Internship report

PABLO ROTONDO

In this report I describe the contributions I have made to the Maple package NewtonGF during my internship in the team Aric in LIP, ENS Lyon / INRIA, under the supervision of Bruno Salvy. The package NewtonGF, developed with Bruno Salvy and Carine Pivoteau, provides a comprehensive set of functionalities enabling fast computation in the context of implicit combinatorial systems, generating functions, and the generation of random discrete structures.

Contents

1 Introduction

1.1 Plan of the report

The document is divided into the introduction, and then one section for each different aspect of NewtonGF covered during my internship

- To begin with I was asked to implement the algorithm described in [PSS12] to decide whether a system of combinatorial species is well-founded. The prexisting implementation, in the package **CombStruct** of the standard Maple library, ran into problems when working with restricted cardinalities, resulting in the rejection of several valid systems. The reader will find this, along with some interesting details that turned up during the implementation of the algorithm, in section 2.
- The next topic was to optimise the Maple implementation of the algorithm that computes the result of applying the exponential $\exp$ to a power series, since B. Salvy and C. Pivoteau had noticed that labelled systems involving SET were not as fast as they could be. This topic, along with a detailed account of the issues found as a result of my looking at the efficiency of similar systems, is described in section 3.
- The last point concerns the radius of convergence of a given system of species. The existing implementation of it consisted in performing a binary search (a.k.a. 'dichotomy' in French). This was deemed slow (and also unproven), and hence B. Salvy and C. Pivoteau suggested investigating and implementing a different approach involving Newton iteration (section 4).

1.2 Notation and definitions

Throughout this report we switch constantly between two views of a 'combinatorial system', viewing it as a system of combinatorial species [Joy81], or as a system of generating functions, corresponding to the counting of the former species.

Definition 1 (Combinatorial Species)**.** A species is a functor $\mathcal{F}\colon \mathbf{B} \to \mathbf{B}$, where $\mathbf{B}$ is the category of finite sets with bijections as morphisms. Given a finite set U, the elements of $\mathcal{F}[U]$ are called $\mathcal{F}$-structures on U. The size of an $\mathcal{F}$-structure s on U is the cardinality $|U|$ of U, and is denoted $|s|$.

Useful examples of species as defined in Definition 1 include

- The species SET of sets, defined by $\text{SET}[U] \triangleq U$.
- The species SEQ of sequences (also known as linear orderings in [Joy81]) is defined by $\text{SEQ}[\emptyset] \triangleq \{\emptyset\}$ and for $U = \{u_1, \ldots, u_n\}$ by $\text{SEQ}[U] \triangleq \{(u_{\sigma(1)}, \ldots, u_{\sigma(n)}) : \sigma\colon [n] \to [n] \text{ permutation}\}$
- The species CYC of cycles is defined by $\text{CYC}[\emptyset] \triangleq \emptyset$ and for $U = \{u_1, \ldots, u_n\}$ by $\text{CYC}[U] \triangleq \{\sigma\colon U \to U : \sigma \text{ is a permutation consisting of one cycle}\}$

Other ubiquitous species are: the *empty* species 0, defined by $0[U] = \emptyset$ for all U, the species $\mathcal{E}$ defined by $\mathcal{E}[U] = \emptyset$ if $U \neq \emptyset$ and $\mathcal{E}[\emptyset] = \{\emptyset\}$, and the species $\mathcal{Z}$, of singletons, defined by $\mathcal{Z}[U] = \{U\}$ if $|U| = 1$ and $\emptyset$ otherwise. Before relating species to generating functions, we need to define the corresponding ring operations over species.

Definition 2 (*Sum* and *product* of species)**.** Given two species $\mathcal{F}$ and $\mathcal{G}$ we define their *sum* $\mathcal{F}+\mathcal{G}$ to be the species satisfying $(\mathcal{F}+\mathcal{G})[U] = \mathcal{F}[U]+\mathcal{G}[U]$ where $+$ denotes the disjoint sum (in category

theory: *coproduct*). When summing multiple species we use the symbol Σ. The *product* of $\mathcal{F}$ and $\mathcal{G}$, written as $\mathcal{F} \cdot \mathcal{G}$ is defined by

$$(\mathcal{F} \cdot \mathcal{G})[U] = \sum_{U = U_1 \uplus U_2} \mathcal{F}[U_1] \times \mathcal{G}[U_2]\,,$$

where $\uplus$ denotes disjoint union, thus the sum $\sum_{U=U_1 \uplus U_2}$ runs over all decompositions of U into such a disjoint union.

Definition 3 (*Substraction* of species)**.** We say that $\mathcal{F}$ is a subspecies of $\mathcal{G}$ if and only if for all finite sets U we have $\mathcal{F}[U] \subset \mathcal{G}[U]$ and for any bijection $\sigma : U \to V$ we have $\mathcal{F}[\sigma] = \mathcal{G}[\sigma]\Big|_{\mathcal{F}[U]}$. In those conditions, we define the substraction $\mathcal{H} = \mathcal{G} - \mathcal{F}$ by the equation $\mathcal{G} = \mathcal{F} + \mathcal{H}$.

One last ingredient, the 'composition' of species.

Definition 4 (Composition of species)**.** Given two species $\mathcal{F}$ and $\mathcal{G}$, where $\mathcal{G}$ satisfies $\mathcal{G}[\emptyset] = \emptyset$ (no structures of size 0), we define the *composition* of $\mathcal{F}$ and $\mathcal{G}$, denoted by $\mathcal{F} \circ \mathcal{G}$ by

$$(\mathcal{F} \circ \mathcal{G})\,[U] = \sum_{\pi \text{ partition of } U} \mathcal{F}[\pi] \times \prod_{s \in \pi} \mathcal{G}[s]\,.$$

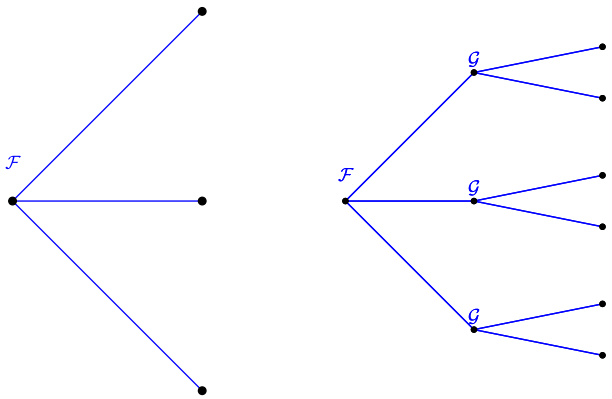

Figure 1.1: An $\mathcal{F}$-structure and a related $\mathcal{F} \circ \mathcal{G}$-structure.

Informally, what this definition tells us is that each $\mathcal{F}$-structure contains itself a set on which we apply a $\mathcal{G}$ structure (see Figure 1.1). We say two species $\mathcal{F}$ and $\mathcal{G}$ are equal if they produce the same sets and bijections. This, however, is not the appropriate notion of equality for us and we shall be more concerned about 'isomorphism'. We say $\mathcal{F}$ and $\mathcal{G}$ are isomorphic if and only if there exists a natural transformation between them; in other words, there exists a family $\alpha_U : \mathcal{F}[U] \to \mathcal{G}[U]$ of bijections (morphisms) that satisfies $\mathcal{G}[\sigma](\alpha_U(s)) = \alpha_V(\mathcal{F}[\sigma](s))$.

Informally, this means that there is a bijection between them that preserves all of the possible combinatorial structures and their relations. When $\mathcal{F}$ and $\mathcal{G}$ are isomorphic, we simply write $\mathcal{F} = \mathcal{G}$.

Remark 1.1. Note that we have the isomorphism $\mathcal{G}(\mathcal{Z}) = \mathcal{G}$ where $\mathcal{G}(\mathcal{Z})$ denotes $\mathcal{G} \circ \mathcal{Z}$. It may seem wasteful to write $\mathcal{Z}$ as an argument, however, this goes in parallel with the corresponding generating functions (for which z is the 'symbol' variable) that we present later on, and also, when

species	0	1	$\mathcal{Z}$	$\mathcal{A}+\mathcal{B}$	$\mathcal{A}\cdot\mathcal{B}$	SEQ	SET	CYC
derivative	0	0	1	$\mathcal{A}'+\mathcal{B}'$	$\mathcal{A}'\cdot\mathcal{B}+\mathcal{A}\cdot\mathcal{B}'$	SEQ $\cdot$ SEQ	SET	SEQ

Table 1.1: Derivatives of the species we will be working with.

setting up our combinatorial systems, it allows us to think about all of the species SET, SEQ and CYC as operators while of 0, $\mathcal{E}$ and $\mathcal{Z}$ as terminals.

Definition 5 (Unlabelled structures)**.** Two $\mathcal{F}$-structures s and t over $\{1,\ldots,n\}$ are said to be isomorphic if and only if there exists a permutation $\pi \in \mathcal{S}_n$ such that $\mathcal{F}[\pi](s) = t$. The isomorphism classes of $\mathcal{F}$-structures over $\{1,\ldots,n\}$ are called *unlabelled $\mathcal{F}$-structures* of size n.

Definition 6 (Derivative)**.** Given a combinatorial species $\mathcal{F}$, its derivative $\mathcal{F}'$ is defined by $\mathcal{F}'[U] := \mathcal{F}[U + \{\star\}]$ where $\star$ is supplementary symbol not in U. A table of the derivatives of the most common species is shown in Table 1.1.

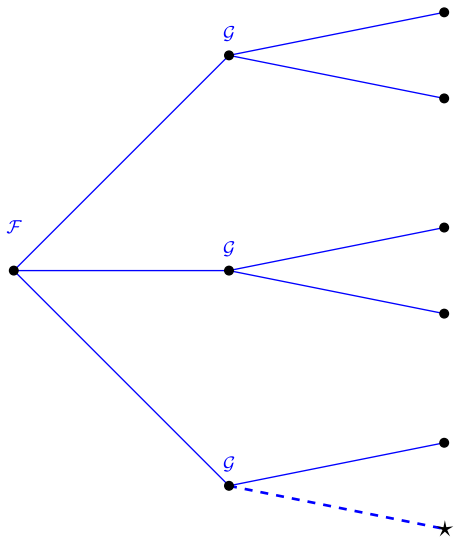

Figure 1.2: Chain rule for species.

Intuitively speaking, differentiating a species $\mathcal{F}(\mathcal{Z})$ amounts to filling one of the spaces reserved for a $\mathcal{Z}$ with a $\star$. For example, we have the chain rule $(\mathcal{F}\circ\mathcal{G})' = (\mathcal{F}'\circ\mathcal{G})\circ\mathcal{G}'$, and its intuitive explanation is as follows (see Figure 1.2): first choose the $\mathcal{F}$ branch where $\star$ is going to be, and then make the replacement of a single $\mathcal{Z}$ for $\star$ within this structure (this amounts to $\mathcal{G}'$).

Definition 7 (Cardinality restrictions)**.** Given a combinatorial species $\mathcal{F}$ and a nonnegative integer n, we define the species

1. $\mathcal{F}_{=n}$ by $\mathcal{F}_{=n}[U] = \emptyset$ if $|U| \neq n$ and $\mathcal{F}_{=n}[U] = \mathcal{F}[U]$ otherwise.
2. $\mathcal{F}_{\geq n}$ by $\mathcal{F}_{\geq n}[U] = \emptyset$ if $|U| < n$ and $\mathcal{F}_{\geq n}[U] = \mathcal{F}[U]$ otherwise.
3. $\mathcal{F}_{\leq n}$ by $\mathcal{F}_{\leq n}[U] = \emptyset$ if $|U| > n$ and $\mathcal{F}_{\leq n}[U] = \mathcal{F}[U]$ otherwise.

Given a species $\mathcal{F}$, there are several power series associated with it. The exponential power series $F(z)$ encodes the number of $\mathcal{F}$-structures on the set $\{1,\ldots,n\}$, producing the so-called labelled enumeration.

Definition 8 (Exponential Generating Function)**.** Let $\mathcal{F}$ be a combinatorial species, the exponential

generating function (EGF) of $\mathcal{F}$ is defined by the formal power series

$$F(z) \triangleq \sum_{n=0}^{\infty} f_n \frac{z^n}{n!}\,, \tag{1.1}$$

where $f_n \triangleq |\mathcal{F}[\{1,\ldots,n\}]|$.

This kind of counting is called labelled because re-arranging the elements of $\{1,\ldots,n\}$ by a permutation σ takes one $\mathcal{F}$-structure s to another (different one!) $\mathcal{F}[\sigma](s)$. even though the structures are 'essentially' the same, we count them as different due to our using a different numbering.

This motivates the unlabelled structures defined in Definition 5, which have their own adequate generating function

Definition 9 (Ordinary Generating Function)**.** Let $\mathcal{F}$ be a combinatorial species, the ordinary generating function (OGF) of $\mathcal{F}$ is defined by the formal power series

$$\tilde{F}(z) \triangleq \sum_{n=0}^{\infty} \tilde{f}_n \frac{z^n}{n!}\,. \tag{1.2}$$

where $\tilde{f}_n$ is the number of unlabelled $\mathcal{F}$-structures of size n, as defined in Definition 5.

Operating with the species defined above translates into generating functions in a simple way, which is shown in Table 1.2. Working with cardinality restrictions, as well as adding more variables to account for other numerical aspects of the constructions, is possible [FS09] and it is not explained in detail here for the sake of briefness.

Finally, to be able to set up a system and work with it, we need the concept of a *multisort species*. Roughly, a multisort species is the analogue of the concept of a function of several variables.

Definition 10 (Multisort species)**.** In the spirit of our definition of combinatorial species, a multisort species of k-variables is a k-functor from the product category $\mathbf{B}\times\ldots\times\mathbf{B}$, where $\mathbf{B}$ is the category of finite sets and bijections, to $\mathbf{B}$. When such a multisort species is applied to sets $U_1,\ldots,U_k$, the resulting set is denoted by $\mathcal{F}[U_1,\ldots,U_k]$ and its size is the sum of the cardinalities of the sets U_i.

Sums and products are easily extended to multisort species, while the composition with k singlesort species (see Definition 1) results in a singlesort species that is defined by partitioning the underlying set into k sets in a similar fashion. This is exemplified here for the case $k=2$

$$\mathcal{H}(\mathcal{G}_1,\mathcal{G}_2)[U] = \sum_{\pi \text{ partition of } U;\ \pi_1+\pi_2=\pi} \mathcal{H}[\pi_1,\pi_2]\times\prod_{s\in\pi_1}\mathcal{G}_1[s]\times\prod_{s\in\pi_2}\mathcal{G}_2[s]\,.$$

Definition 11 (Partial derivative)**.** Let $\mathcal{H}(\mathcal{Y}_1,\ldots,\mathcal{Y}_k)$ be a k-sort multisort species. We define

$$\frac{\partial\mathcal{H}}{\partial\mathcal{Y}_i}[U_1,\ldots,U_k] = \mathcal{H}[U_1,\ldots,U_{i-1},U_i+\{\star_i\},U_{i+1},\ldots,U_k]\,, \tag{1.3}$$

where $\star_i$ is a new symbol not present in U_i.

Partial differentiation of composition of species satisfy many of the desirable properties of the classical case in Real Analysis. For example, when composing $\mathcal{F}(\mathcal{X},\mathcal{Y})$ with two singlesort species $\mathcal{G}_1$ and $\mathcal{G}_2$ we have

$$(\mathcal{F}(\mathcal{G}_1,\mathcal{G}_2))' = \frac{\partial\mathcal{F}}{\partial\mathcal{X}}\cdot\mathcal{G}_1' + \frac{\partial\mathcal{F}}{\partial\mathcal{Y}}\cdot\mathcal{G}_2'\,.$$

We will use **bold**-case font to mean a vector of 'something'. Concretely, The above $\mathcal{H}(\mathcal{Y}_1,\ldots,\mathcal{Y}_k)$ in Definition 11 is denoted also by $\mathcal{H}(\boldsymbol{\mathcal{Y}})$. Similarly, when we have an array of multisort species $\mathcal{H}_1(\mathcal{Y}_1,\ldots,\mathcal{Y}_k),\ldots,\mathcal{H}_m(\mathcal{Y}_1,\ldots,\mathcal{Y}_k)$ we denote it by $\boldsymbol{\mathcal{H}}(\boldsymbol{\mathcal{Y}})$ for short, being implicit the corresponding lengths of the vectors.

operation	EGF	OGF
$\mathcal{F}+\mathcal{G}$	$F(z)+G(z)$	$F(z)+G(z)$
$\mathcal{F}\cdot\mathcal{G}$	$F(z)\cdot G(z)$	$F(z)\cdot G(z)$
$\textsc{Seq}(\mathcal{G})$	$\frac{1}{1-G(z)}$	$\frac{1}{1-G(z)}$
$\textsc{Set}(\mathcal{G})$	$\exp(G(z))$	$\exp\left(\sum_{k\geq 0}\frac{G(z^k)}{k}\right)$
$\textsc{Cyc}(\mathcal{G})$	$\log\left(\frac{1}{1-G(z)}\right)$	$\sum_{k\geq 0}\frac{\varphi(k)}{k}\log\left(\frac{1}{1-G(z^k)}\right)$

Table 1.2: Typical species compositions and their corresponding generating functions. Here $F(z)$ and $G(z)$ denote the corresponding (EGF or OGF) generating functions of $\mathcal{F}$ and $\mathcal{G}$ respectively, and $\varphi(k)$ is Euler's totient function, counting the number of integers coprime to k in $\{1,\ldots,k\}$.

2 Well-founded systems

The first stage of the internship consisted in both, getting used to Maple and programming the algorithm for deciding whether a system of combinatorial species is well-founded described in [PSS12]. In this context, the concept of well-foundedness is defined in terms of the convergence of the fixed point iteration given by the system.

Definition 12 (Convergence of a sequence of species)**.** A sequence $\left(\mathcal{Y}^{[n]}\right)_{n\in\mathbb{N}}$ converges to a species $\mathcal{Y}$ if and only if for all $k\geq 0$ there exists $N\in\mathbb{N}$ such that for all $n\geq N$ we have $\mathcal{Y}^{[n]}_{\leq k}=\mathcal{Y}_{\leq k}$.

Definition 13 (Well-founded combinatorial system)**.** We say that a system $\boldsymbol{\mathcal{Y}}=\boldsymbol{\mathcal{H}}(\boldsymbol{\mathcal{Z}},\boldsymbol{\mathcal{Y}})$ is well-founded if and only if the iteration defined by $\boldsymbol{\mathcal{Y}}^{[0]}=\mathbf{0}$ and

$$\mathcal{Y}^{[k+1]}=\boldsymbol{\mathcal{H}}(\boldsymbol{\mathcal{Z}},\boldsymbol{\mathcal{Y}}^{[k]})\,,\qquad\qquad(k\geq 0)\tag{2.1}$$

is well-defined and converges to a species $\boldsymbol{\mathcal{Y}}^{[\infty]}$, with no zero coordinates.

Observe that the restriction about the 'zero coordinates' is not really a restricting one, since one can actually detect which coordinates always remain $\mathbf{0}$, and we can simply remove them from the system.

Historically, a first sufficient criterion to assert the existence of a species satisfying a system of equations *and* compute its solution was given by A. Joyal in [Joy81]. This criterion mimics in some way the classical Implicit Function Theorem, and, due to its simplicity, it was extended in a similar fashion in [PSS12] to a general criterion that is both necessary and sufficient to assert the well-foundedness of a system in the sense of Definition 13.

Theorem 1 (Joyal's Implicit Species Theorem)**.** *Let $\boldsymbol{\mathcal{H}}$ be a vector of multisort species, with $\boldsymbol{\mathcal{H}}(\mathbf{0},\mathbf{0})=\mathbf{0}$ and such that the Jacobian $\partial\boldsymbol{\mathcal{H}}/\partial\boldsymbol{\mathcal{Y}}(\mathbf{0},\mathbf{0})$ is nilpotent. Then the system of equations*

$$\boldsymbol{\mathcal{Y}}=\boldsymbol{\mathcal{H}}(\boldsymbol{\mathcal{Z}},\boldsymbol{\mathcal{Y}})\,,\qquad\boldsymbol{\mathcal{Y}}=(\mathcal{Y}_1,\ldots,\mathcal{Y}_m)\,,\tag{2.2}$$

admits a solution $\boldsymbol{\mathcal{S}}$ such that $\boldsymbol{\mathcal{S}}(\mathbf{0})=\mathbf{0}$, which is unique up to isomorphism. Moreover, the sequence

$$\boldsymbol{\mathcal{Y}}^{[k+1]}=\boldsymbol{\mathcal{H}}(\boldsymbol{\mathcal{Z}},\boldsymbol{\mathcal{Y}}^{[k]})\,,\qquad\qquad(k\geq 0)\tag{2.3}$$

starting from $\boldsymbol{\mathcal{Y}}^{[0]}=\mathbf{0}$, converges to a solution.

Informally, the Jacobian matrix $\boldsymbol{\partial\mathcal{H}}/\boldsymbol{\partial\mathcal{Y}}(\boldsymbol{\mathcal{Z}},\boldsymbol{\mathcal{Y}})$ gives the dependencies between the species $\mathcal{Y}_i$, while $\boldsymbol{\partial\mathcal{H}}/\boldsymbol{\partial\mathcal{Y}}(\mathbf{0},\mathbf{0})$ gives the dependencies through paths of zero size. Hence, having $\boldsymbol{\partial\mathcal{H}}/\boldsymbol{\partial\mathcal{Y}}(\mathbf{0},\mathbf{0})$ nilpotent tells us that there are no 0-size loops (these would allow us to generate infinitely many structures without increasing the size) in the 'dependency graph'.

2.1 Well-foundedness at 0

Before delving into the necessary and sufficient conditions for a system to be well-founded, it is helpful and simple to start with the case $\boldsymbol{\mathcal{H}}(\mathbf{0},\mathbf{0}) = \mathbf{0}$. Such systems are called well-founded at $\mathbf{0}$ when well-founded in the sense defined in Definition 13.

The solution consists of two steps [PSS12, 3.2]. First the decision of whether there is a coordinate that will always remain 0 is achieved by Algorithm 1, and finally deciding whether it is indeed well-founded reduces to deciding whether the Jacobian $\boldsymbol{\partial\mathcal{H}}/\boldsymbol{\partial\mathcal{Y}}(\mathbf{0},\mathbf{0})$ is nilpotent, thus getting Algorithm 2. This ensures that it is not possible to generate infinitely many structures of any given size.

Algorithm 1 Algorithm to decide if there is a 0 coordinate.

Input: a vector of species $\boldsymbol{\mathcal{H}} = (\mathcal{H}_1, \ldots, \mathcal{H}_m)$ such that $\boldsymbol{\mathcal{H}}(\mathbf{0},\mathbf{0}) = \mathbf{0}$.
Output: a set S consisting of the indices of the coordinates that will always remain 0 when applying the iteration (2.1) to the system $\boldsymbol{\mathcal{H}}$ starting from $\boldsymbol{\mathcal{Y}}^{[0]} = \mathbf{0}$.

```
Compute 𝒴^[m] = ℋ^m(𝒵, 0)  ▷ Here ℋ^{k+1}(𝒵, 𝒴) ≜ ℋ(𝒵, ℋ^k(𝒵, 𝒴)) and ℋ^0(𝒵, 𝒴) ≜ 𝒴.
S ← ∅
for each coordinate 𝒴_i^[m] in 𝒴^[m] do
    if 𝒴_i^[m] = 0 then S ← S ∪ {i}
return S.
```

Algorithm 2 Algorithm to decide if a system is well-founded at 0.

Input: a vector of species $\boldsymbol{\mathcal{H}} = (\mathcal{H}_1, \ldots, \mathcal{H}_m)$ such that $\boldsymbol{\mathcal{H}}(\mathbf{0},\mathbf{0}) = \mathbf{0}$.
Output: Answer to 'Is the system $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{H}}(\boldsymbol{\mathcal{Z}}, \boldsymbol{\mathcal{Y}})$ well-founded at $\mathbf{0}$?

```
Compute 𝒥 = ∂ℋ/∂𝒴(0, 0)
if 𝒥^m = 0_{m×m} then
    return Call Algorithm 1
else
    return NO
```

In practice we exploit the morphism between species and EGFs; the Jacobian is computed from a system of generating functions in the single variable z. Once we know a term is nonzero, we can just replace it with a 1, since, by positivity of the coefficients of the generating functions, this will not change the final decision $\boldsymbol{\mathcal{J}}^m = \mathbf{0}_{m\times m}$. Finally, instead of raising the matrix to the power of m, it is faster to multiply $\boldsymbol{\mathcal{J}}$ by a vector of 1s while it is not $\mathbf{0}$ (at each iteration we have m^2 multiplications), and stopping if it is not $\mathbf{0}$ after m iterations. Again, this works by positivity.

2.2 General Implicit Species Theorem

In general, many combinatorial systems do not satisfy the condition $\boldsymbol{\mathcal{H}}(\mathbf{0},\mathbf{0}) = \mathbf{0}$, e.g. consider

$$\mathcal{Y} = \mathcal{H}(\mathcal{Z}, \mathcal{Y}) \triangleq 1 + \mathcal{Z}\,\mathcal{Y}\,, \tag{2.4}$$

which defines $\mathcal{Y} = \textsc{Seq}(\mathcal{Z})$, however, we have $\mathcal{H}(0,0) = 1$.

The issue is solved by introducing a modified system, known as the companion system [PSS12], which does satisfy that the value at $(\mathbf{0},\mathbf{0})$ is $\mathbf{0}$.

Definition 14 (Companion system)**.** Let $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{H}}(\boldsymbol{\mathcal{Z}}, \boldsymbol{\mathcal{Y}})$ be a system, its companion system is defined by

$$\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{K}}\left(\mathcal{Z}_1, \boldsymbol{\mathcal{Z}}, \boldsymbol{\mathcal{Y}}\right) , \qquad \boldsymbol{\mathcal{K}} \triangleq \boldsymbol{\mathcal{H}}\left(\boldsymbol{\mathcal{Z}}, \boldsymbol{\mathcal{Y}}\right) - \boldsymbol{\mathcal{H}}\left(\mathbf{0}, \mathbf{0}\right) + \mathcal{Z}_1 \, \boldsymbol{\mathcal{H}}\left(\mathbf{0}, \mathbf{0}\right) , \tag{2.5}$$

where $\mathcal{Z}_1$ is a new species variable different from all of those appearing in $\boldsymbol{\mathcal{Z}}$.

We state here the Theorem [PSS12, Theorem 5.5] providing the link between the well-foundedness of $\boldsymbol{\mathcal{H}}$ and the well-foundedness of $\boldsymbol{\mathcal{K}}$ at $\mathbf{0}$. For this, however, we will first have to introduce the notion of a partially-polynomial species.

Definition 15 (Partially-polynomial)**.** A multisort species $\mathcal{F}(\boldsymbol{\mathcal{Z}}_1, \boldsymbol{\mathcal{Z}}_2)$ is polynomial in $\boldsymbol{\mathcal{Z}}_1$ if and only if for all $n \geq 0$, the species $\mathcal{F}_{=(.,n)} = \sum_{k \geq 0} \mathcal{F}_{=(k,n)}$ if polynomial.

Theorem 2. *Let $\boldsymbol{\mathcal{H}} = (\mathcal{H}_1, \ldots, \mathcal{H}_m)$ be a vector of species. The combinatorial system $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{H}}\left(\boldsymbol{\mathcal{Z}}, \boldsymbol{\mathcal{Y}}\right)$ is well-founded if and only if*

- *the companion system $\boldsymbol{\mathcal{K}}$ is well-founded at $\mathbf{0}$, and*
- *the species $\boldsymbol{\mathcal{S}}_1(\mathcal{Z}_1, \boldsymbol{\mathcal{Z}})$, solution of $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{K}}\left(\mathcal{Z}_1, \boldsymbol{\mathcal{Z}}, \boldsymbol{\mathcal{Y}}\right)$ with $\boldsymbol{\mathcal{S}}_1(0, \mathbf{0}) = \mathbf{0}$, is polynomial in the variable $\mathcal{Z}_1$.*

In such case the solution to the system is simply $\boldsymbol{\mathcal{S}}_1(1, \boldsymbol{\mathcal{Z}})$.

It remains here to explain how we decide whether the solution to the companion system is polynomial in $\mathcal{Z}_1$ or not. To do so, we use Algorithm 3 [PSS12, pp. 17-18].

Algorithm 3 Algorithm to decide whether the solution to a system is partially polynomial.

Input: a vector of species $\boldsymbol{\mathcal{H}} = (\mathcal{H}_1, \ldots, \mathcal{H}_m)$ such that $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{H}}(\mathcal{Z}_1, \boldsymbol{\mathcal{Z}}_2, \boldsymbol{\mathcal{Y}})$ is well-founded at $\mathbf{0}$.
Output: Answer to 'Is the solution of $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{H}}(\mathcal{Z}_1, \boldsymbol{\mathcal{Z}}_2, \boldsymbol{\mathcal{Y}})$ polynomial in $\mathcal{Z}_1$? '

```
S_0 := 0
for i ∈ {1, ..., m} do
    S_0 := H(Z_1, 0, S_0)
if H(Z_1, 0, S_0) ≠ S_0 then return NO
else
    J_0 := ∂H/∂Y(Z_1, 0, S_0(Z_1))
    if J_0^m ≠ 0_{m×m} then return NO
    else
        for i ∈ {1, ..., m} do
            if (S_0)_i ≠ 0 and H is not polynomial in Y_i then return NO
        return YES
```

Several comments are in order. First $\boldsymbol{\mathcal{S}}_0$, after the first loop, equals to $\boldsymbol{\mathcal{S}}(\mathcal{Z}_1, \mathbf{0})$ if the solution $\boldsymbol{\mathcal{S}}$ is defined and polynomial in $\mathcal{Z}_1$. This is a consequence of [PSS12, Proposition 4.2] which tells us that the solution $\boldsymbol{\mathcal{S}}_0$ of an implicit and unisort system $\boldsymbol{\mathcal{Y}} = \tilde{\boldsymbol{\mathcal{H}}}(\mathcal{Z}_1, \boldsymbol{\mathcal{Y}})$ satisfying the hypothesis of Theorem 1 is polynomial (meaning that $(\boldsymbol{\mathcal{S}}_0)_{\geq N+1} = \mathbf{0}$ for large enough N) if and only if the iteration in Theorem 1 remains constant after m iterations, where m is the length of the vector $\boldsymbol{\mathcal{Y}}$. In the current case we have $\tilde{\boldsymbol{\mathcal{H}}}(\mathcal{Z}_1, \boldsymbol{\mathcal{Y}}) := \boldsymbol{\mathcal{H}}(\mathcal{Z}_1, \mathbf{0}, \boldsymbol{\mathcal{Y}})$.

Second, notice that the check '$\boldsymbol{\mathcal{H}}$ is not polynomial in $\mathcal{Y}_i$' is not actually as simple as checking whether we have an *actual* polynomial in $\mathcal{Y}_i$, because we have to use Definition 15. Indeed, for example $\textsc{Seq}(\mathcal{Y}_1 + \mathcal{Z}_1)$ is not polynomial in $\mathcal{Y}_1$, but $\textsc{Seq}(\mathcal{Y}_1\, \mathcal{Z}_1)$ is.

However, since we are talking about an expression built out of the constructors $\mathcal{E}$, $\mathcal{Z}_1$, $+$, $\times$, SEQ, CYC, SET, MSET and PSET, and the symbols $\mathcal{Y}_1, \ldots, \mathcal{Y}_m$, instead of the solution to an implicit system like in

Algorithm 3, we can solve the problem directly by using the combinatorial properties of our constructors inductively. This is made explicit as follows

Proposition 2.1 (Characterization of partially polynomial expressions)**.** *If $\mathcal{P}(\mathcal{Z}, \mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_m)$ and $\mathcal{Q}(\mathcal{Z}, \mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_m)$ are nonzero species, depending on the species variables $\mathcal{Y}_1, \ldots, \mathcal{Y}_m$, then we have that*

- *the sum $\mathcal{P} + \mathcal{Q}$ and the product $\mathcal{P} \times \mathcal{Q}$ are polynomial in $\mathcal{Y}_1$.*
- *Let $\mathcal{G} \in \{\textsc{Set}_{\leq k}, \textsc{Seq}_{\leq k}, \textsc{Cyc}_{\leq k}, \textsc{MSet}_{\leq k}, \textsc{PSet}_{\leq k}\}$, then $\mathcal{G}(\mathcal{P})$ is polynomial in $\mathcal{Y}_1$.*
- *Let $\mathcal{G} \in \{\textsc{Set}_{=k}, \textsc{Seq}_{=k}, \textsc{Cyc}_{=k}, \textsc{MSet}_{=k}, \textsc{PSet}_{=k}\}$, then $\mathcal{G}(\mathcal{P})$ is polynomial in $\mathcal{Y}_1$.*
- *Let $\mathcal{G} \in \{\textsc{Set}, \textsc{Seq}, \textsc{Cyc}, \textsc{MSet}, \textsc{PSet}\}$, then $\mathcal{G}(\mathcal{P})$ is polynomial in $\mathcal{Y}_1$ if and only if $\mathcal{P}(\mathbf{0}, \mathcal{Y}_1, \mathbf{0}, \ldots, \mathbf{0}) = \mathbf{0}$.*
- *Let $\mathcal{G} \in \{\textsc{Set}_{\geq k}, \textsc{Seq}_{\geq k}, \textsc{Cyc}_{\geq k}, \textsc{MSet}_{\geq k}, \textsc{PSet}_{\geq k}\}$, then $\mathcal{G}(\mathcal{P})$ is polynomial in $\mathcal{Y}_1$ if and only if $\mathcal{P}(\mathbf{0}, \mathcal{Y}_1, \mathbf{0}, \ldots, \mathbf{0}) = \mathbf{0}$.*

Proof. We explain here why, if $\mathcal{G} \in \{\textsc{Set}, \textsc{Seq}, \textsc{Cyc}, \textsc{MSet}, \textsc{PSet}\}$, then $\mathcal{G}(\mathcal{P})$ is polynomial in $\mathcal{Y}_1$ if and only if $\mathcal{P}(\mathbf{0}, \mathcal{Y}_1, \mathbf{0}, \ldots, \mathbf{0}) = \mathbf{0}$.

($\Longrightarrow$) Assume $\mathcal{A}(\mathcal{Y}_1) = \mathcal{P}(\mathbf{0}, \mathcal{Y}_1, \mathbf{0}, \ldots, \mathbf{0}) \neq \mathbf{0}$. Then we immediately have that $\mathcal{G}(\mathcal{A}) \subset \mathcal{G}(\mathcal{P})$, and $\mathcal{G}(\mathcal{A})$ however, is not polynomial in $\mathcal{Y}_1$, which is a contradiction.

($\Longleftarrow$) This is a particular case of [PSS12, Lemma 4.6]. □

2.3 Final comments

The algorithm shown in this section replaces that introduced in [Zim91, Section 1.4], which was implemented in the CombStruct package in Maple. Its implementation is, on the whole, conceptually simple, in the sense that it does not involve any complex algorithms, and it readily covers cases involving cardinality constraints that resulted in the program refusing a well-founded grammar.

The code implemented has been tested extensively, considering combinatorial systems that already worked with the previous implementation, and that did not work well previously. One of the possible drawbacks of the current implementation is that, even though it is a first working solution, it repeatedly uses both power series (to exploit the automatic maple reductions) and combinatorial expressions.

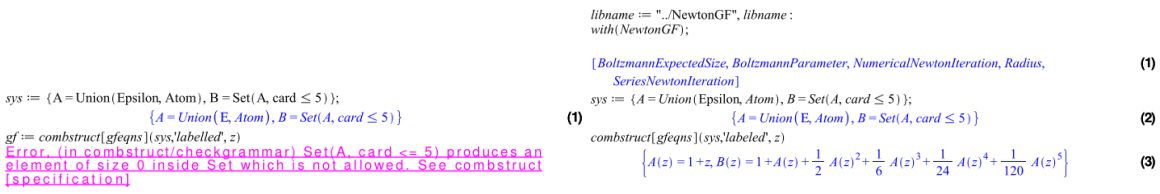

Figure 2.1: Example of a well-founded combinatorial system that previously did not work.

3 Newton-Raphson for formal power series

Given a formal power series $F(z) = \sum_{k \geq 0} f_k z^k \in R[\![z]\!]$ with $f_0 = 0_R$, where R is a commutative unital ring of characteristic 0, there are certainly very simple ways to compute $E(z) = \exp(F(z)) \in$

$R[\![z]\!]$, for instance by using the equation

$$E(z) = 1 + \int F'(z)\, E(z) dz\,,$$

which, for $n \geq 1$, translates into

$$e_n = \frac{1}{n} \sum_{k=0}^{n-1} e_k\, (n-k)\, f_{n-k}\,.$$

However, this naive implementation results in an algorithm that requires an order of $\Theta(N^2)$ arithmetic operations over R to compute $e_N = [z^N]E(z)$, by saving all of the $e_0, \ldots, e_{N-1}$ in a table as we compute them.

The exponential, however, can be computed more efficiently by using the principle of the so-called 'Newton iteration' as long as we have a fast multiplication algorithm for polynomials available [Bos10]. Maple adapts the algorithm it uses to multiply polynomials according to their degree (e.g. algorithms that are better asymptotically, such as Karatsuba's or the FFT can have 'large constants' when compared to the trivial one, leading to worse performance for smaller inputs), achieving a multiplication better than the trivial one, and hence we expected to obtain gains by implementing our own version of $\exp$ for formal power series, since, interestingly, Maple did not appear compute $\exp(F(z))$ by a Newton-like iteration.

Supposing $M(N)$ is an upper bound for the number of arithmetic operations that are needed to compute the product of two polynomials of degree N over a commutative unital ring R of characteristic 0, the time it takes to compute the exponential of formal power series up to degree by using the Newton-like version below is $O(M(N))$, provided that $M(n+n') \leq M(n) + M(n')$ is satisfied [Bos10], and this is also true for the generic Newton iteration for formal power series.

3.1 Computing the exponential of a formal series

Newton iteration

In the generic Newton Iteration, given $\Phi(X, Y) \in R[\![X, Y]\!]$, and the implicit equation $\Phi(X, F) = 0$ we solve for $F \in R[\![X]\!]$ by using the iteration

$$\mathcal{N}(S) = S - \frac{\Phi(X, S)}{\partial_Y \Phi(X, S)} \tag{3.1}$$

repeatedly, starting from $F_0 = 0$. The important property of this iteration is that, under the right conditions, if the solution is F and $F_n = F + O(X^n)$, then we have $\mathcal{N}(F_n) = F + O(X^{2\,n})$, thus the number of right coefficients doubles with each iteration.

Theorem 3 (Newton iteration)**.** *Let R be a commutative unital ring, satisfying char$(R) = 0$, and let $\Phi \in R[\![X, Y,]\!]$ be such that $\Phi(0,0) = 0$ and $\partial_Y \Phi(0,0)$ is a unit in R. Then there is a unique series $S \in R[\![X]\!]$ such that $\Phi(X, S) = 0$ and $S(0) = 0$. Moreover, if F is such that $S - F = O(X^N)$, then $\mathcal{N}(S) - F = O(X^{2\,n})$.*

Example 3.1 (Fast reciprocal)**.** An example that will come in handy later on in the report is that of computing the multiplicative inverse of $G \in R[\![X]\!]$ with $G(0) = 1_R$. Consider $\Phi(X, Y) = (1_R + Y)^{-1} - G$. Such a $\Phi(X, Y)$ indeed satisfies the condition of Theorem 3, and moreover, the unique solution $F \in R[\![X]\!]$ satisfies $F(0) = 0$, and $1_R + F$ is the inverse of G.

The Newton iteration, when explicitly written out is simply

$$\mathcal{N}(S) = S - \frac{(1_R + S)^{-1} - G}{-(1_R + S)^{-2}} = S + (1_R + S)\ (1_R - (1_R + S)\, G)\ ,$$

and thus, with minor changes, we get the following recursive algorithm:

Algorithm 4 Compute the reciprocal of a formal power series.

Input: an integer $N > 0$ and a truncated series $G \bmod .X^N$, with $G(0)$ a unit.
Output: the inverse $G^{-1} \bmod .X^N$.
  if $N = 1$ **then**
    return g_0^{-1}, where $g_0 = G(0)$
  else
    Compute recursively the inverse S of G module $X^{\lceil N/2 \rceil}$
    return $S + S\,(1 - S\,G)$ module X^N.

Remark 3.1. Here it is important to observe that $1 - S\,G = O(X^{\lceil N/2 \rceil})$, and so the term $S\,(1 - S\,G)$ can only affect the coefficients of X^i with $i \geq \lceil N/2 \rceil$.

Computing $\exp(F)$

The idea when computing the exponential of a formal power series will be similar, however here our equations will involve either an integral or a derivative. Let us suppose we have computed S_N satisfying $S_N = \exp(F) \bmod X^N$, we will show how to compute $\mathcal{N}(S) = \exp(F) \bmod X^{2\,N}$.

Inspired by Remark 3.1, we try to find an appropriate approximation G_N of the G defined by

$$\exp(F) = S_N - S_N \cdot G\,, \tag{3.2}$$

such that $G_N = G \bmod X^{2N}$.

Differentiating both sides

$$F'\ \exp(F) = S'_N - S'_N \cdot G - S_N\, G'\,,$$

which equals, by using (3.2),

$$F'\,(S_N - S_N \cdot G) = S'_N - S'_N \cdot G - S_N\, G'\,,$$

that is

$$G' = \left(\frac{S'_N}{S_N} - F'\right) + \left(F' - \frac{S'_N}{S_N}\right) G\,,$$

but here observe that $\frac{S'_N}{S_N} - F' = O(X^{N-1})$ and $G = O(X^N)$, thus $\left(F' - \frac{S'_N}{S_N}\right)\ G = O(X^{2N-1})$ and in fact we will be done if we pick

$$\begin{aligned} G_N &= \int \left(\frac{S'_N}{S_N} - F'\right) dX \mod X^{2N} \\ &= \int \left(\frac{S'_N}{S_N} - \left(F' \bmod X^{2N}\right)\right) dX \mod X^{2N}\,. \end{aligned}$$

even though computing S_N^{-1} can be done fast as we have already seen, this method may be improved by updating an approximation of $\exp(-F) = \exp(F)^{-1}$ in each iteration (see [Bos10, Exercise 8] and also [BCO$^+$07, Figure 1]) leading to the following algorithm

Algorithm 5 Compute the exponential of a formal power series.

Input: an integer $N > 0$ and a truncated series $F_N = F \bmod .X^N$, with $F(0)$ a unit.
Output: a pair $(\exp(F) \bmod X^N, \exp(-F) \bmod X^{\lceil N/2 \rceil})$ of truncated series.

if $N = 1$ **then**
  return $(1, 1)$.
else
  Compute recursively (A, B), the result $F_{\lceil N/2 \rceil} \leftarrow F_N \bmod X^{\lceil N/2 \rceil}$ and $N \leftarrow \lceil N/2 \rceil$.
  Compute $C = B + B\,(1 - A\,B) \mod X^{\lceil N/2 \rceil}$, which satisfies $C = \exp(-F) \bmod X^{\lceil N/2 \rceil}$.
  return $(A - A \int C\,(A' - F'_N\,A)dX \mod X^N, C)$.

This algorithm corresponds to the procedure 'goExp' in my Maple implementation.

3.2 Experimental results and further improvements

The algorithm shown above should, and does, result in time savings for labelled systems involving SET as can be seen in Figure 3.1. One could stop here and think 'problem solved', however, through extensive testing I found that systems involving CYC did not seem to be as fast as I expected them to be (see Figure 3.2). This led to my implementing a Newton iteration for the logarithm, which can be done simply by exploiting what we know from Example 3.1 in view of

$$\log(F) = \int \frac{F'}{F} dX\,. \tag{3.3}$$

This is implemented in the Maple package by the functions 'goRecip' that computes the series inverse, and 'series/ln' which then uses 'goRecip' to compute the logarithm of a formal power series. The resulting efficiency boost can be appreciated in Figure 3.2.

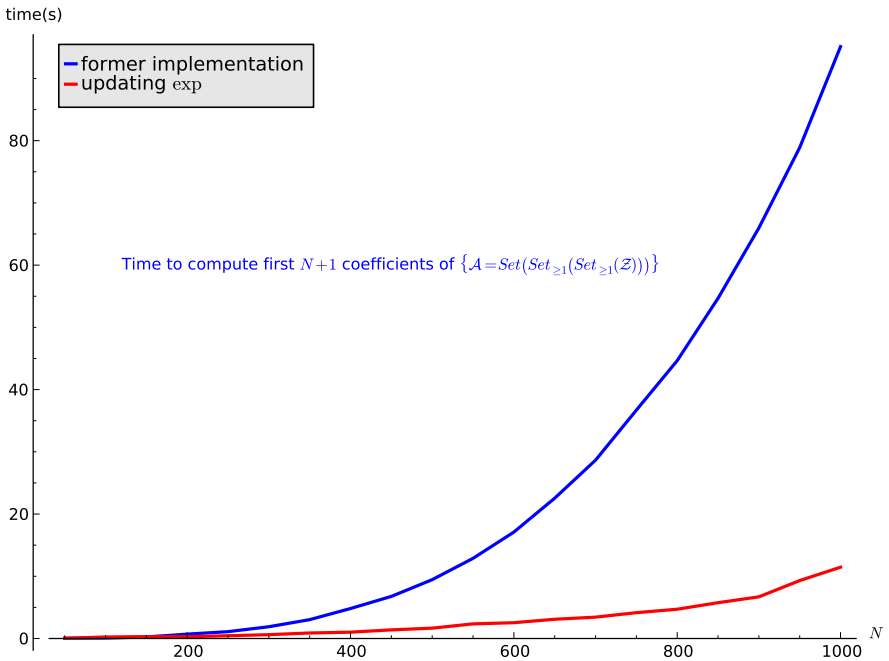

Figure 3.1: Comparison of the running times before and after changing the exponential.

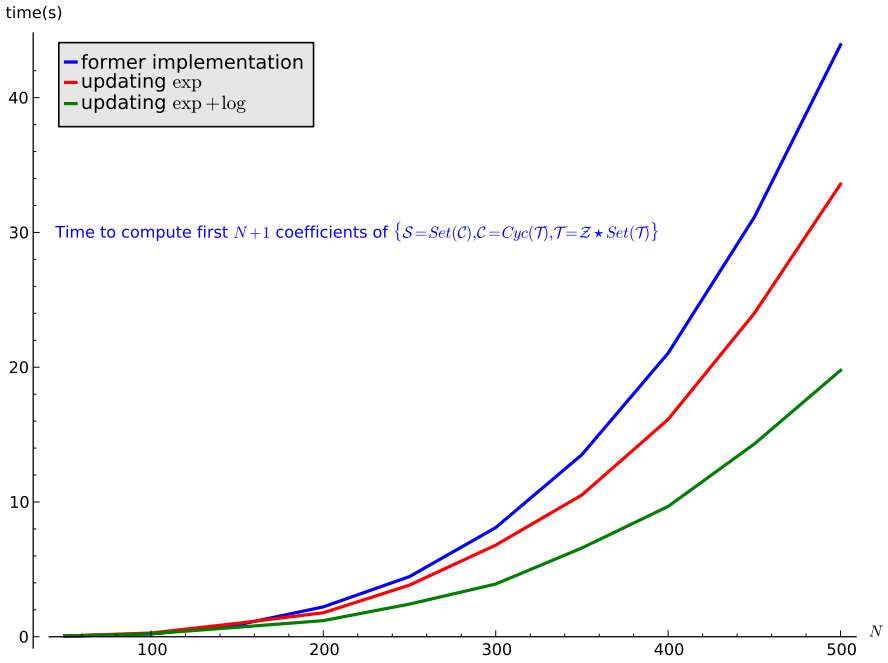

Figure 3.2: Comparison of the running times before and after changing the exponential, and the logarithm.

Moving on, by testing I also found that the way in which Maple computes the multiplicative inverses of formal power series is not fast either. This problem becomes apparent when we want to compute coefficients of systems that make extensive use of the operator SEQ. This problem is solved simply by, when translating the combinatorial system into generating functions, defining a new operator 'QuasiInverse' whose series expansion is computed by using the previously implemented 'goRecip', and whose derivative[1] is defined to be

$$(\textsc{QuasiInverse}(f))' \triangleq f' \cdot (\textsc{QuasiInverse}(f))^2 \ . \tag{3.4}$$

The reason why we do not re-implement the series inverse in this case is that this is not possible; this function lies in the kernel of Maple.

When it comes to the translation from combinatorial systems to generating functions, SEQ is the only operator that involves a multiplicative inverse properly speaking. However the operator CYC was implemented, in both the labelled and the unlabelled case, by writing terms of the form $\log\left(\frac{1}{1-\ldots}\right)$. This can be rewritten by exploiting the identity $\log(1/F) = -\log(F)$, thus removing the need to compute an inverse.

At this point, by experimenting with some unlabelled systems I reached the contradictory conclusion that after these changes the coefficient computations seemed to run slower. Then I noticed that the problem lied in the way the expansions were computed for the Pólya operators [FS09, Theorem I.2].

To explain the problem and the solution, we pick the concrete example of $\textsc{Cyc}(\mathcal{F})$, which translates into the OGF

$$\mathrm{Log}[f] = \sum_{k\geq 1} \frac{\varphi(k)}{k} \log\left(\frac{1}{1-f(z^k)}\right) \tag{3.5}$$

where f is the OGF of $\mathcal{F}$, and φ is Euler's totient function. Here it is evident how $\log(1/F) = -\log(F)$ helps, but the issue was that the term $\log\left(\frac{1}{1-f(z^k)}\right)$ was completely recomputed for each

[1] This is necessary since the package is actually going to perform a Newton iteration over the system.

k, thus I reimplemented it by computing $G(z) = -\log(1 - f(z))$ just once. The idea is that then computing $\mathrm{Log}[f]$ mod. z^N simply amounts to adding over $1 \leq k \leq N$ the terms $\sum_{j=0}^{\lfloor N/k \rfloor} g_j z^{j\,k}$, where $g_j = [z^j]G(z)$ are the coefficients of G, multiplied by the weights $\frac{\varphi(k)}{k}$.

3.3 Section summary

The following list is a summary of the code changes corresponding to this section:

- Newton-like iteration for $\exp(F)$.
- Newton iteration for $\log(F)$.
- Newton iteration for $F \mapsto (1 - F)^{-1}$, through the introduction of 'QuasiInverse'.
- Eliminated the need to compute inverses when having cycles $\log\left(\frac{1}{1-F}\right) = -\log(1 - F)$.
- Eliminated unnecessary recomputations when working with Pólya Operators.

A comparison of the running time for the above systems can be seen in Figure 3.3.

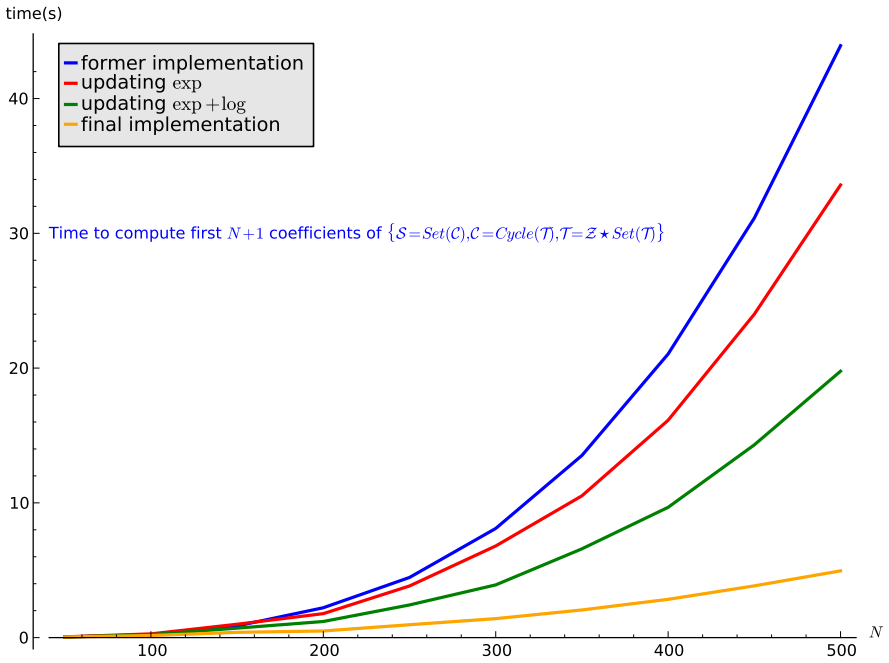

Figure 3.3: Running time comparison for a labelled system.

Final comments and ideas

Although there is a clear advantage in using the new code for large N, it is fair to say that for the very small N the performance may sometimes be a little slower. This is a point that **may** be improved by working out the small cases separately and using a more direct algorithm (say $\Theta(N^2)$).

Finally, it might be possible to improve the performance further for some systems involving

$$\textsc{QuasiInverse}(\textsc{QuasiInverse}_{\geq k}(A(X)))\,, \tag{3.6}$$

by writing it as $(1-A(X))\cdot \text{QUASIINVERSE}(A(X)+(A(X))^k)$. Of course, other similar simplifications involving $\log$ and the QuasiInverse are also possible. However, these kind of optimisations are not all that important now that QUASIINVERSE is done by a Newton iteration.

4 Computing the radius of convergence

Given a combinatorial system $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{H}}(\mathcal{Z}, \boldsymbol{\mathcal{Y}})$, and $\boldsymbol{y} = \boldsymbol{\Phi}(z, \boldsymbol{y})$, the translation of $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{H}}(\mathcal{Z}, \boldsymbol{\mathcal{Y}})$ into generating functions, the radius of convergence ρ of the system is minimum of the radius of convergence of the generating functions $y_1(z), \ldots, y_m(z)$. The radius of convergence determines the 'rough' asymptotic growth of the coefficients when the system satisfies certain properties [FS09].

The implementation of the function 'Radius' in NewtonGF computes the radius of convergence of a given system of combinatorial specifications by resorting to a binary search. The idea is that, when $t \in [0, \rho)$, doing a numerical Newton iteration for $\boldsymbol{y} = \Phi(z, \boldsymbol{y})$ with $z = t$ converges to the solution $\boldsymbol{y}(z)$ due to [PSS12, Lemma 9.12], while, somewhat heuristically, this same iteration is assumed to diverge when $t > \rho$. In practice, this 'diverges' is tested by looking for signs of convergence within a fixed number of iterations of the Newton iteration.

Such a binary search does not converge very fast to ρ, in fact, the number of correct digits increases just linearly, whereas the number of correct digits almost doubles with each iteration when we have a converging numerical Newton iteration. We are interested, then, in converging Newton iterations that give the convergence radius ρ of a system $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{H}}(\mathcal{Z}, \boldsymbol{\mathcal{Y}})$. To get the system to be $(m+1)\times(m+1)$, we must add one equation more to those corresponding to $\boldsymbol{y} = \boldsymbol{\Phi}(z, \boldsymbol{y})$.

In order to do this, note that by the Implicit Function Theorem [FS09, Theorem B.4, Theorem B.6], until $|z| > 0$ is large enough so that the circle contains a zero of the Jacobian of $\boldsymbol{y} - \boldsymbol{\Phi}(z, \boldsymbol{y})$, the solution remains analytic. Thus we shall work with the system

$$\begin{cases} y_1 &= \Phi_1(z; y_1, \ldots, y_m) \\ \vdots &= \vdots \\ y_m &= \Phi_m(z; y_1, \ldots, y_m) \\ 0 &= \det\left(I_{m\times m} - \frac{\partial \boldsymbol{\Phi}}{\partial \boldsymbol{y}}(z; y_1, \ldots, y_m)\right) \end{cases}, \tag{4.1}$$

known as the *characteristic system* in [FS09].

When the underlying dependency graph ($\mathcal{Y}_i \to \mathcal{Y}_j$ if and only if $\frac{\partial}{\partial y_j}\Phi_i \neq 0$) is strongly connected all of the GFs $y_j(z)$ have the same radius of convergence, and, when some other analytic conditions are satisfied, we are guaranteed that the nature of the singularities is the ubiquitous square-root type.

The restriction that the underlying graph be strongly connected is not a constraining one. Indeed, we can compute ρ starting from the sink strongly connected components and move up the DAG (directed acyclic graph) of strongly connected components. If, say, a strongly connected component depends on a variable in another connected component, then there is a simple procedure to decide what happens with the singularities similar to that of [FS09, Theorem IV.8]. Henceforth we assume the dependency graph to be **strongly connected**.

Theorem 4 (Drmota-Lalley-Woods)**.** *Let $\boldsymbol{y} = \boldsymbol{\Phi}(z, \boldsymbol{y})$ be a nonlinear **polynomial** system, that is a-positive, a-irreducible and a-proper. Then all of the component functions $y_j(z)$ have the same radius of convergence $\rho < \infty$, and there exist functions h_j analytic at the origin such that, in a neighbourhood of $z = \rho$ we have*

$$y_j = h_j\left(\sqrt{1 - \tfrac{z}{\rho}}\right).$$

Here *a-positive* means that the coefficients of the polynomials Φ_j are non-negative for such j, *a-irreducible* means that the dependency graph is strongly connected, while *a-proper* means that each Φ_j satisfies a Lipschitz-like condition [FS09, p. 489] for the respective polynomials.

Systems that are produced by considering only sums and multiplications of species (and thus SEQ too) fall into this category. We are interested in the general case of the combinatorial systems that can be built from our primitives SEQ, SET, CYC, $+$, $\times$, and there we have no guarantee that we will still get a square-root singularity. We think that a similar result should still hold for such cases, however, this does tell us, at least, that for quite a general set of specifications such that the root $\rho > 0$ of $\det\left(I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}(z; y_1(z), \ldots, y_m(z))\right)$ that is closest to the origin is a double root.

4.1 Reduction to one-dimension

One might be tempted to exploit the existing function in NewtonGF to compute $\boldsymbol{y}(z)$ for a given z with $0 \le z < \rho$ to reduce the problem to that of finding the root of a single variable function

$$x \mapsto \det\left(I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}(x; y_1(x), \ldots, y_m(x))\right) ,$$

however, methods produced in this way run into troubles as we briefly explain below. To read about a more sound method, the reader is referred to subsection 4.2.

Note that the algorithm implemented in the official version of NewtonGF, which uses a binary search, is based on the same principle: given $x > 0$, we try to evaluate $\boldsymbol{y}(x)$ by a Newton iteration.

We will write x instead of z in this subsection to emphasize the fact that it will be a real number satisfying $0 \le x < \rho$. Given a fixed x close to ρ, we must, at each step, compute $y_1(x), \ldots, y_m(x)$ from $\boldsymbol{y} - \Phi(x; \boldsymbol{y}) = \mathbf{0}$. For $x = \rho$ we have $\det\left(I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}(x; \boldsymbol{y})\right) = 0$, and so the Jacobian in the computation of $\boldsymbol{y}(x)$ by Newton-Raphson is singular. We claim that this slows down the Newton iteration to compute $\boldsymbol{y}(x)$ as $x \to \rho^-$, and second, we claim that even controlling that x remains within the interval $[0, \rho)$ can be a problem.

4.1.1 Cost of the 'inner' Newton iteration for $\mathbf{y}(x)$

Let us first consider the one dimensional case $m = 1$, and let $g(y) = y - \Phi(x; y)$. When y is close to the zero $\theta = y(x)$ a Newton iteration produces

$$y \mapsto y - \frac{g(y)}{g'(y)} = \theta - \frac{1}{2}\frac{g''(\theta)}{g'(\theta)}(y-\theta)^2 + O((y-\theta)^3) . \tag{4.2}$$

Here what we should notice is that the constant $-\frac{g''(\theta)}{2\,g'(\theta)}$ that accompanies the 'square of the previous error' is actually large. Indeed, $g'(\theta)$ gets closer and closer to 0 as x gets closer to ρ because

$$g'(y) = 1 - \frac{\partial}{\partial y}\Phi(x; y) ,$$

is exactly the determinant that must be 0 at $x = \rho, y = y(\rho)$. The second derivative

$$g''(y) = -\frac{\partial^2}{\partial^2 y}\Phi(x; y)$$

is negative and nonzero close to $x = \rho, y = y(\rho)$, as long as $\Phi(x; y)$ is not linear in y.

To explain how small $g'(\theta) = g'(y(x))$ actually is, let us assume for a moment that we had $y(z) = h\left(\sqrt{1 - \frac{z}{\rho}}\right)$ for a function h that is analytic at the origin, as in Theorem 4. After a change of variables and a simple Taylor expansion we obtain

$$\frac{g''(\theta)}{g'(\theta)} \sim \frac{1}{2\,h'(0)\,\sqrt{1 - \frac{x}{\rho}}}\,. \qquad (x \to \rho)$$

We expect this problem to be resolved by keeping track of the values of $\boldsymbol{y}(x)$ for the previous iteration, indeed, because $\boldsymbol{y}(x) \to \boldsymbol{y}(\rho)$ as $x \to \rho^-$ (even if they are not analytic at $z = \rho$, they have continuity from the left). The conjecture here is that, with this idea, and a Newton iteration over x, we may require just a small number of iterations (maybe $\log$ or constant) to compute $\boldsymbol{y}$ at the next point, given the values on the previous one).

4.1.2 Computing ρ while keeping $0 \le x < \rho$.

We have talked about the computation of $\boldsymbol{y}(x)$ provided that $0 \le x < \rho$, it remains to explain here how to compute ρ. The idea now is to use a numerical Newton-Raphson iteration to compute ρ by looking for the root of the determinant $f(x) = \det\left(I_{m\times m} - \frac{\partial \boldsymbol{\Phi}}{\partial \boldsymbol{y}(x)}(x; \boldsymbol{y})\right)$. Here, however, there is no guarantee that we will not, at some point, get to the right of ρ and not be able to compute $\boldsymbol{y}(x)$ anymore. In fact, this may happen repeatedly as the following example shows.

Example 4.1. Consider the binary trees with weight only at the leaves

$$\mathcal{A} = \mathcal{Z} + \mathcal{A} \times \mathcal{A}\,.$$

In this case we have a explicit solution for the OGF

$$A(z) = \frac{1 - \sqrt{1 - 4\,z}}{2}\,.$$

The Jacobian of the characteristic system is

$$\begin{pmatrix} 1 - 2A & 0 \\ 0 & 1 \end{pmatrix}$$

with determinant $1 - 2\,A = \sqrt{1 - 4\,z}$. Notice that this already makes no sense when $z > \frac{1}{4} = \rho$.

A Newton iteration gives

$$x \mapsto \frac{1}{2} - x\,,$$

which is guaranteed to take us from $[0, 1/4)$ to $(1/4, 1/2]$, thus taking us out of the convergence radius. Thus, given $x > 1/4$ the equation $A = x + A^2$ has no solution A and the method fails because we cannot compute our current $\boldsymbol{y}(x) = (A(x))$. $\Diamond$

This problem is not a rare one, indeed, it actually stems from the fact that we have a singularity of the square-root type in a large number of cases, due to Theorem 4.

Let $f(z) = \det\left(I_{m\times m} - \frac{\partial \boldsymbol{\Phi}}{\partial \boldsymbol{y}}(z; \boldsymbol{y}(z))\right)$, and let us write $f(z) = h\left(\sqrt{1 - \frac{z}{\rho}}\right)$ where h is analytic in some neighbourhood of 0, as in the case of Theorem 4 (note that $f(z)$ depends polynomially on $y_1, \ldots, y_m$).

Observe that $h(0) = 0$ by setting $z = \rho$, and **suppose first** that $h'(0) \neq 0$. Then we have

$$z - \frac{f(z)}{f'(z)} = z + \frac{2\,\rho\,h\left(\sqrt{1-\frac{z}{\rho}}\right)}{h'(0)}\,\sqrt{1-\frac{z}{\rho}} + O\left((1-\frac{z}{\rho})^1\right)$$

which, by analyticity $h\left(\sqrt{1-\frac{z}{\rho}}\right) = h'(0)\,\sqrt{1-\frac{z}{\rho}} + O\left((1-\frac{z}{\rho})^1\right)$, implies

$$z - \frac{f(z)}{f'(z)} = 2\rho - z + O\left((1-\frac{z}{\rho})^1\right)\,,$$

exactly as in the case in the example above.

A common solution for this class of problems with a double-root is to consider

$$z \mapsto z - \frac{1}{2}\,\frac{f(z)}{f'(z)}\,,$$

instead of the traditional Newton iteration. Indeed, the same calculation leads us to

$$z - \frac{1}{2}\,\frac{f(z)}{f'(z)} = \rho + O\left((1-\frac{z}{\rho})^1\right)\,,$$

and now we must collect the remainder terms to get

$$z - \frac{1}{2}\,\frac{f(z)}{f'(z)} = \rho - \rho\,\frac{h''(0)}{2\,h'(0)}\,(1-z/\rho)^{3/2} + o\left((1-z/\rho)^{3/2}\right)\,,$$

and so, to stay on the left of ρ for the next iteration, we must show that $\dfrac{h''(0)}{h'(0)} > 0\,.$

Here we do not have a general proof, but in terms of experiments, this idea has not worked out very well, some systems repeatedly yield values that are on the right of ρ.

Example 4.2. Let us consider

$$\mathcal{Y} = \mathcal{Z} + \mathcal{Y} \times \textsc{Seq}_{\geq 1}\left(\mathcal{Z} \times \mathcal{Y}\right)\,.$$

In this case we have $\rho = \sqrt{3-2\,\sqrt{2}} \doteq 0.414213562\ldots$ and $y(\rho) = \frac{2-\sqrt{2}}{2\,\sqrt{3-2\,\sqrt{2}}} \doteq 0.707106781\ldots$ while we can compute

$$z - \frac{1}{2}\,\frac{f(z)}{f'(z)} = \rho + \theta\,(1-z/\rho)^{3/2} + o((1-z/\rho)^{3/2})\,, \tag{4.3}$$

with $\theta \doteq 0.20403568\ldots$, which means that the above method takes us out of the interval $[0,\rho)$ when the point z is close enough to ρ. Of course, equation (4.3) implies that we have

$$z - \frac{f(z)}{f'(z)} = 2\rho - z + 2\theta\,(1-z/\rho)^{3/2} + o((1-z/\rho)^{3/2}) \tag{4.4}$$

for the usual Newton-iteration, exactly as we explained above. $\Diamond$

4.1.3 Hidden implementation details

So far we have not explained some important implementation details about this method. The avid reader may have noticed that, in fact, to compute $f'(z)$, where $f(z) = \det\left(I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}(z;\boldsymbol{y})\right)$, we require $y_1'(z), \dots, y_m'(z)$. Indeed

$$f'(z) = \frac{\partial}{\partial z}\det\left(I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}\right) + \sum_{k=1}^{m} \frac{\partial}{\partial y_k}\det\left(I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}\right) y_k'(z)\,. \tag{4.5}$$

We explain how this issue can be solved. Consider the equation $\boldsymbol{y}(z) = \mathbf{\Phi}(z;\boldsymbol{y}(z))$, and differentiate it to get

$$\boldsymbol{y}' = \frac{\partial \mathbf{\Phi}}{\partial z} + \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}} \cdot \boldsymbol{y}'\,,$$

thus

$$\boldsymbol{y}' = \left(\mathrm{I}_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}\right)^{-1} \frac{\partial \mathbf{\Phi}}{\partial z}\,. \tag{4.6}$$

The matrix $\left(\mathrm{I}_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}\right)^{-1}$ is already computed during the Newton iteration for $\boldsymbol{y}(z)$ in the inner-level. Then, knowing the values $z, y_1(z), \dots, y_m(z)$ and applying automatic-differentiation we get $\frac{\partial \mathbf{\Phi}}{\partial z}$ which then allows us to get $y_1'(z), \dots, y_m'(z)$.

In practice the derivative $f'(z)$ is not computed symbolically, because computing the symbolic expression for a determinant has no known fast solution. However, computing a determinant numerically is much easier, and computing the derivative of a determinant can be done, for instance, by differentiating row-wise and adding the results of the determinants. Thus we compute the numerical values of these determinants that we are to add by using automatic differentiation, plugging in the values of $z, y_1(z), \dots, y_m(z), y_1'(z), \dots, y_m'(z)$, and finally adding we obtain $f'(z)$.

Thus the algorithm is actually not as simple to implement efficiently as it may seemed at first sight.

4.2 A joint-computation method

The above method fails in that it requires us to compute $\boldsymbol{y}(t)$, which is possible when $t < \rho$, but not more in general. A more reasonable way then, is to compute $\boldsymbol{y}$ and ρ simultaneously by a single multi-dimensional Newton-Raphson for

$$f(z,\boldsymbol{y}) = \begin{pmatrix} y_1 - \Phi_1(z;\boldsymbol{y}) \\ \vdots \\ y_m - \Phi_m(z;\boldsymbol{y}) \\ \det\left(I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}(z;\boldsymbol{y})\right) \end{pmatrix}.$$

In this case, the iteration reads

$$(z,\boldsymbol{y}) \mapsto (z,\boldsymbol{y}) - (\mathrm{Jac}_f(z,\boldsymbol{y}))^{-1} f(z,\boldsymbol{y})\,,$$

where

$$\mathrm{Jac}_f(z,\boldsymbol{y}) = \begin{pmatrix} -\frac{\partial \mathbf{\Phi}}{\partial z}(z;\boldsymbol{y}) & I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}(z;\boldsymbol{y}) \\ \frac{\partial}{\partial (z,\boldsymbol{y})}\det\left(I_{m\times m} - \frac{\partial \mathbf{\Phi}}{\partial \boldsymbol{y}}(z;\boldsymbol{y})\right) \end{pmatrix}.$$

In this case the difficulty lies in choosing the initial point $(z_0, \boldsymbol{y}_0)$ in such a way that the Newton iteration will converge to the right fixed-point. This problem is illustrated by Figure 4.1, where we observe that some initial conditions seem to lead us away from the right fixed point.

The advantage of this method, however, lies in the fact that we may start from a point (z_0, y_0) even with $z_0 > \rho$ and still perform the computation. The initial conditions may even be complex numbers, although the computations involved in such a case may be heavier.

In practice, the algorithm I propose here is simply using the algorithm that was previously implemented in the package to a lower precision to get the starting point. Of course, a more detailed analysis, showing that this algorithm does converge to $(\rho, \boldsymbol{y}(\rho))$ when we are start from small enough ball around it, is still pending.

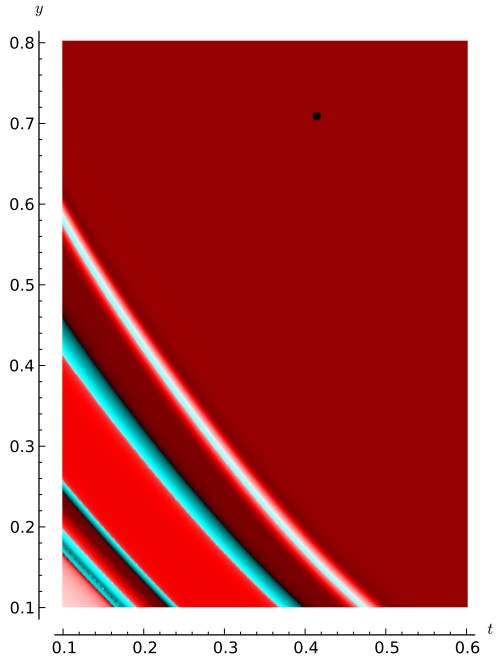

Figure 4.1: Result of applying 3 iterations for $\mathcal{Y} = \mathcal{Z} + \mathcal{Y} \times \textsc{Seq}_{\geq 1}(\mathcal{Z} \times \mathcal{Y})$, starting from (t, y). In this case we have $\rho = \sqrt{3 - 2\sqrt{2}} \doteq 0.414213562\ldots$ and $y(\rho) = \frac{2-\sqrt{2}}{2\sqrt{3-2\sqrt{2}}} \doteq 0.707106781\ldots$ The corresponding final value in $\mathbb{C}$ is indicated as follows: The absolute value of it is indicated by the brightness (with white being infinity and black being zero), while its argument is represented by the hue (red being *positive* real). The black point indicates the solution $(\rho, y(\rho))$.

5 Future work

Here we briefly summarise the points which require further work/investigation

i) Use either the generating functions or the combinatorial expressions for the implementation of the well-foundedness algorithm.

ii) Understand why, when adding direct computations for the small values of N in the functions 'goExp' and 'goRecip', we do not reduce the execution time.

iii) Have a *proven* method to compute the radius of convergence. I was not able to complete this part during my internship, however, it may be possible to show that the method in subsection 4.2

works, provided that we are in some small enough ball of ρ (which should be computable).

Bibliography

[BCO+07] Alin Bostan, Frédéric Chyzak, François Ollivier, Bruno Salvy, Éric Schost, and Alexandre Sedoglavic. Fast computation of power series solutions of systems of differential equations. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *SODA*, pages 1012–1021. SIAM, 2007.

[Bos10] Alin Bostan. Algorithmes rapides pour les polynômes, séries formelles et matrices. Journées Nationales du Calcul Formel 2010, May 2010. Notes d'un cours dispensé aux Journées Nationales du Calcul Formel 2010.

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics.* Cambridge University Press, 2009.

[Joy81] André Joyal. Une théorie combinatoire des séries formelles. *Adv. in Math.*, 42(1):1–82, 1981.

[PSS12] Carine Pivoteau, Bruno Salvy, and Michèle Soria. Algorithms for combinatorial structures: Well-founded systems and Newton iterations. *J. Comb. Theory, Ser. A*, 119(8):1711–1773, 2012.

[Zim91] P. Zimmermann. *Séries génératrices et analyse automatique d'algorithmes*. ANRT [diff.], 1991.