Automatic demonstration of mathematical conjectures

Floriane Magera

July 2016

Contents

1 Introduction **3**
1.1 About Walnut 3
1.2 Description of the master thesis 3

2 Reminder about numeration systems **5**

3 Addition bit by bit **8**
3.1 Implementation 10

4 Extended Normalization **11**

5 Part I : Frougny's article **13**
5.1 Rules 13
5.1.1 Reductions 13
5.1.2 Unfoldings 14
5.2 Creating the transducer 15
5.2.1 Generate the state space 15
5.2.2 Generate transitions 15
5.2.3 Implementation 17

6 Part II : method using simple automata **19**
6.1 Problem 19
6.2 Reduction automaton 20
6.3 Unfolding automaton 24
6.4 Creation of the base automaton 28
6.5 Strategy one 29
6.5.1 Aim 29
6.5.2 Composition 31
6.5.3 Implementation 32
6.5.4 Composition with delayed versions 33
6.5.5 Consequence 37

6.6 Strategy two 37
6.6.1 New method to nest rule applications 37
6.6.2 Results 37

7 Part III : post normalization 38
7.1 Initial output fonction 38
7.2 Why is this also working? 39
7.3 Adaptation to the new approach 40
7.4 Implementation 41

8 Normalization 42
8.1 Construction of the transducers 44
8.2 Implementation 46

9 Results and limitations 47
9.1 Other Pisot systems 47
9.2 Expand the alphabet 47
9.3 Efficiency problems 47

Chapter 1

Introduction

1.1 About Walnut

Walnut is an application that can be used to check some combinatorial properties of 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'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.reference paper I shallit. Walnut uses the automated way of representing sequences and operations on automata to check the former properties. Each formula given as an input to the tool is translated in automata and operations on them.

Exemple d'utilisation de Walnut

The tool allows to use different kind of numeration bases. One can add custom bases, the only condition is to provide the addition automaton of the new base. The tool comes with Fibonacci's and Tribonacci's addition automaton, which was useful to verify our results.

1.2 Description of the master thesis

The goal of this work is to provide a way to generate the addition automaton for some Pisot numeration system. We limitate ourself to Pisot numeration systems defined by a Fibonacci polynomial i.e polynomials of the following form :

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

from this equation, we can create numeration systems U defined by a strictly increasing sequence of positive integers (u_n) with $u_0 = 1$, u_1,... u_{m-1} arbitrary chosen. Then $u_{m+j} = u_{m+j-1} + ... + u_j$, $j \geq 0$. Given the degree of the polynomial(this value will be denoted by m along the paper) and the first values of the basis $\{u_0, ...u_{m-1}\}$, we want to output the addition automaton in that base. We will denote by F_m the numeration system defined by the Fibonacci polynomial of order m.

We will also assume 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, assume it is properly the case.

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

- **Addition bit by bit** : the name of this step speaks for itself
- **Extended normalization** : The result of the previous step is a number on the initial alphabet doubled. We need to transform the number such that it comes back to the initial alphabet.
- **Normalization** : the latter number is put in normal form.

The core problem of the work is the extended normalization. This part is based on an article of Christiane Frougny reference. We tried to apply her method and at least to obtain the same result.

The Lash library has been used to create automata and then to operate on them. The programing language used to develop the thesis is thus the C language. plus de blabla ici

The illustrations provided will be mostly examples in the F_2 numeration system, as it is the simplest one and the number of states in the different automata stay low even in the final result.

Chapter 2

Reminder about numeration systems

From Rigo's book Let $\boldsymbol{U} = (U_n)_{n\geq 0}$, an increasing sequence of positive integers. U_0 is equal to 1 in order to be able to represent every integer. 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 $rep_U(n)$. Thus the normal representation of a natural number is the smallest representation of that number.

Greedy algorithm to obtain the normal 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, we compute the normal representation $rep_U(l)$ with the following algorithm. First, we look for k such that $U_k \leq l < U_{k+1}$, we note c_k the result of the euclidean division of l by U_k thus $l = c_k U_k + r_k$ where r_k is the remainder. We repeat this procedure with the remainder until we obtain :

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

Note that the c_j are not restricted here. For this work, we need the c_j to be restricted to a certain alphabet. It depends on the numeration base used, as we give the choice of the first values of the base, we trust the user to provide values that allows every integer to be representable on our chosen alphabet $\mathcal{A} = [0, 1]$.

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

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

$\forall n \geq 0, a_0...a_{k-1} \in \mathbb{Z}$.

Completeness

We say that a numeration system defined by F_m and an alphabet $\mathcal{A} = [0, 1]$ is complete if every positive integer is representable. One way to check that a system is complete is to show that

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

for every $k \geq 0$.

- if $k \geq m$, the condition is trivially satisfied by the linear recurrence equation of Fibonacci numeration systems.
- if $k < m$, we show 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 $m > 2$, we have $U_0 + U_1 \geq U_2 - 1$, which gives

  $$2 < U_2 \leq 4$$

  . We will 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,

  $$U_k \leq 2^k$$

We have $U_{i-1} < U_i \leq 2^i$ with $1 < i < m$, so the user has to chose the first values of the base in this interval.

Pisot numeration system

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 nice 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. We will see that we can find a deterministic finite automaton which accepts the language

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

- The addition in the numeration system is also a regular language. Thus we will build an automaton which accepts the language

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

Transducers

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$\delta : Q \times A \to Q$. The output function is $\lambda : Q \times A \to B^*$.

In the work, we will mostly deal with transducers. Note that transducers can be considered as automata with tuples on their transitions, only if the transducer is synchronous. Such that we do not have to define the output function λ.

Chapter 3

Addition bit by bit

The First step of the addition process is the easiest. As you can see on Figure 3.1, we created an automaton A which accepts the language

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

. The automaton is composed of only one state.

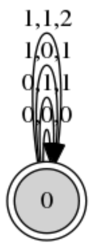

Figure 3.1: Simplest transducer for addition bit by bit for F_2

We can go further and require the inputs x and y to be in their normal form. We can obtain that 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.

The automaton N accepting the normalized words is thus composed of m states, all are 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.

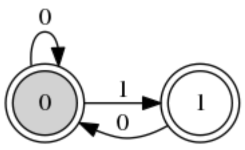

Figure 3.2: Automaton accepting normalized words in F_2

In order to ensure that x and y are normalized, we made the cartesian product of the automaton N with itself, and then composed the resulting automaton with A. The composition operation was not available in Lash, we added it. The implementation of this new operation is described in section 6.5.2 .

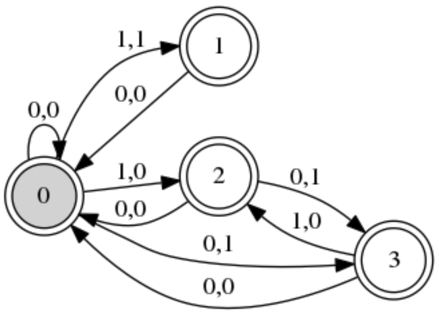

Figure 3.3: Automaton reading two normalized words in F_2

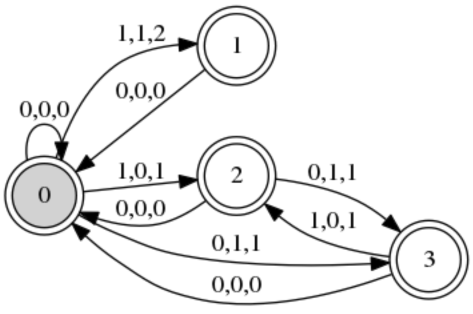

Figure 3.4: Final addition transducer for normalized input in F_2

3.1 Implementation

The code created to generated this part is located in `addition.c`. Only two functions are needed :

- `normalized` that generates an automaton representing the normal representation of the naturals.
- `addition` that creates the simple addition automaton seen on Figure 3.1

The composition of these automata is performed in the main function.

Chapter 4

Extended Normalization

In the previous chapter, we have added two numbers in their normal representation, but 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 you can see in the following simple example in F_2: $01 + 10 = 11$.

For these reasons, we need first to reduce the alphabet of the output, and then we will put it in its normal form. Here we focus on the reduction of the alphabet, which is called the extended normalization. Some mechanisms presented will be reused in the normalization part.

The problem was to answer the question : knowing the Fibonacci polynomial, how can we transform a word on $[0, 2]^*$ to another word on $[0, 1]^*$? For example, in F_2, it is easy to see that the following transformation is correct, but what is the correct procedure to transform it 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.

First we will present the method proposed by Christiane Frougny to solve this problem, then our approach. We want to obtain an equivalent result such that the properties demonstrated by C. Frougny still hold for our implementation. The final implementation relies heavily on her work, with a shift of the complexity from the algorithm to generate the transducer to operations on automata.

Chapter 5

Part I : Frougny's article

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. We explain here the method specified in our 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 capital and will be used further, whilst the rest of the procedure is not used in the final implementation.

5.1 Rules

In order to transform a word, we can use the polynomial. We can derive two types of rules from it : reductions and unfoldings.

5.1.1 Reductions

These are performed by using the polynomial on factors of size m+1. The left-most letter is thus increased by one and all the others are decreased by one. The reduction can be applied several times. The set of possible reductions is described by :

$\{\ 0(\mathrm{i}_1 + 1)...(i_{m-1} + 1)l \rightarrow 1(i_1)...(i_{m-1})(l-1)|$
$1 \leq l \leq 2, 0 \leq i_1, ..i_{m-1} \leq 2\}$

(5.1)

Reductions are also used for the last step of normalization.

5.1.2 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 :

$$\begin{aligned}&\{0(i_1+1)...(i_{j-1}+1)(i_j+1)\nu_{j+1}...\nu_{j+m} \rightarrow\\ &1(i_1)...(i_{j-1})(i_j-1)\nu_{j+1}...\nu_m(\nu_{m+1}+1)...(\nu_{m+j}+1)|\\ &1 \leq j \leq m-1, 0 \leq i_1,..i_{m-1} \leq 2,\\ &\nu_{j+1}...\nu_{j+m} \in [0,2], \nu_{j+1}...\nu_m <_{lex} 1^{m-j-1}1\}\end{aligned}$$

(5.2)

Where $<_{lex}$ denotes the lexicographic ordering.
Unfoldings are needed specifically in the case of an extended alphabet, where we can face words of the type 020^n that can not be reduced with a simple reduction. Note also that unfolding can extend the alphabet further to $[0,..3]$.

Now we have two sets of rules, we will use them 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.

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

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

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

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

$$03221 \rightarrow 12121$$

And then no other rule is applicable. If we transform it in the other way we get

$$03221 \rightarrow 21021$$

We see that the last factor can be reduced too.

$$03211 \rightarrow 21110$$

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

5.2 Creating the transducer

5.2.1 Generate the state space

The state space is directly extracted from the rules. The states represent the current factor being considered at a certain point when we are 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.

5.2.2 Generate transitions

From each state q, we consider each possible input x. Let w = qx. We denote by $w\backslash y$ the word w from which the factor y has been removed. There are three cases :

- There is no rule matching w. We have 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 suffixes matching, we take the longest suffix. 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 $w \rightarrow w'$ and w' can not be reduced anymore (otherwise we apply again reduction rules), we consider w'. There are again two cases. We name l the last letter of w':
    - l belongs to the reduced alphabet, so $l \in [0, 1]$ for us. Then $w'\backslash l$ is written on the output state and the next state is l-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 shape of unfoldings, there will always be at most (m-1) '3' created, so the last digit can not be a '3'. Then we must consider w' which is of the form : $(h+1)(i_1)...(i_{m-1})1$, let k be the greatest integer such that $i_k = 0$ and $i_{k+1}, ...i_{m-1} \geq 1$. We output everything before i_k : $(h+1)i_1..i_{k-1}$. And the next state is $0i_{k+1}..i_{m-1}1$.
    If there is no such k, it means that we can apply one more reduction. Then we apply the same procedure.

- An *unfolding* rule is applicable. If $w \rightarrow w'$, we consider w'. We write on the output tape n, the longest strict prefix of w' such that n belongs to the reduced alphabet and such that the suffix $w' \backslash n$ begins with '0'. This decomposition exists because of the requirement $\nu_{j+1}...\nu_m <_{lex} 1^{m-j-1}1$ in equation 5.2 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 are powers of two : $U_0 = 2^0, ...U_{m-1} = 2^{m-1}$. We still explain her method, even if it was actually never used. The method we used is explained in section 3.3.

We define here the output function $\omega(q)$. There are several cases :

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

- If it is not the case, we have to normalize q, but by definition of the states there are no rule that can be applied. Let $q = hq_1...q_l$ with $1 \leq l \leq 2m-2$.
  The approach proposed is to continue the base with $U_{-1} = 1, U_{-2} = 0, ...U_{-m} = 0$. We prolong q with m zeros : $q = hq_1...q_l, 0^m$ If $l \leq m-1$, we can apply an unfolding rule .

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

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

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

  if $m \leq l \leq j+m-1$ The same method as previously is applied : we prolong q and we also apply one unfolding, 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 '-1'. If $q_l < 2$, then we add one to q_l. Otherwise we can apply a second unfolding with $q_{l+2}...ql + m = 0$. As a result, q_l will be reduced, and if q_{l+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 5.1.

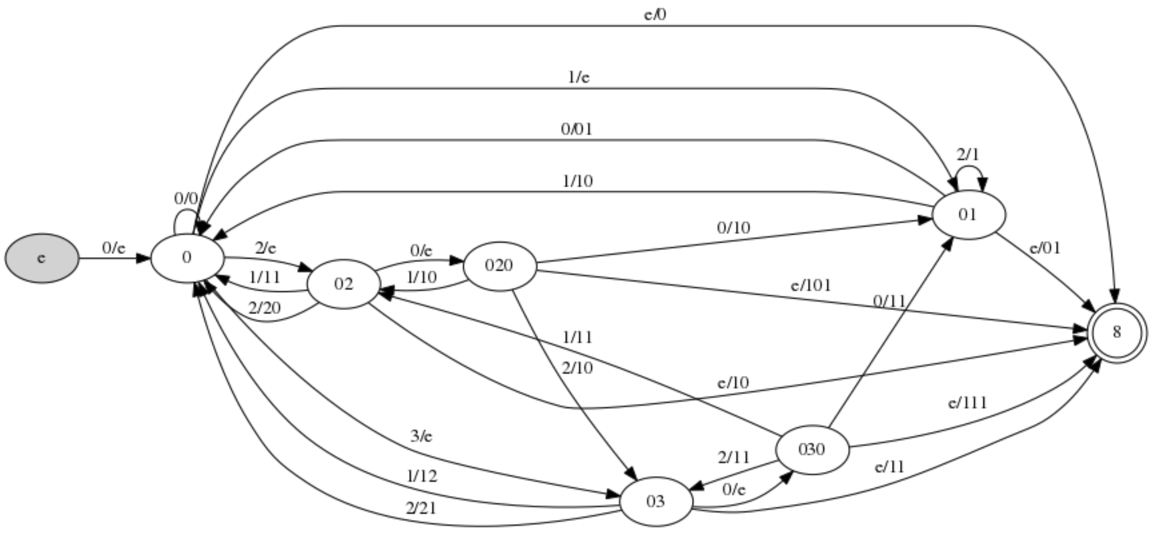

Figure 5.1: Transducer obtained with Frougny's method

We cannot use this transducer in our work, but we want to keep the same effect and strategy : the transducer applies rules, and a nice property about it 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, we will keep it in memory.

The problem is that some of the labels of the transducer contain the empty word, and thus the transducer can not be translated in a deterministic finite automaton. Justifier plus ?.

5.2.3 Implementation

The translation of the transducer was not considered as a problem at first, so we started implementing the generation of all the data needed to create the transducer. We generated the rules, the states and the transitions. Only the transitions are neglected in the final version of the work.

Pattern

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

We must provide a lot of different operations :

- For

Rules

States

Transitions

HashTable

Generation of the data

Chapter 6

Part II : method using simple automata

We cannot use the transducer proposed by C. Frougny, but we still wanted to have the same effect. In that way, we can obtain the same efficiency : the extended normalization is 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. In the previous implementation, alphabet expansion was considered. This is not anymore the case.

One solution would have been to transform the transducer to make it synchronous, but we preferred to try to make the algorithmic simpler and to rely on simple operations on automaton.

6.1 Problem

We want to mimic the behaviour of the transducer previously introduced. First, we can formalize its effect. The transducer applies rules, in a sequential way on words, from left to right. We can think of the application of one rule as a basic action. This action is repeated maybe an infinite number of times, but it is not a simple loop : as rule application may be nested, the previous result of a transformation can be fed into the next application. But we need also to have the possibility of not applying any rule for a while. Whatever our solution to mimic the asynchronous transducer, we need a basic automaton that applies one of the rules.

We want to generate a transducer that transform a word matching a rule. There are two types of rules as we have seen. We will first generate an automaton for each type.

In the pseudo-code of the two following subsections, there are references to functions of the lash library. The names of those functions speak for themselves. We simplified some names and some usages in order to keep the pseudo-code more compact.

6.2 Reduction automaton

The reduction automaton is the simplest. We want 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 systems grows quite fast, so we tried to reason on the definition of the reduction rules. In order to build it in an efficient way, we can generalize the form of reductions. If we observe the set of reductions for F_2 modified for maximal reduction :

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

We can divide this set two subsets :

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

and

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

These subsets correspond to the number of times we can apply a reduction on the factor.

Then we observe that if we generate these rules, for each member of a set, there is always a digit that justifies the number of reductions. For example : if we take the rule $021 \rightarrow 110$, the one in the left member is the restricting digit. In our implementation, we have to pay attention to that in order to generate complete rules. We could generate all rules for example we could have :

$$022 \rightarrow 111, 022 \rightarrow 200$$

But it would lead to a less compact automaton, and moreover, it would bring non-determinism.

Once these observations made, we can explain our algorithm, which is recursive.

```
Function get_red(int alph_max, int m)
    automaton a = new_automaton();
    int init = auto_add_new_i_state(a);
    for i = 1; i < 2*alph_max; i++ do
        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);
    end
    return a;
```

Algorithm 1: Algorithm to generate reduction automaton

```
Function red_rec(automaton a, int curr_state, int to_red, bool
  is_restricted,int alph_max, int tr_left)
    if tr_left == 0 then
        return ;
    else
        int new_state = auto_add_state(a);
        if !is_restricted then
             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 then
                int new_state2 = auto_add_state(a);
                for i = to_red+1; i <= alph_ + 1; i++ do
                    auto_add_transition(a, curr_state, new_state2,
                     "i/i-to_red");
                end
                red_rec(a, new_state2, to_red, false, alph_max,
                 tr_left-1);
            end
        else
            for  i = to_red; i <= alph_max + 1; i++ do
                auto_add_transition(a, curr_state,
                 new_state,"i/i-to_red");
            end
            red_rec(a, new_state, to_red, is_restricted, alph_max,
             tr_left-1);
        end
        if tr_left == 1 then
            auto_mark_accepting(a, new_state);
        end
    end
```

Algorithm 2: Recursive algorithm to construct the reduction automaton

In the `gen_red` function, we create 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.

The `red_rec` function takes as arguments :

- `a` : the automaton
- `curr_state` :the current state from which we are building the automaton
- `to_red` : the number of reductions that we apply, in our case 1 or 2, but it could be more with an alphabet expansion
- `is_restricted` : a boolean to remind whether we have already a restricting agent
- `alph_max` : the upper bound of the doubled alphabet
- `tr_left` : the number of transition that we still need to build

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

- Either we have already created a transition with a restricting digit, then we add transitions reading number from `to_red` to `alph_max+1` and outputting the read value minus `to_red`
- Either we have no restricting digit, so we can create one or decide to create it later if this is not the last transition that must be created.
  – If we create the restricting transition, we just add one transition labeled with `to_red` as input and zero as output, then we make the recursive call with `true` as the new value of `is_restricted`.
  – Otherwise we can leave it for later and we behave as previously in the first case considered, creating many transitions to another new state.

We did not use the theoretical definition of the rules that we also implemented. Note that if we wanted to expand the alphabet, the algorithms would not have to change. We mentioned that we tried to find an algorithm to generate this automaton in an efficient way. We could also 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. We did not try this solution, as the number of rules grows fast, we keep the option to save memory and computing time. The automaton resulting is shown on Figure 6.1

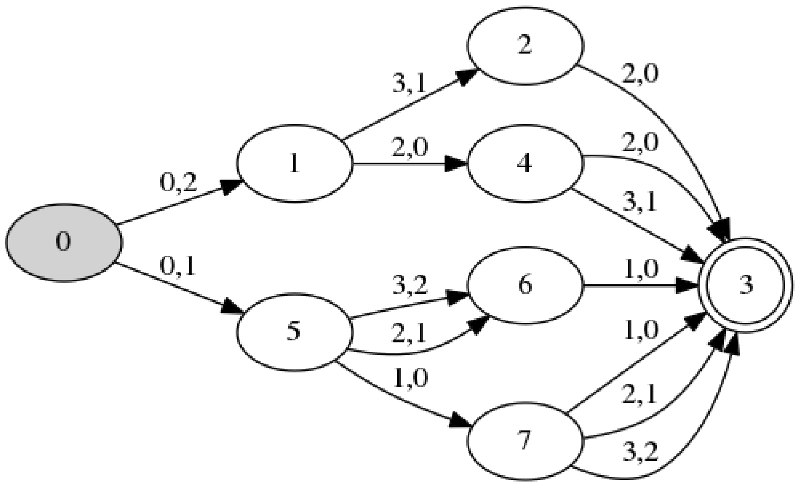

Figure 6.1: Transducer applying one reduction rule

6.3 Unfolding automaton

Now we want to generate the unfolding automaton. For once we will use the F_3 numeration system. 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 unfolding is $[m+2, 2m]$. So for F_2 the size of unfolding rules is fixed to 4, whilst for F_3 it is $[5, 6]$.

Just like we did for reductions, we observe the properties of the unfolding rules for F_3. We will just write a few, as there are many.

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

While for reductions we could divide the set of rules according to the number of reductions possible, here we can divide them by size :

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

and

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

Then we notice that there is a middle factor that remains unchanged. The length of this factor is defined by the rule's length. In unfoldings, we expand the jth term ≥ 2, then we reduce the first m terms. 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.

There is one last thing to remark : the unchanged factor should be $<_{lex} 1^{m-j}$ so we should remember whether we have already put a zero in the middle factor when generating it. This will be dealt with in a similar way than the restricting digit in the reduction case.

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

```
Function 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 j = 1; j < m ; j++ do
        un_rec(a, second, j,1, 2*alph_max, m,false);
    end
    return a;
```

Algorithm 3: Algorithm to generate unfolding automaton

The first function is simple : we create the two first states, because the first transition is always the same, then for each length possible, we make a recursive call to construct the rest of the automaton. Again, we have 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.

```
Function un_rec(automaton a, int state, int j, int curr_tr,int
  alph_max, int m, bool restr)
    if $curr\_tr == m+j+1$ then
        return;
    end
    int new_state = auto_add_state(a);
    if $curr\_tr == order + j$ then
        auto_mark_accepting(a, new_state);
    end
    if $curr\_tr == j$ then
        for $i = 2; i \leq alph\_max+1; i++$ do
            auto_add_transition(a, state, new_state, "i/i-2");
        end
        un_rec(a, new_state, j, curr_tr+1, alph_max, m, false);
    end
    else if $curr\_tr < j$ then
        for $i = 1; i \leq alph\_max+1; i++$ do
            auto_add_transition(a, state, new_state, "i/i-1");
        end
        un_rec(a, new_state, j, curr_tr+1, alph_max, m, false);
    end
    else if $curr\_tr > j \;\; curr\_tr \leq m$ then
        if restr then
            for $i = 0; i \leq alph\_max; i++$ do
                auto_add_transition(a, state, new_state, "i/i");
            end
            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 $curr\_tr != m$ then
                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);
            end
        end
    end
    else
        for $i = 0 ; i \leq alph\_max; i++$ do
            auto_add_transition(a, state, new_state, "i/i+1");
        end
        un_rec(a, new_state, j, curr_tr+1, alph_max, m, restr);
    end
```

Algorithm 4: Algorithm to generate unfolding automaton

The second function is quite longer. We explain quickly the arguments :

- a : the automaton
- state : the index of the state we are working on
- j : the value j, the index of the digit we will reduce 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 limitated, i.e if it already satisfies the condition $<_{lex} 1^{m-j}$

We act depending on the number of transitions already created. We know that we have to generate m+j+1 in final. If they are all generated, we end the recursive call, otherwise, if we are creating the last transition and thus the last state, we mark the fresh state as accepting. Then from our position in this path, we distinguish 4 parts in every rule application :

1. $curr\_tr < j$, we generate all transitions that read as input a number between $[1, 3]$ and outputs it decremented. All transitions go from the current state to the new one.
2. $curr\_tr = j$, we are at the digit that will be decreased by 2. We do just as before, except that inputs $\in [2, 3]$.
3. $j < curr\_tr \leq m$, we are in the middle factor that will stay unchanged. We have a condition to fulfill. Either it is already done, and then `restr` is true, then we read any input in $[0, 2]$ and output the same value.

   Otherwise, we can fulfill the condition now by adding a transition labelled by "0/0".If we are not at the last term of the middle factor, we can also choose to satisfy the condition later. Then we create a second new state and we make a transition with label "1/1" to it.
4. $m < curr\_tr \leq j + m$, we read any input in the doubled alphabet and we output that value incremented.

This implementation relied more on the definition of the unfolding rules. We would like to emphasize that the middle unchanged factor exists because there is at first a term inside it that prevents the application of a reduction rule. This is essential difference between reductions and unfoldings. Unfoldings exists only if there is a factor of size m containing first a digit ≥ 2 followed by a zero.

The resulting automaton is shown on Figure 6.2. 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.

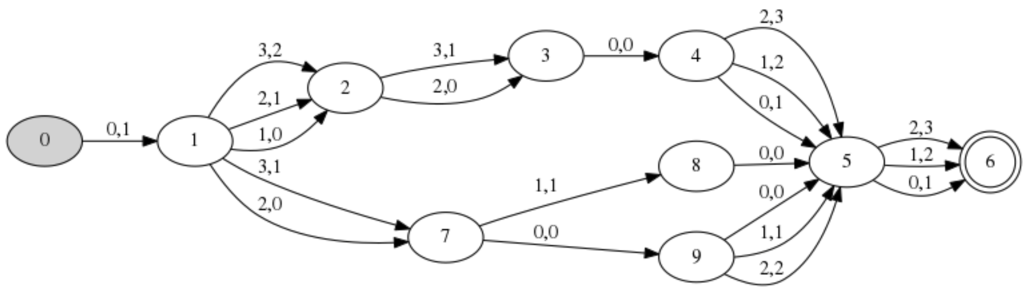

Figure 6.2: Transducer applying one unfolding rule

6.4 Creation of the base automaton

Now that we have two transducers applying reductions and unfoldings, we just need 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 $l \in [m+1, 2m]$. The result of the union is shown in Figure 6.3. Once again, we show the result for the Tribonacci numeration system.

We still have to figure out how to use this simple automaton in order to perform extended normalization on factors that do not exactly match the left member of a rule ? We certainly need to add an identity function to the transducer, but then we would also accept infinite words while we had a nice compact interval.. We will also certainly add a loop in order to allow the sequential application of rules. But how can we have the nesting we aim ? We focus on these problems in the next section.

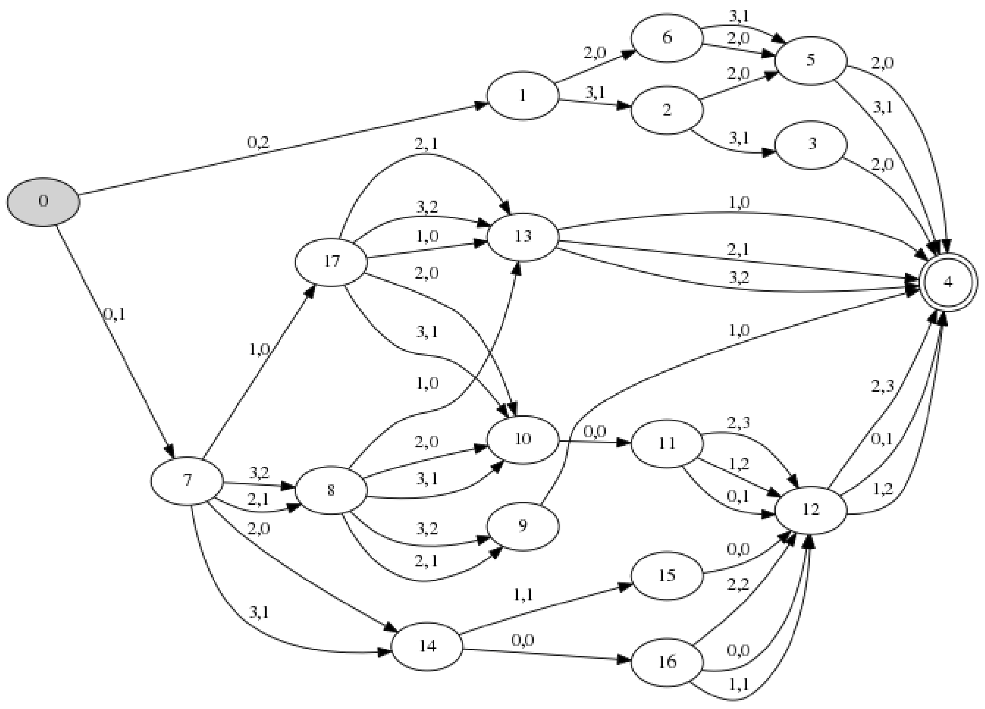

Figure 6.3: Transducer applying one rule in F_3

6.5 Strategy one

6.5.1 Aim

We create a schema to show the effect we would like to obtain. It is represented on Figure 6.4 The arrow in bold represents a word or a sub-word. The blue rectangles represent a transformation of a factor by the application 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.

We already have an automaton that corresponds to the effect of a blue box. Now we have to find out how to modify it to obtain the good result. We know that we will have to use parts of modified factors as input to the next transformation. We have first to create that operation, which basically is

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

The application of an automaton $\mathcal{A}$ to the language accepted by another automaton $\mathcal{B}$. We create thus the operation of composition, which was not

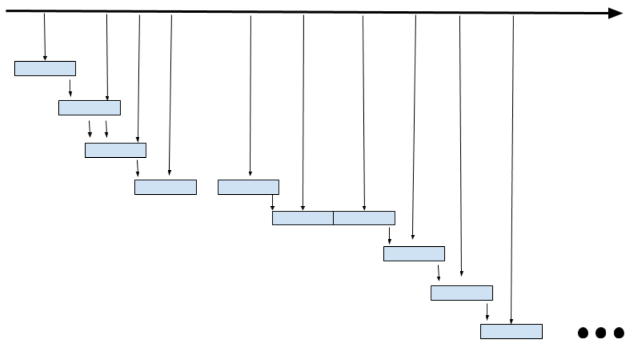

Figure 6.4: Effect on a word

present in the lash library.

6.5.2 Composition

We explain first the simple way to apply composition, then we will explain our implementation.

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

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

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

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.

We will expand the alphabet of both transducers. We modify the alphabets such that :

$$A_1' = \Sigma_1 \times \Sigma_2 \times \Sigma_3$$
$$A_2' = \Sigma_1 \times \Sigma_2 \times \Sigma_3$$

This can be obtained by modifying each transition of the transducers : for each transition, to create n new transitions with $|\Sigma| = n$, with the same start and end, just the labels are expanded with the nth 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. We will obtain a new automaton $\mathcal{I}$ whose alphabet is :

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

Now we don't need the second composant, as it is a temporary one. We finish 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. We did not implement it as simply, because we need a lot of different composition operations. 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 we need a more general composition that works also with automatons with 3 tapes for examples.

6.5.3 Implementation

The implementation of the composition is performed in the file `composition.c`. First we introduce the algorithm to expand the alphabet of an automaton. We explain the arguments

- the automaton to modify.
- the position at which we should place the new composant, either at the beginning or at the end.
- the number of composants per transition of automaton a.
- the alphabet of the new composant.

```
Function add_ composant(automaton a, int position, int ln, int alph)
    automaton uni = auto_new_uni(alph);
    if position == 0 then
        a = auto_product(a, uni, ln, 1);
    else
        a = auto_product(uni, a, 1, ln);
    end
```

Algorithm 5: Algorithm to expand one automaton's alphabet

Then the algorithm consists in building an automaton accepting anything on the aimed alphabet `alph`. This step is summarized in the pseudo-code. Then if we had the new composant at the beginning of each transition, we make the product $\mathcal{U} \times \mathcal{A}$ 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 we have is the composition of the addition algorithm and the extended normalization. The addition automaton has two input tapes and one output, while the extended normalization automaton has tape for input and one for output. S we want to keep just the input tapes of the addition and the output tape of the extended normalization. We have thus some extra parameters to allow such changes.

As arguments we have :

- Each automaton and the length of its transitions labels.

- The alphabet of the new components, for us 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 we know how much components we should add to each automaton.

```
Function compose(automaton a, int ln_a, automaton b,int ln_b, int alph, int ln)
    while ln_a < ln do
        add_composant(a,ln_a, ln_a, alph);
    end
    while ln_b < ln do
        add_composant(b,0, ln_b alph);
    end
    automaton c = auto_intersection(a, b);
    int to_project = (ln_a+ ln_b)-ln;
    if ln == ln_a // ln == ln_b then
        to_project = 0;
    end
    for int i = 0, l = ln ; i < to_project, l > 0; i ++; l- do
        result = auto_projection(result, l, ln_a -1 -i);
    end
    return c;
```

Algorithm 6: Composition algorithm

We can explain the core algorithm : first we perform the addition of new components to each automaton, then we compute the intersection. The number of projections to perform is then computed and we project starting from the length of the first automaton.

6.5.4 Composition with delayed versions

The principle of this strategy is to use composition of the base automaton with it self delayed. Several approaches were tried to obtain a satisfying result. In any case, we want to allow a factor to stay unchanged and we need to apply the rules several times as well. The first step is thus to add an identity part to the automaton, and make it a loop. The identity part should sometimes be limited in size. One can see the resulting automaton for F_2 on Figure 6.5.

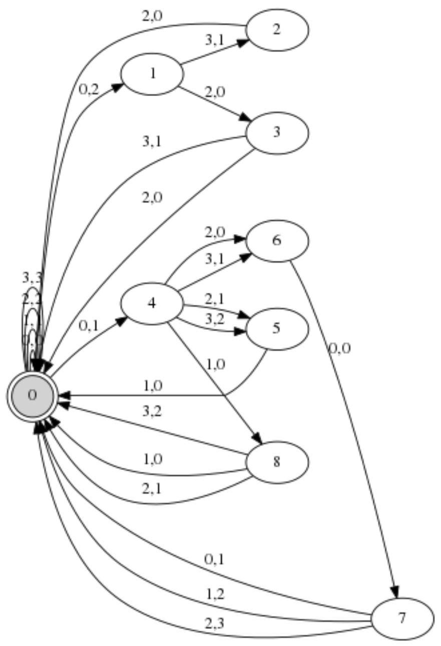

Figure 6.5: Base automaton with identity and loop for F_2

Approach 1

The first idea is thus to compose the previous automaton with it self delayed. If we use the same representation as in Figure 6.4, we would have : On the

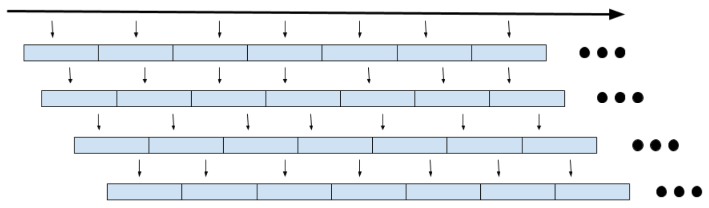

Figure 6.6: Effect of the first approach

figure 6.6, we put 3 delayed versions of the base automaton. In order to cover all the nestings possible, we consider the length of the rules : $|rules| \in [m+1, 2m]$. The nesting of a rule is only possible if in the previous transformation, a '0' was created. But since the length of every application vary, there is no point in defining which delayed are relevant. We choose to compose the automaton with its 2m-1 delayed versions.

Problems We don't get exactly the same effect as Frougny. Morevover, the order in which the transformations are made is different, it is not sequential anymore. We can not be sure whether or not 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 we have a word without any zero like $w = 0022222$ the loop in the base automaton is useless. Indeed we would need a real nesting to be able to reduce the word, as we have a composition with 2m-1 automata, if there are more than (2m-1)m factors '22'/'21'/'12', the word won't be on the good alphabet. We can see an example in F_2 :

$$0\textcolor{red}{012}1212212212 \to \textcolor{red}{0101}1212212212$$

$$01011212212212 \to 01011212212212$$

$$01\textcolor{red}{011}212212212 \to 01\textcolor{red}{100}212212212$$

$$0110\textcolor{red}{021}2212212 \to 0110\textcolor{red}{110}2212212$$

Result after the first pass through the transducer. 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.

$$\textcolor{red}{0110110221}2212 \to \textcolor{red}{1001002001}2212$$

$$10010\textcolor{red}{02001}2212 \to 10010\textcolor{red}{10011}2212$$

$$1001010\textcolor{red}{011}2212 \to 1001010\textcolor{red}{100}2212$$

$$100101010\textcolor{red}{022}12 \to 100101010\textcolor{red}{200}12$$

As a result, the word is indeed not in the desired form. So the method is not working.

This method was still developed, and even if it had worked, it would not have been usable. Indeed, the composition is a rather heavy operation. In the case of F_2, we need to compose 3 times the automaton with itself. 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, we might obtain an automaton with thousands of states, whilst the final automaton we aim in that case has only 15...

For these reasons, this approach is not relevant. The fact that it could not work was discovered late as we never obtained a usable result, the program would not terminate.

Approach 2

The problem with the last procedure is that the nesting is not performed as we would want to be. The idea is then to compose the base automaton, but the version without the loop. So we compose it with its 2m-1 delayed versions, then we add a loop. The effect is represented in blue on Figure 6.7.

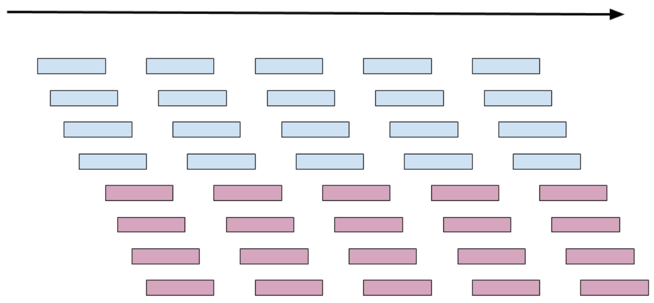
Figure 6.7: Effect of the second approach

One block composed of 2m applications acts on factors of length 4m-1. The problem is that there is no nesting between two blocks.So we compose this first automaton to a second one with itself delayed 2m times. Now the nesting is guaranteed on 8m-2.

Results If we try this version without loop, and test it on words of limited length, we obtain that the automaton resulting accepts a language that includes the language of the addition on limited words. But this result is lost after the loop, because we will always have a cut in the sequence of overlappings. This solution does not work either.

Even if it worked, we still have problems with heavy compositions. 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.

6.5.5 Consequence

We have seen that we could not obtain the same effect with those two approaches. Moreover we observe that composition should not be used too much for efficiency problems. We come up in the next section with a way to avoid these heavy composition, such that the final automaton performing extended normalization has as much states as the base automaton seen on Figure 6.5.

6.6 Strategy two

We will use a method that we already used for normalization, except that we formalize it. The method we explain here does not uses composition, it modifies the base automaton to allow the overlapping of the rules.

6.6.1 New method to nest rule applications

We start from the observation that if there is an overlapping, we will have an factor output that we will use 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 match. The idea is then to bypass the initial state. We create a new transition 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.

From the form of the rules,

6.6.2 Results

Chapter 7

Part III : post normalization

We need post normalization because we might increase the input alphabet during the extended normalization. The extended normalization takes as input words on the doubled alphabet $[0, 2]$. But we have seen that unfoldings can produce words on $[0, 3]$. For example the rule $0202 \rightarrow 1003$. We want to obtain a reduced alphabet, so if a word finishes by a factor matching the left member of the previous rule, we need to apply a further transformation.

Here we differ from Frougny's approach by not forcing $U_0 ... U_{m-1}$ to be powers of 2. This method was developped when trying to adapt the transducer described by Frougny. So we start from the states defined in her work, keep in mind that these states are the remainder of the factor we are treating. If we reach the end of the word in a certain state, the label of the state is the current remainder, if the label is 2 digit long, we have to output 2 digits too, preferably on the final output alphabet $\mathcal{A} = [0, 1]$.

We start by explaining the initial approach developped for Frougny's transducer and then its final use.

7.1 Initial output fonction

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

Otherwise, we try to normalize q knowing the m-1 first numbers of the numeration system $u_{m-1} .. u_0$.

Our 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, we will prefer to transform it to "012" than "020", in that way, we allow directly the use of a reduction rule at the next step. So we apply a little twist to the greedy algorithm showed in the reminder. We know the first values of the base, and we only use these to normalize the subword $q_{m-1}..q_0$.

Expansion We compute the value of the integer $i = q_{m-1}..q_0$, we note $s = \sum_{k=0}^{m-1} U_k$. If $i > s$, we will use the euclidean division : $i = ns + r$. 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 we take back the full word q, with the ending subword transformed. If a rule can now be applied, we apply it and if necessary we go back to the expansion step and so on until either no rule is applicable, or until the word is on the final alphabet.

7.2 Why is this also working?

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

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

with at least one of the $i_k \geq 2$ As $|s| \leq m$ there is no rule applicable for this case. We rely only on the assumption of a complete system on the chosen alphabet. If the first values are correctly chosen, 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 we consider the set of states derived from unfoldings :

$$s' = 0(i_1+1)...(i_{j-1}+1)(i_j+1)\nu_{j+1}...\nu_{j+m-1} | 1 \leq j \leq m-1, 0 \leq i_1, ..i_{m-1} \leq 2, \nu_{j+1}...\nu_{j+m-1} \in [0,2],$$

The shape of the states guarantees that there is at least one zero in the ending of the state, and also $i_j \geq 1$ belongs to the ending of the word. If we consider 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 we can use the twisted version of the greedy algorithm to ensure that there is no more zero in that part of the word, thus allowing a reduction to be used later.

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, we know that the number will be normalizable with the greedy algorithm.

It is also sufficient to say that a normalization rule can be applied later. We say that when there is a factor $w \geq 1^m$. So when we will go through the next pass of in the transducer. If we have $0w$, a rule is applied directly, or if we face a factor $0vw$ with $v \geq 1^l, 1 \leq l$

- If $l \geq m$, then as we reduce as much as we can, at least one zero will be created in the factor v : we will then consider from the right-most zero created the remaining factor v': $0v'w$ if $|v'| \geq m$ we repeat the same operation until we create a zero the new remainder $|v'| < m$ and then we will indeed start reducing w.
- If $l < m$ then w is directly reduced. There may be several sequential application of rules.

7.3 Adaptation to the new approach

As explained, the transducer can not be translated in a DFA. We still use the data created for this step in our final implementation. 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. Our approach was then to create a synchronous transducer, which reads the state and output its output value.

The implementation of that automaton is really simple : for each state not in [0,1], we read synchronously its label "l" and its output "o" value digit by digit. We form then new transition labels "l_i, o_i" and we direct this new transition to a freshly created state, from which we will generate the next transition if we are not at the end of the label, in which case we only make the state accepting. The automaton should be able to read words longer than 2m, so the initial state is accepting and there is a self-loop reading and outputting "i, i" with $i \in [0, 2]$

The resulting automaton is then normalized with the lash function `auto_minimize`. The result is a automaton performing extended normalization in the special case of words or factors smaller than 2m. It must be composed with our extended normalization automaton to form the complete extended normalization automaton.

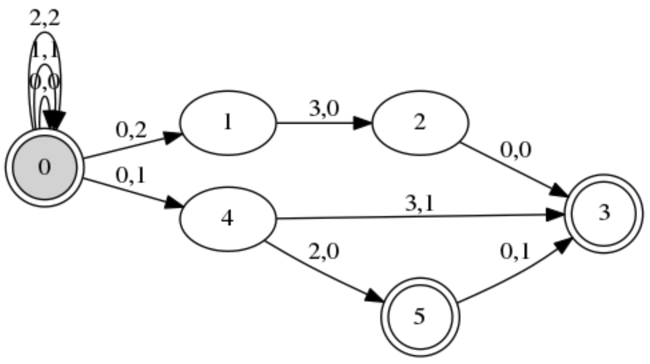

Figure 7.1: F_2's transducer for post normalization

7.4 Implementation

Chapter 8

Normalization

How we generate it, examples. According to **previous papers**, normalization can be obtained through one pass in two transducers. These transducers just apply the transformation

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

repeted 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.

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

We can show that for each of these transducer, there is a worst case that would need a number y of application of the transducer alone, y depending only on the input word.

Right to left transducer worst case

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 :

$$\begin{aligned}0111111 &\rightarrow 1001111\\ 1001111 &\rightarrow 1010011\\ 1010011 &\rightarrow 1010100\end{aligned} \tag{8.1}$$

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

Left to right transducer worst case

Similarly, the worst case for the left to right transducer is a word in the form : $0(1^{m-1}0)^n1^m$. At each application the transducer can only transform again the last part of the read word. An exemple of some executions on the word $w = 0101011$:

$$\begin{aligned}0101011 &\rightarrow 0101100\\ 0101100 &\rightarrow 0110000\\ 0110000 &\rightarrow 1000000\end{aligned} \tag{8.2}$$

Once again, we observe that we have to apply the transducer to the word n+1 times. There is again no limit on the number if passes needed to normalize the word.

We can observe that the worst case of each transducer is perfectly handled by the other one. Which gives an insight on why we should apply them 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 implementation, we faced a limitation. We want to 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.

8.1 Construction of the transducers

We stated before what we want to achieve : two transducers applying sequentially the reduction rule $\{011 \to 100\}$. We start thus from a transducer applying the rule. You can see two transducers on Figures 8.1 and 8.2 for F_2.

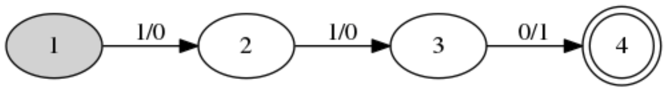

Figure 8.1: Simplest transducer normalizing once from right to left

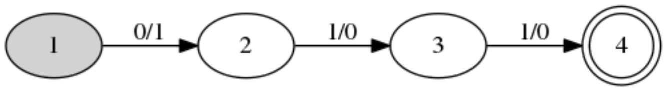

Figure 8.2: Simplest transducer normalizing once from left to right

Then we want to be able to repeat this transformation. A first step is to merge states 4 and 1 on Figures 8.1 and 8.2, 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 we would also like to allow nested transformations :

$$01011 \to 1000$$

$$01111 \to 10100$$

the first transformation is nested for the transducer reading from the right and the second one is for the left transducer.

In the case of the right transducer, we can consume the '1' created in a further transformation. Similarly, for the left transducer, the 'O' created can become a '1' in the next transformation. We are merging the incoming and the outgoing transitions from the initial and final state. So we create a transition going from the penultimate state to the second one, for the right transducer, we have to merge the labels "0/1" with "1/0" which gives "0/0" and for the left one : "1/0" and "0/1" which gives "1/1". The merging is possible only because the output of the incoming transition and the input of the outgoing one are equal. You can see the result in the F_2 case on Figures 8.3 and 8.4. The transitions resulting of the merging are in red.

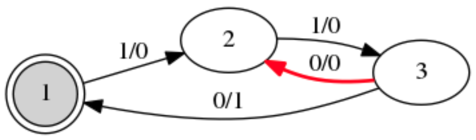

Figure 8.3: Transducer normalizing from right to left

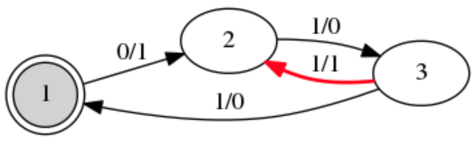

Figure 8.4: Transducer normalizing from left to right

The next step is to reverse the right transducer, as the addition transducer will be left subsequential. Then we also want 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 8.5 and 8.6.

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

Note that on the left transducer, we allowed the generation of "11" factors. We compose the left transducer with the right one, so these factors will be eliminated further.

Finally the resulting transducer is shown on Figure 8.7.

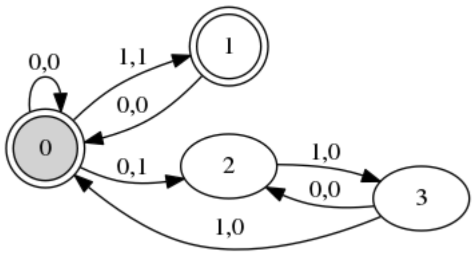

Figure 8.5: Transducer normalizing from right to left

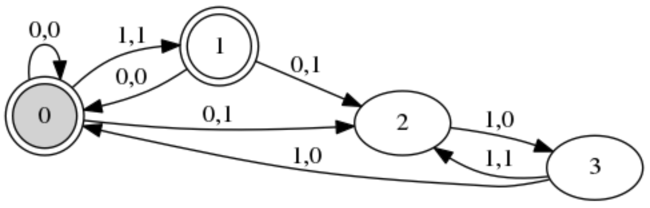

Figure 8.6: Transducer normalizing from left to right

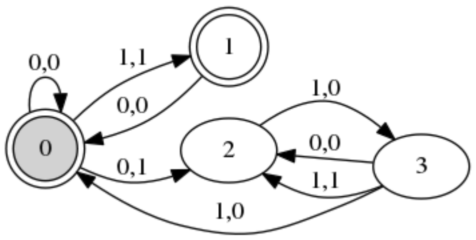

Figure 8.7: Transducer performing normalization in F_2

8.2 Implementation

Chapter 9

Results and limitations

Show visible output (so F_2), talk about limitations in time (exponential growth of computing time, due to composition). Could we extend this work to other Pisot bases = would the rules work for that ?

9.1 Other Pisot systems

9.2 Expand the alphabet

9.3 Efficiency problems

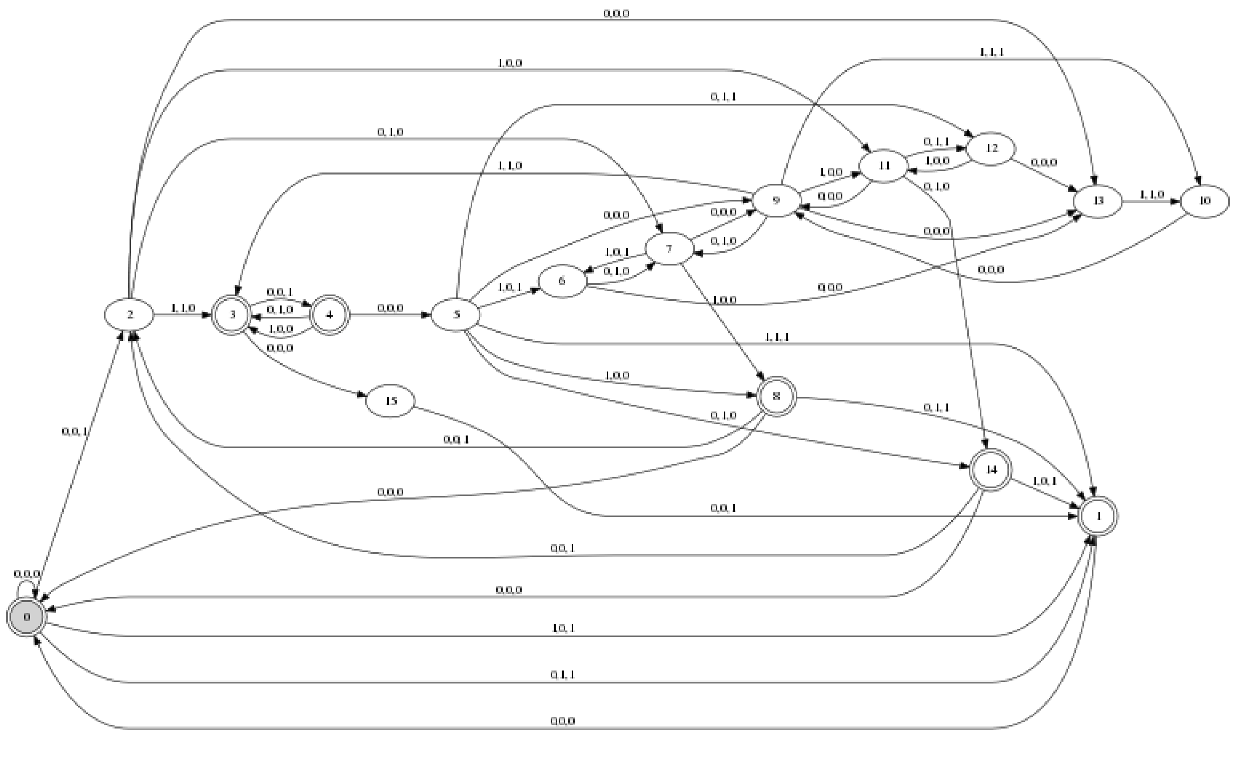

Figure 9.1: Addition automaton for F_2