_max, m, true);
            if(curr_tr != m}
            {
                int new_state2 =auto_add_state}(a);
                auto_add_transition(a, state, new_state2, "1/1");
                un_rec(a, new_state, j, curr_tr+1, alph_max, m, restr);
            }
        }
    }
    else
    {
        for(int i = 0 ; i <= alph_max; i++)
            auto_add_transition}(a, state, new_state, "i/i+1");

        un_rec(a, new_state, j, curr_tr+1, alph_max, m, restr);
    }
}
```

Listing 4: Recursive algorithm to generate unfolding automaton

This implementation relied heavily on the definition of the unfolding rules.

6.2.3 Composition

The implementation of the composition is performed in `composition.c`. First the algorithm to expand the alphabet of an automaton is presented. The arguments are explained :

- the automaton to modify.

- the position at the new component should be, either at the beginning or at the end.

- the number of components per transition of automaton a.

- the alphabet of the new component.

```
void add_composant}(automaton a, int position, int ln, int alph)
{
    automaton uni = auto_new_uni(alph);
    if(position == 0)
        a = auto_product(a, uni, ln, 1);
    else
        a = auto_product(uni, a, 1, ln);
}
```

Listing 5: Algorithm to expand one automaton's alphabet

The algorithm proceeds by building an automaton accepting anything on the aimed alphabet `alph`. This step is summarized in the pseudo-code. Then if there should be a new component at the beginning of each transition, the following product is made : $\mathcal{U} \times \mathcal{A}$ otherwise it is $\mathcal{A} \times \mathcal{U}$

The composition algorithm follows the theoretical explanation given before, except that it allows to compose automaton of different forms. One example of usage is the composition of the bit by bit addition and the extended normalization. The addition automaton has two input tapes and one output, while the extended normalization automaton has one tape for input and one for output. So the aim is to keep just the input tapes of the addition and the output tape of the extended normalization. There thus some extra parameters to allow such changes.

There are the following arguments :

- Each automaton and the length of its transitions labels.

- The alphabet of the new components, here it is always the same, but it could be different, that case is not handled

- The length of the intermediary automaton (before projection), such that the number of components to be added to each automaton is known.

```
automaton compose(automaton a, int ln_a,automaton b,int ln_b,
                                                int alph, int ln)
{
    while(ln_a < ln)
        add_composant(a,ln_a, ln_a, alph);

    while(ln_b < ln)
        add_composant(b,0, ln_b alph);

    automaton c = auto_intersection(a, b);
    int to_project = (ln_a+ ln_b)-ln;
        if(ln == ln_a || ln == ln_b)
                to_project = 0;
        for(int i = 0, l = ln ; i < to_project, l > 0; i ++, l--)
                result = auto_projection(result, l, ln_a -1 -i);
        return} c;

 }
```

Listing 6: Composition algorithm

The core algorithm is as simple as this: first the addition of new components to each automaton is performed, then the intersection is computed. The number of projections to perform is then computed and the projection starts from the length of the first automaton to the left.

6.2.4 Nested rules

For the nesting of the rules, an approach was developed in Chapter 4. This approach consisted in merging the incoming paths to the initial path with its outgoing paths, if their label match, i.e. if the output of the incoming path is identical to the outgoing path. The merging results in a new transition going from the initial state from the incoming path to the end state of the outgoing path. This transition is labeled with the input of the incoming path and the output of the outgoing path.

The implementation consists in three functions located in `composition.c`. The two first functions collect all the incoming and outgoing paths, and the last one creates the transitions depending on the paths collected. A new data structure was introduced to represent paths.

```
typedef struct
{
    int nb_trans;
    uint1* tran_in;
    uint1* tran_out;
    int state;
    int key;
} path;
```

Listing 7: Path data structure

This data structure maintains as information the length of the path `nb_trans`, two tables of bytes to store what is read on input and what is output on the path, the state defining the path the other state always being the initial state, and its key in the hashtable.

Then the pseudo-code of the function which retrieve incoming paths is presented. It is the most complicated as it finds paths in backwards order. It starts from the initial state, and it looks for states that have an outgoing edge to it. After saving the identifiers of the edge, it starts back from that state which has such an outgoing transition. The same procedure is repeated until it generates a path of sufficient length. Then a path instance is created and stored in an hashtable. For incoming paths, the key in the hashtable depends on what is output on the tape.

First the arguments are explained :

- `a`: the automaton in which the paths are searched.
- `state_end`: The current state from which the current path is being build : it starts from the initial state, then it finds a state which has an outgoing transition to the initial state, and this state becomes the new current state in the next call.
- `left` : the number of transitions to find.
- `path` : a table with the number of the states of the states belonging to the path.
- `trans` : a table to remember the index of the outgoing transition taken along the path. So the path can be retrieved by first looking at $path[j]$ for the state, then it identifies the transition from the state stored and

the number in $tran[j]$. Transitions are identified in LASH by their starting state and one index.

- `ini_st` : The length of the path to be constructed
- `path_in` : an hashtable in which the paths found are found
- `visited` : a table to avoid cycles

```
void get_path_in(automaton a, int state_end, int left, int* path,
        int* trans,int ini_st,hash_tab* path_in, int* visited)
{
    visited[ini_st - left] = state_end;
    if(left == 0)
    {
        uint1* tab_in = malloc(ini_st/2);
        uint1* tab_out = malloc(ini_st/2);
        int rev = ini_st/2 -1;
        for(int i = 0 ; i < ini_st; i++)
        {
                transition t = auto_transition(a, path[i], trans[i]);

                if(i%2)
                    tab_in[rev-i/2] = auto_transition_label_ptr(t,1);

                else
                    tab_out[rev-i/2] = auto_transition_label_ptr(t,1);

        }
        int key = hash(tab_out);
        path p = create_path(ini_st,tab_in, tab_out, state_end, key);
        insert_entry(path_in,p, key);
        return;
    }
    for(each state s in a )
    {
        if(s in visited)
            continue;
        int nb_out = auto_nb_out_transition(a, s);
        for(int i  = 0; i < nb_out; i++)
        {
            transition t = auto_transition(a, s, i);
            if(auto_transition_dest(t) == state_end)
                {
                    path[ini_st-left] = s;
                    trans[ini_st-left] = i;
                    get_path_in(a, s, left-1, path, trans, ini_st,
                                            path_in, visited );
                }
        }
    }
 }
```

Listing 8: Algorithm to find the incoming paths

The algorithm to retrieve the outgoing paths is of the same kind, except that the direction is more convenient : one only has to follow the outgoing transitions from the initial state. So reversing the `path` and `trans` is not neeeded to create the path's read values. It is not needed to check every state of the automaton to look for the next state in the path neither, one just checks the outgoing transitions of the current state. The arguments are totally similar.

```
void get_path_out(automaton a, int state_end, int left, int* path,
            int* trans, int ini_st, hash_tab* path_out, int visited)
{
    visited[ini_st - left] = state_end;
    if(left == 0)
    {
        uint1* tab_in = malloc(ini_st/2);
        uint1* tab_out = malloc(ini_st/2);
        for(int i = 0 ; i < ini_st; i++)
        {
            transition t = auto_transition(a, path[i], trans[i]);
            if(i%2)
                tab_in[i/2] = auto_transition_label_ptr(t,1);
            else
                tab_out[i/2] = auto_transition_label_ptr(t,1);
        }
        int key = hash(tab_in);
        path p = create_path(ini_st,tab_in, tab_out, state_end, key);
        insert_entry(path_out,p, key);
        return ;
    }
    for(each transition t in outgoing_transition(state_end))
    {
        state s = auto_transition_dest(t);
        if(s in visited)
            continue;
        path[ini_st-left] = s;
        trans[ini_st-left] = index(t);
        get_path_out(a, s, left-1, path, trans, ini_st, path_out,
                                                visited );
    }
 }
```

Listing 9: Algorithm to find the outgoing paths

Now the focus is on the merging of the outgoing and incoming paths in transitions.

```
void new_comp(automaton a, int order)
{
    hash_tab* path_in = createhash_tab();
    hash_tab* path_out = createhash_tab();
    for(int j = 1; j <= order; j++)
    {
        for(int i = 0; i < auto_nb_states(a); i++)
        {
            if(auto_accepting_state(a, i))
            {
                get_path_in(a, i, 2*j, [], [], 2*j, path_in, []);
                get_path_out(a, i, 2*j, [], [], 2*j, path_out, [] );
            }
        }
    }
    for(each path p in path_out)
    {
        if(hasKey(path_in,p->key))
        {
            for(each path q in path_in[key])
            {
                if(p->tran_in == q->tran_out)
                {

                    uint1* label = malloc(p->nb_trans);
                    for(int k = 0; k < p->nb_trans; k++)
                    {
                        if(k%2 == 0)
                            label[k] = q->tran_in[k/2];
                        else
                            label[k] = p->tran_out[k/2];
                    }
                    auto_add_new_transition(a, q->state, p->state,
                                            p->nb_trans, label);
                }
            }
        }
    }
}
```

Listing 10: New operation algorithm

As one may have seen, the interesting lengths to nest rules are contained in the interval $[1, m]$: a path should start from a zero generated in the first rule application.

- For reduction rules, the factor transformed is of size $m + 1$, the m last digits being decreased.
- For unfolding rules, we know that there is a zero in the m last digits of the factor.

So the algorithm starts by generating the paths of these lengths. Note that the lengths are doubled in the algorithm because the transitions are serialized, which means that a transition with a label "x/y" will be transformed in two sequential transition "x", then "y", and an intermediary state is added. So if a path of length n is desired, one will look for one path of length $2n$ in the implementation. The input read on the path will be represented by the transitions with index k, with $k \mod 2 = 0$ and the output with indexes k; $k \mod 2 = 1$.

Then the algorithm looks for matches between paths. The paths are stored in two hashtables, the incoming path's keys are defined by their output values, and the outgoing path's key by their input values. Of course the same hashing function was used, thus it looks for paths matching at the same index in both tables. The match is not guaranteed by the fact that two paths have the same key, as there are collisions. So if there is a real match, the sequential label of the new transition is build, and the new transition is added.

This concludes the section about implementation details. The remaining details are quite simple and do not deserve to be explained here.

Chapter 7

Conclusion

The contribution of this work is a new way to generate the addition automaton for Fibonacci numeration system. The procedure developed can be implemented using only finite deterministic automatons as all the transducers described in the work are synchronous. The results obtained are discussed here, as well as remaining problems and possible improvements.

7.1 Testing

In this section, the tests performed to test the efficiency of the method are explained. As the final result follows the same mechanism as Frougny, one could expect to obtain the same results, i.e. the exact addition automaton with the m compositions of the extended normalization automaton as it is the only point of divergence with Frougny's work. The method has been proved equal in the case of the traditional Fibonacci and Tribonacci numeration systems as the exact addition automaton was obtained.

Walnut comes with the addition automata for Fibonacci and Tribonacci. These automatons were used to first prove that the final automata obtained are included in these automata and the equality between them was proved.

From these results, one can assume that the method will also produce the correct addition automata for higher orders. Sadly, there were no other examples of other addition automata for higher orders to support this statement further.

In order to observe the efficiency of each step on words, some methods were created to watch the output values of the transducers on certain input words.

This method was mostly useful to debug the extended normalization part. This testing method used again composition to produce an automaton whose accepting path represent all possible transformation for the input word. For some steps there is more than one path as some steps are not deterministic. The final result is deterministic as only normalized words are accepted and that representation is unique.

The goal of the master thesis is thus completed, as the correctness of the method has been shown consistent in all known cases. The final addition automaton is represented in Figure 7.1.

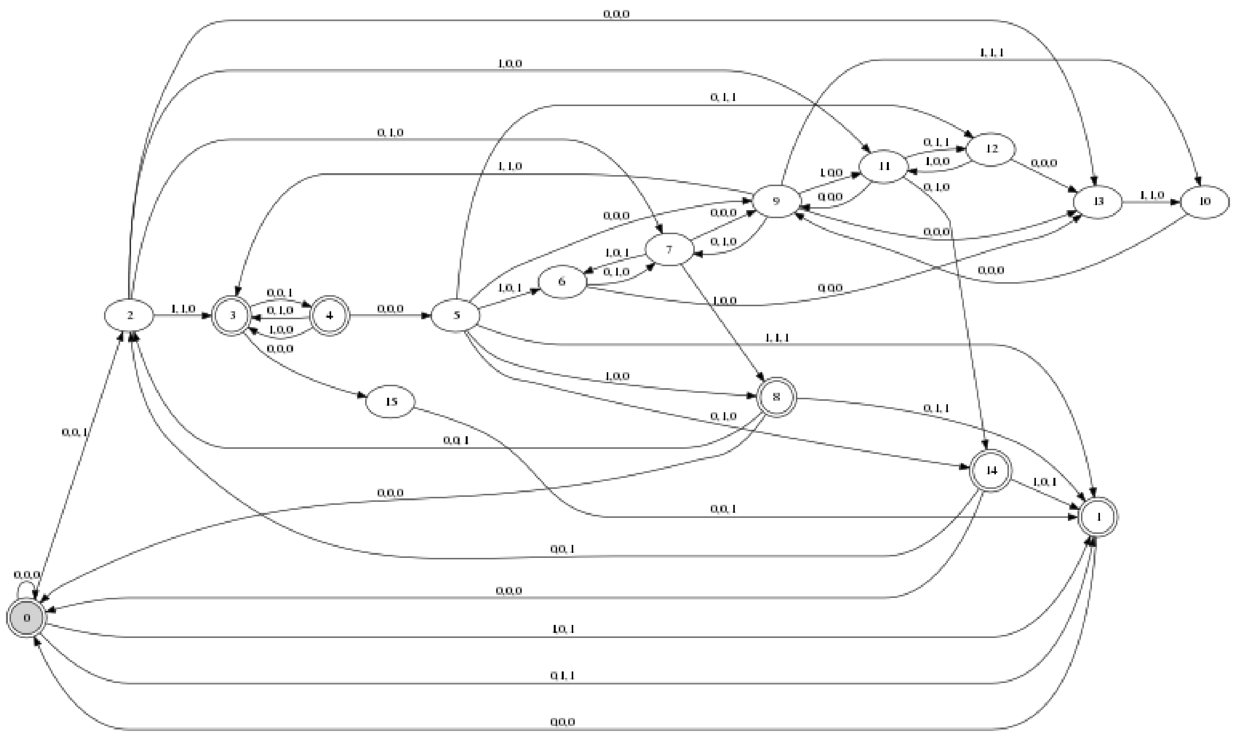

Figure 7.1: Addition automaton for F_2

There are always some points that could be improved and further work.

7.2 Possible improvement

	Fibonacci	Tribonacci
Final automaton size	47	461 (states)
Final extended normalization automaton	207	33 999 (states)
Initial extended normalization automaton	47	175(states)
Computation time	67 ms	1min49

Table 7.1: Measures of efficiency

The computation time is quite long, for Tribonacci one waits about 2 minutes before getting a result. The source of this slowness is composition. Composition of big automata is really costly. For Tribonacci, the automaton of nested rules has about 175 states. It is then composed twice with itself. The first composition results in an automaton of about 2000 states, the second one in 35000 states. Some relevant numbers about the size of the automata generated are given in the table 7.2 and the total execution time is given too.

The growth of the state space in the extended normalization step is quite spectacular but it is explained by the intersection operation. Hopefully the size of the final automaton is reduced thanks to the composition with the addition and mostly with the normalization automaton.

If there is something that should be improved, it is the creation of the extended normalization automaton. The number of composition is already decreased in comparison with the first strategy of composition with delayed automata.

7.3 Further work

One interesting point would be to extend the work to other Pisot numeration system, which are still defined by a linear recurrence relation. The reduction rules would still be convenient, but the unfoldings might be very different. To adapt the approach developed in this work, the modifications to be performed would be:

1. The generation of the reduction automaton, which is not expected to be complicated,

2. The generation of the unfolding automaton: this step will certainly be more complex according to the polynomial of the new numeration system.

3. The post normalization step : the current methods relies on the fact that there should be m non zero sequential digits to perform a reduction after. This step should be adapted according to the new recurrence relation in order to still favour the application of a reduction rule.

Moreover, we are not sure that the extended normalization could be as efficient as in this work.

Then one might want or might be forced to expand the alphabet if one wants to use other Pisot numeration systems.

List of Figures

List of Listings

Bibliography

[Hil05] D. Hilbert. "ON THE FOUNDATIONS OF LOGIC AND ARITHMETIC". In: *The Monist* 15.3 (1905), pp. 338–352. ISSN: 00269662. URL: `http://www.jstor.org/stable/27899603`.

[Göd31] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems.* Dover Publications, 1931.

[Pis46] Charles Pisot. "Répartition (mod 1) des puissances successives des nombres réels." French. In: *Comment. Math. Helv.* 19 (1946), pp. 153–160. ISSN: 0010-2571; 1420-8946/e. DOI: `10.1007/BF02565954`.

[Sta84] Ryan Stansifer. *Presburger's Article on Integer Airthmetic: Remarks and Translation.* Tech. rep. TR84-639. Cornell University, Computer Science Department, Sept. 1984. URL: `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639`.

[Fro88] Christiane Frougny. "Linear Numeration Systems of order 2". In: *Information and computation* 77 (1988), pp. 233–259.

[Fro92a] Christiane Frougny. "Representations of Numbers and Finite Automata". In: *Mathematical systems theory* 25 (1992), pp. 37–60.

[Fro92b] Christiane Frougny. "Systèmes de numération linéaires et theta-représentations". In: *Theoretical computer science* 94 (1992), pp. 223–236.

[Hen+95] J.G. Henriksen et al. "Mona: Monadic Second-order logic in practice". In: *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019.* 1995.

[Fro99] Christiane Frougny. "Fibonacci Representations and Finite Automata". In: *IEEE Transaction on Information Theory* 37.2 (1999), pp. 393–399.

[Sch02] Stephan Schulz. "E - a Brainiac Theorem Prover". In: *AI Commun.* 15.2,3 (Aug. 2002), pp. 111–126. ISSN: 0921-7126. URL: http://dl.acm.org/citation.cfm?id=1218615.1218621.

[DN09] V. Diekert and D. Nowotka. *Developments in Language Theory: 13th International Conference, DLT 2009, Stuttgart, Germany, June 30–July 3, 2009, Proceedings.* Lecture Notes in Computer Science. Springer, 2009. ISBN: 9783642027369. URL: https://books.google.be/books?id=bsoy1HDy36YC.

[Ray10] Sandip Ray. *Scalable Techniques for Formal Verification.* Springer, 2010.

[GSS12] Daniel Goc, Luke Schaeffer, and Jeffrey Shallit. "The Subword Complexity of k-Automatic Sequences is k-Synchronized". In: *CoRR* abs/1206.5352 (2012). URL: http://arxiv.org/abs/1206.5352.

[HS12] Dane Henshall and Jeffrey Shallit. "Automatic Theorem-Proving in Combinatorics on Words". In: *CoRR* abs/1203.3758 (2012). URL: http://arxiv.org/abs/1203.3758.

[RNP13] Michel Rigo, Rampersad Narad, and Salimov Pavel. "On the Number of Abelian Bordered Words". In: *Lecture Notes in Computer Science* 7907 (2013), pp. 420–432.

[Du+14] Chen Fei Du et al. "Decision Algorithms for Fibonacci-Automatic Words, with Applications to Pattern Avoidance". In: *CoRR* abs/1406.0670 (2014). URL: http://arxiv.org/abs/1406.0670.

[Rig14] Michel Rigo. "Formal Languages, Automata and Numeration Systems". In: Wiley, 2014. Chap. 5.

[DPR16] Eric Duchene, Aline Parreau, and Michel Rigo. "Deciding game invariance". working paper or preprint. Mar. 2016. URL: https://hal.archives-ouvertes.fr/hal-01283830.

[Mou16] Hamoon Mousavi. *Automatic Theorem Proving in Walnut.* Feb. 2016. URL: https://cs.uwaterloo.ca/~shallit/Papers/aut3.pdf.

[Boi] Bernard Boigelot. *LASH : Liège Automata-based Symbolic Handler.* Accessed : 15-08-2016.