A quantitative study of fork-join processes with non-deterministic choice: application to the statistical exploration of the state-space

Antoine Genitrini, Martin Pépin, Frederic Peschanski

A quantitative study of fork-join processes with non-deterministic choice: application to the statistical exploration of the state-space

Antoine Genitrini[a], Martin Pépin[a], Frederic Peschanski[a]

[a]*Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6 - UMR 7606, F-75005 Paris, France.*
This research was partially supported by the ANR MetACOnc project ANR-15-CE40-0014.

Abstract

We study concurrent programs with non-deterministic choice, loops and a fork-join style of coordination under the lens of combinatorics. As a starting point, we interpret these programs as combinatorial structures. We propose a framework, based on analytic combinatorics, allowing to analyse their quantitative aspects such as the average number of execution path induced by the choice operator, or the proportion of executions of a program with respect to its number of execution prefixes. Building on this theoretical investigation, we develop efficient algorithms aimed at the statistical exploration of their state-space. The first algorithm is a uniform random sampler of bounded executions, providing a good default exploration strategy. The second algorithm is a uniform random sampler of execution prefixes of a given bounded length, offering a more fine-grained generation tool, thus enabling to bias the exploration in a controlled manner. The fundamental characteristics of these algorithms is that they work on the syntax of the programs and do not require the explicit construction of the state-space.

Keywords: Concurrency, Non-determinism, Fork-join processes, Loops, uniform random generation, combinatorics

1. Introduction

Analysing the state-space of concurrent programs is a notoriously difficult task, if only because of the infamous *state explosion* problem. Several techniques have been developed to mitigate this explosion: symbolic encoding of the state-space, partial order reductions, exploiting symmetries, etc. An alternative approach is to adopt a probabilistic point of view, for example by developing statistical analysis techniques such as [1]. The basic idea is to generate random executions from program descriptions, sacrificing exhaustiveness for the sake of tractability. This idea of empowering formal verification with probabilistic tools has been first presented in [2] where the authors introduce the notion of Monte-Carlo model-checking. An important question which is raised in the paper is: how to control the distribution of the sampled objects? There is a crucial difference between generating an *arbitrary* execution and generating a *controlled* random execution according to a known (typically the uniform) distribution. Only the latter allows to estimate the coverage of the state-space of a given analysis. Moreover, the *uniform* distribution plays an important role for that

Email addresses: antoine.genitrini@lip6.fr (Antoine Genitrini), martin.pepin@lip6.fr (Martin Pépin), frederic.peschanski@lip6.fr (Frederic Peschanski)

matter as it *a priori* gives the best coverage in the absence of any further information. Our goal in this paper is to provide such samplers for a class of concurrent programs we specify below.

The starting point of our approach is to find a suitable combinatorial interpretation of the fundamental constructions of concurrency. In previous work, we have separately studied the combinatorial interpretation of *parallelism*, seen as increasing labellings [3], *non-determinism*, seen as partial labellings [4], and *synchronisation* as constrained labellings. In this paper, we integrate these various interpretations, and a new interpretation for loops, into a single *combinatorial specification*, providing a unified framework for studying these programs using the *symbolic method* from [5]. This is a significant leap forward in terms of expressiveness compared to our previous work, because of the presence of loops, but also because we integrate the non-determinism in a non-trivial language with synchronisation. Moreover, at the combinatorial level, this integration is not straightforward and requires to develop a different approach from our previous work, as we explain at the beginning of Section 3.

The main interest of this framework is that it allows to obtain *generating functions* describing the possible executions of a given program, in a systematic way. At the theoretical level, this is often a suitable starting point for the study of quantitative problems in *analytic combinatorics*. For instance, in the present paper, we quantify precisely the number of execution paths induced by the non-deterministic choice operator in the *average case*, thus giving a key witness of the expressiveness of the choice operator. Another example of quantitative results on concurrency obtained via analytic combinatorics is the study of the typical shape of the state-space of programs, though in a simpler model, obtained in [3]. In Section 4, we present a similar result for our model, and at a more precise scale since we describe the state-space of individual programs.

At a more practical level, the framework we develop is also an interesting source of algorithmic investigation, which is the main concern of this article. The first algorithmic problem we study is that of *counting* the number of executions (of bounded length, when in the presence of loops) of the programs. This is the question one has to answer to quantify the so-called *state-explosion* phenomenon, and this is an important building block of our algorithmic toolbox. Unfortunately, counting executions of concurrent programs is hard in the general case. We show in [6] that, even for simple programs only allowing *barrier synchronisation*, counting executions is a #P-complete[1] problem. Fork-join parallelism enables a good balance between tractability and expressiveness by enforcing some structure in the state-space. A second problem is caused by non-determinism because for each non-deterministic choice we have to select a unique branch of execution. Moreover, choices can be nested so that the number of possibilities can grow exponentially. Relying on an efficient encoding of the state-space as generating functions, we manage to count executions without expanding the choices. Of course counting executions has no direct practical application, but it is an essential requirement for us to build two complementary and more interesting analysis techniques. First, we develop a uniform random sampler of executions relying on that counting information. In the loop-free fragment of our model it is uniform over all executions whereas, in the general case, it samples uniform executions of a given (bounded) length. Without prior knowledge of the state-space, the uniform distribution yields the best coverage and thus offers a good default exploration strategy of the state-space. As a second and alternative approach, we provide a uniform random sampler of execution *prefixes*. This algorithm offers an alternative to the uniform generation of

[1] A problem is in #P if it consists in counting the number of accepting paths of a polynomial-time non-deterministic Turing machine. It is #P-complete if, in addition, every other problem in #P reduces to it in polynomial time.

executions giving more flexibility, as it allows one to introduce *bias* in the generation, while still maintaining *control* over the distribution of the sampled objects. A fundamental characteristic of all the algorithms presented in the present paper is that they work on the *syntactic* representation of the programs, thus avoiding the explicit construction of the state-space. This allows to analyse systems with a large state space.

The outline of the paper is as follows. In Section 2, we present a first version of the program class of non-deterministic fork-join programs. We introduce the notion of global choice, which characterises the influence of non-determinism in this class and we provide precise quantitative results on this notion. We then describe a counting algorithm and a uniform random sampler of executions for such programs. In Section 3, we extend the model with loops and offer an alternative approach to random sampling. Finally, in Section 4, we study the execution prefixes of programs with loops, both from a quantitative and an algorithmic point of view. At the end of each section, we support all the proposed algorithms with an experimental evaluation of their performance, thus establishing the tractability of our approach.

Related work

Our study combines viewpoints and techniques from concurrency theory and combinatorics. A similar line of work exists for the so-called "true concurrency" model (by opposition to the interleaving semantics that we use in our study) based on the trace monoid using *heaps combinatorics* (see [7, 8]). To our knowledge these only address the parallelism issue and not non-determinism *per se*. In [9], the authors cover the problem of the uniform random generation of words in a class of synchronised automata. This approach is able to cover a slightly more expressive set of programs but this comes at the cost of the construction of a product (synchronizing) automaton of exponential size in the worst case. Another approach, investigated in the context of Monte-Carlo model-checking, is based on the combinatorics of *lassos*, which relates to the verification of temporal-logic properties over potentially *infinite* executions. In [10], the authors of this method highlight the importance of uniformity. Later [11] gives a uniform random sampler of *lassos*, however relying on the explicit, costly construction of the whole state-space, hence impractical for even small processes. Finally [12] studies the random generation of executions in a model similar to the one we cover by extending the framework of Boltzmann sampling. Although Boltzmann samplers are usually fast, they turn out to be impractical in this context because of the heavy symbolic computations imposed by the interplay between parallelism and synchronisation.

Compared to [4], which discusses non-determinism without synchronisation, we show here that the approach hits a wall when introducing loops and requires a new encoding of the state-space for the non-determinism to be studied in a more expressive language. This paper extends [13] with, in particular, new quantitative results regarding the typical number of global choices and the proportion of executions with respect to execution prefixes.

2. A combinatorial interpretation of non-deterministic fork-join programs

The goal of this section is to introduce the combinatorial tools that will be used throughout the paper and to study a first class of concurrent programs featuring a fork-join programming style with non-determinism. The interest of this class is that it showcases how different features of concurrency such as the interleaving semantics of the pure merge operator [3], series-parallel synchronisation [14] and non-determinism [4] can be integrated and studied in a unified framework.

In Section 3, this class will be extended with a loop construction in order to gain more expressiveness. This extension has several technical implications on the combinatorial framework at use here, which will be discussed.

2.1. Non-deterministic Fork-Join processes (without loops)

Definition 1 (Non-deterministic fork-join programs)**.** *Given a set of symbols $\mathcal{A}$ representing the "atomic actions" of the language, we define the class of non-deterministic fork-join programs (over this set $\mathcal{A}$), denoted NFJ as follows:*

$$\begin{array}{rcll} P, Q & ::= & P \parallel Q & \textit{parallel composition} \\ & | & P; Q & \textit{sequential composition} \\ & | & P + Q & \textit{non-deterministic choice} \\ & | & a \in \mathcal{A} & \textit{atomic action} \end{array}$$

It is important to mention now that in the present paper, we will not specify further what the content of the atomic actions is. We treat them as black boxes and assume all action names within a term to be different. We also consider programs up to injective relabelling, so that $(a \parallel b)$ and $(d \parallel c)$ represent the same program. Our focus is set on the order in which these actions can be fired and scheduled by the different operators of the language. In other words, we study the *control-flow* of concurrent programs as an approximation of their behaviour. In all our examples we use lower-case Roman letters as unique identifiers to distinguish between actions.

We give NFJ an *interleaving* semantics, which means that an execution is seen as a *sequence* of small atomic steps and that the executions of $P \parallel Q$ are all the possible interleavings of an execution of P and an execution of Q. We start by defining a "step" relation of the form $P \xrightarrow{a} P'$ between two programs and an atomic action describing one small computation step. When $P \xrightarrow{a} P'$, we say that "program P reduces to P' by firing a". The inference rules defining the step relation are given in Figure 1.

$$\frac{}{a \xrightarrow{a} 0}\ (\text{act}) \qquad \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}\ (\text{Lpar}) \qquad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}\ (\text{Rpar})$$

$$\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q}\ (\text{seq}) \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}\ (\text{Lchoice}) \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}\ (\text{Rchoice})$$

Figure 1: Semantic of NFJ given as a set of inference rules defining a step relation

As a convenience, we allow the program on the right-hand-side of the step relation to be either a regular NFJ program or a special symbol 0 representing a program that has completed its execution. We also consider the following rewriting rules allowing to use 0 with the parallel and sequential composition operators in the conclusions of rules (Lpar), (Rpar) and (seq):

$$(0; P) = P$$
$$(P \parallel 0) = (0 \parallel P) = P$$

We are now equipped to define the executions of the language as a sequence of steps ending on the empty program 0.

Definition 2 (Execution)**.** *An execution of an NFJ program P_0 is a sequence of steps of the form $P_0 \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \dots \xrightarrow{a_{n-1}} P_{n-1} \xrightarrow{a_n} P_n = 0$ where for all i, the step $P_{i-1} \xrightarrow{a} P_i$ is a* proof-tree*, that is it contains all the applied rules and not simply its conclusion.*

We refer to the set of all possible executions of a program as its state-space*.*

In the absence of ambiguity, which is the case in this section, a proof-tree can be safely identified to its conclusion. As an example, an execution of the program $P = (a; b) \parallel (c+d)$ may fire either a, c or d at the first step. The proof-trees corresponding to these three possible first steps are given below.

$$\dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} 0}\,(\text{act})}{a; b \xrightarrow{a} b}\,(\text{seq})}{(a; b) \parallel (c + d) \xrightarrow{a} b \parallel (c + d)}\,(\text{Lpar}) \qquad \dfrac{\dfrac{\dfrac{}{c \xrightarrow{c} 0}\,(\text{act})}{c + d \xrightarrow{c} 0}\,(\text{Lchoice})}{(a; b) \parallel (c + d) \xrightarrow{c} a; b}\,(\text{Rpar})$$

$$\dfrac{\dfrac{\dfrac{}{d \xrightarrow{d} 0}\,(\text{act})}{c + d \xrightarrow{d} 0}\,(\text{Rchoice})}{(a; b) \parallel (c + d) \xrightarrow{d} a; b}\,(\text{Rpar})$$

Definition 2 gives a natural semantics to NFJ. However it does not highlight the influence of non-determinism in the structure of the state-space. It is interesting to separate the application of the choices from the interleaving semantics of the fork-join core of the language. In order to achieve this, we use the notion of *global choice*. Informally, a global choice of a program P is a program obtained by selecting one of the two alternatives in each sub-term of the form $P_1 + P_2$ and removing the other.

Definition 3 (Global choices)**.** *The set of global choices of P, denoted* choices(P) *is a set of* choice-free *programs obtained from P inductively as follows:*

$$\begin{aligned}
&\text{choices}(a) = \{a\}\\
&\text{choices}(P + Q) = \text{choices}(P) \cup \text{choices}(Q)\\
&\text{choices}(P \parallel Q) = \{(P' \parallel Q') \mid P' \in \text{choices}(P); Q' \in \text{choices}(Q)\}\\
&\text{choices}(P; Q) = \{(P'; Q') \mid P' \in \text{choices}(P); Q' \in \text{choices}(Q)\}
\end{aligned}$$

With this notion at hand, the non-determinism can be untangled from the interleaving semantics since an execution can now be seen as the combination of a global choice and an execution of this global choice. The selection of the global choice carries all the non-determinism coming from the choices of the program whereas the execution of this global choice contains all the expressiveness of the interleaving semantics. Moreover, it is easy to prove that an execution of a choice-free program fires *every* atomic action in the program exactly once. Therefore a global choice containing n atomic actions only has executions of length n and such an execution can be identified to a labelling of its actions with integers of $[\![1; n]\!]$ corresponding to their position in the sequence of steps.

For instance, the program $P = m;(w \parallel ((t + (c; g)); (s + n); p)); e$ models the beverage vending machine pictured in Figure 2. One of its possible executions corresponds to firing the following sequence of actions (in this order) m, t, n, w, p, e and the global choice corresponding to this execution is $P' = m;(w \parallel (t; n; p)); e$. Figure 2 gives a graphical representation of this execution as a labelling of P'. In the rest of this section, we will rely on this second point of view on executions to reason about them. To this end, we now introduce the combinatorial tools that we will need to model both the class of NFJ programs as a whole, and the set of executions of a given program, as combinatorial objects. This will enable us to analyse them using tools from analytic combinatorics.

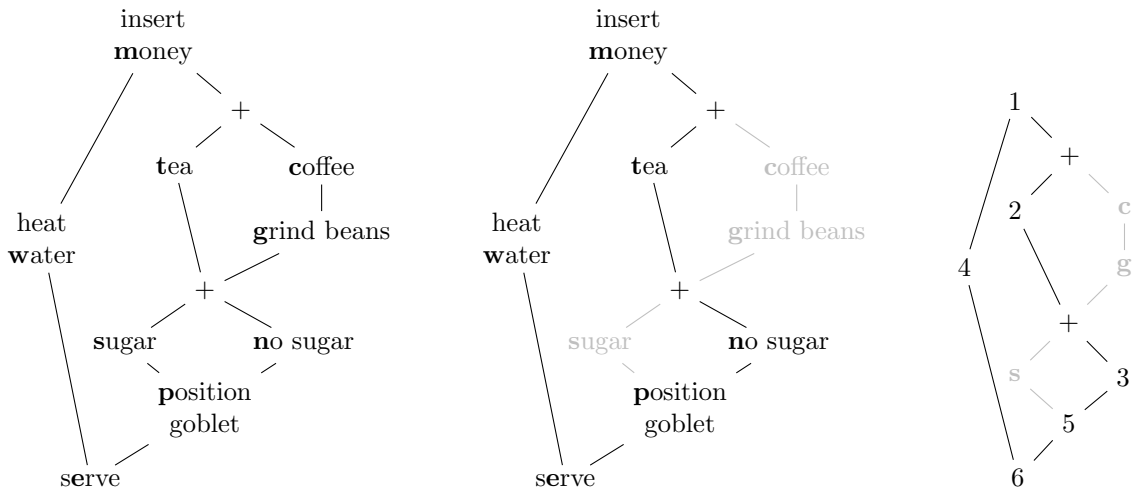


Figure 2: A simple beverage vending machine modelled by $P = m;(w \parallel ((t + (c; g)); (s + n); p)); e$, one of its 4 global choices and one of the 4 possible labellings of this global choice, representing the firing sequence m, t, n, w, p, e. This program has 18 executions in total

2.2. The combinatorial toolset part 1: modelling programs

We introduce here the notions of combinatorial class, combinatorial specification, and generating function. This is a brief introduction as we limit ourselves to the tools that are necessary to tackle the problems covered in the present article. An in depth presentation of the techniques used here can be found in [5].

Definition 4 (Combinatorial class)**.** *A combinatorial class $\mathcal{C}$ is a set of objects equipped with a size function $|\cdot| : \mathcal{C} \to \mathbb{N}$ such that for all $n \in \mathbb{N}$ the set $\mathcal{C}_n = \{c \in \mathcal{C} \mid |c| = n\}$ of objects of size n of $\mathcal{C}$ is finite.*

For example, one combinatorial class of interest for us is the class of all NFJ programs where the size $|P|$ of a program P is defined as the number of atomic actions it contains, that is:

$$|(P \parallel Q)| = |(P + Q)| = |(P; Q)| = |P| + |Q|$$
$$|a| = 1$$

Table 1 gives the list of all NFJ programs of size at most 3 and their number.

Table 1: All NFJ programs of size at most 3 and their number

size n	all programs of size n	#
1	a	1
2	$(a+b)$, $(a \parallel b)$, $(a;b)$	3
3	$(a+(b+c))$, $(a+(b \parallel c))$, $(a+(b;c))$, $((a+b)+c)$, $((a \parallel b)+c)$, $((a;b)+c)$ $(a \parallel (b+c))$, $(a \parallel (b \parallel c))$, $(a \parallel (b;c))$, $((a+b) \parallel c)$, $((a \parallel b) \parallel c)$, $((a;b) \parallel c)$ $(a;(b+c))$, $(a;(b \parallel c))$, $(a;(b;c))$, $((a+b);c)$, $((a \parallel b);c)$, $((a;b);c)$	18

Since the number of elements of a given size of a combinatorial class $\mathcal{C}$ is finite, it is possible to define its generating function $C(z)$ as the formal power series $C(z) = \sum_{n\geq 0} c_n z^n$ where c_n is the cardinality of $\mathcal{C}_n$, that is the number of objects of size n in $\mathcal{C}$. The reason behind considering a generating function rather than working on the sequence c_n directly is twofold. First, generating functions behave nicely with respect to high-level operations on combinatorial classes, like the Cartesian product or the disjoint union. This often allows to obtain recurrence relations on the sequences — and even sometimes an explicit formula — in a systematic and elegant way with few manual computations. Second, this enables the use of analytic techniques when this function converges, allowing to obtain information on the asymptotic behaviour of the sequences via complex analysis. This approach has been used successfully in a variety of contexts and has been popularised by the book [5].

As we just mentioned, generating functions behave nicely with respect to some high-level operations on combinatorial classes. For instance we define the disjoint union $\mathcal{C}$ of two classes $\mathcal{A}$ and $\mathcal{B}$, denoted by $\mathcal{C} = \mathcal{A} + \mathcal{B}$, as the combinatorial class whose underlying set is the disjoint union of the elements of $\mathcal{A}$ and the elements of $\mathcal{B}$ and whose size function $|\cdot|_{\mathcal{C}}$ is such that

$$|c|_{\mathcal{C}} = \begin{cases} |c|_{\mathcal{A}} & \text{if } c \in \mathcal{A} \\ |c|_{\mathcal{B}} & \text{if } c \in \mathcal{B} \end{cases} \quad \text{where} \quad \begin{array}{l} |\cdot|_{\mathcal{A}} \text{ is the size function of } \mathcal{A}; \\ |\cdot|_{\mathcal{B}} \text{ is the size function of } \mathcal{B}. \end{array}$$

It is easy to check that the generating function of $\mathcal{C} = \mathcal{A}+\mathcal{B}$ satisfies the formula $C(z) = A(z)+B(z)$. Using similar notations, we can also define the Cartesian product $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ of two combinatorial classes $\mathcal{A}$ and $\mathcal{B}$ where the size of an element (a, b) of $\mathcal{C}$ is $|(a, b)|_{\mathcal{C}} = |a|_{\mathcal{A}} + |b|_{\mathcal{B}}$. Hence, the set $\mathcal{C}_n$ of objects of $\mathcal{C}$ of size n is $\{(a, b) \mid a \in \mathcal{A}_i, b \in \mathcal{B}_j, i + j = n\}$. As a consequence the cardinality c_n of $\mathcal{C}_n$ is $\sum_{i+j=n} a_i b_j$ where a_i is the cardinality of $\mathcal{A}_i$ and b_j is the cardinality of $\mathcal{B}_j$. Thus, we obtain a simple expression for the generating function of $\mathcal{C}$, that is $C(z) = A(z) \cdot B(z)$.

It follows that if one is able to describe a combinatorial class using only these constructions (and possibly recursion), then it is straightforward to obtain an equation satisfied by the generating function of the class. Such a description is called a *combinatorial specification*. The process of finding a specification for the system under study and then applying automatic rules to derive its generating function is called the *symbolic method*. Table 2 gives a few of the constructions we will need in this section and their translations in terms of generating functions. The neutral class $\mathcal{E}$ is not useful for this section but will be later in the article.

As an example of application, it is not too difficult to obtain a specification of the class $\mathcal{F}$ of NFJ programs:

$$\mathcal{F} = \mathcal{Z} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F} + \mathcal{F} \times \mathcal{F}. \tag{1}$$

This specification makes use of the disjoint union, the Cartesian product and a new construction $\mathcal{Z}$

Table 2: Some constructions of the symbolic method

	$\mathcal{C}$	$C(z)$
Neutral class: one element of size 0	$\mathcal{E}$	1
Atomic class: one element of size 1	$\mathcal{Z}$	z
Disjoint union	$\mathcal{A}+\mathcal{B}$	$A(z)+B(z)$
Cartesian product	$\mathcal{A}\times\mathcal{B}$	$A(z)B(z)$

called the *atomic* class. This is the class containing only one element of size one. Here it represents the atomic actions of the language, indeed the program made of only one action has size one and there is only one such program (up to relabelling). The other three terms represent the three other constructions of the language. A program that is not reduced to a single action is either a parallel composition, a sequential composition or a choice between two programs. Moreover these three sets are disjoint, hence the disjoint union. Also, note that the equal sign here denotes an isomorphism rather than an equality, in the sense that there is a bijection between the two sides of the equality that preserves the size. The equal sign, when used in a specification, will always denote an isomorphism rather than a strict equality.

From equation (1) and using the transformation rules of the symbolic method recalled in Table 2, we obtain that the generating function f of NFJ programs satisfies $f(z) = z+3f(z)^2$. This equation can be solved explicitly so that we have a closed formula for f:

$$f(z) = \frac{1-\sqrt{1-12z}}{6}. \tag{2}$$

This leads us to our first result on the number of NFJ programs.

Theorem 1 (Number of NFJ programs)**.** *For $n>0$, the number f_n of NFJ programs containing exactly n atomic actions is given by $f_n = \frac{3^n}{12n-6}\binom{2n}{n}$. Moreover we have $f_n \underset{n\to\infty}{\sim} \frac{12^{n-1}}{\sqrt{n^3\pi}}$.*

Proof. Recall that the number f_n of programs of size n is the coefficient of degree n in the generating function $f(z)$, which we denote by $[z^n]f(z)$. Moreover we have that

$$\sqrt{1-u} = -\sum_{n\geq 0}\frac{4^{-n}}{2n-1}\binom{2n}{n}u^n$$

$$\text{hence}\quad 1-\sqrt{1-12z} = \sum_{n\geq 1}\frac{3^n}{2n-1}\binom{2n}{n}z^n$$

$$\text{and therefore}\quad f_n = \frac{3^n}{12n-6}\binom{2n}{n}.$$

The equivalent for f_n is then obtained by applying Stirling's formula. □

The number of NFJ programs is not interesting as such, but we give it here for two reasons. First it gives an example of application of the symbolic method which demonstrates how quickly one can get to a counting formula, and an asymptotic estimation, using this approach. And second, we need this result as an auxiliary result to prove a more insightful quantitative theorem on the class of NFJ programs: an NFJ program has, in average, relatively few global choices.

The natural intuition is that, since a global choice can be any combination of local choices, the typical number of global choices of a program should grow exponentially with the size of the program. This intuition is correct, but what is less natural is that the growth rate of the number of global choices is quite small. In Theorem 2 we prove that the typical number of global choices, that is their average number over all programs of size n, grows as fast as about 1.11438^n.

Theorem 2 (Average number of global choices)**.** *The average number of global choices of an NFJ program of size n has an equivalent of the form $A \cdot B^n$ when $n \to \infty$ where $A \approx 6.89446$ and $B = \frac{49}{27+12\sqrt{2}} \approx 1.11438$.*

The practical implication of this somewhat surprising result is that, for an average program of reasonably small size, it might be possible to enumerate *all* global choices because their number is such a "small" exponential. As an example, the average number of global choices for programs of size $n = 100$ is approximatively 348 261.

Proof of Theorem 2. The idea to prove Theorem 2 is to count programs annotated with one of their global choices. In other words, we consider the combinatorial class $\mathcal{G}$ of all the pairs of the form (P, P') where $P' \in \text{choices}(P)$ and where the size function of $\mathcal{G}$ is defined by $|(P, P')| = |P|$. The number of objects of size n in $\mathcal{G}$ is therefore the sum, over all programs of size n, of their number of global choices. In order to get the *average* number of global choices given in the statement of the theorem, it will suffice to divide this quantity by the number of programs of size n obtained in Theorem 1.

The class $\mathcal{G}$ can be specified as follows:

$$\mathcal{G} = \mathcal{Z} + \mathcal{G} \times \mathcal{G} + \mathcal{G} \times \mathcal{G} + (\mathcal{G} \times \mathcal{F} + \mathcal{F} \times \mathcal{G}). \tag{3}$$

And the combinatorial interpretation of the above formula is the following:

- A single action has only one global choice, hence there is only one annotated program (P, P') in $\mathcal{G}$ where $P = a$. This is specified by $\mathcal{Z}$.
- A global choice of $P \parallel Q$ is by definition the parallel composition of a global choice of P and a global choice of Q. Hence, the set of annotated programs where the outermost constructor is $\parallel$ is isomorphic to a Cartesian product of $\mathcal{G}$ with itself.
- The same reasoning applies to the sequential composition.
- Finally, the most interesting case is that of the choice construction. A global choice of a program of the form $P + Q$ is either a global choice of P or a global choice of Q. Hence, the set of pairs of the form $(P + Q, R)$ in $\mathcal{G}$ are such that $(P, R) \in \mathcal{G}$ (or $(Q, R) \in \mathcal{G}$) and the second program Q (or P) is a regular, non-annotated, NFJ program. Therefore the sub-class of such terms is isomorphic to $\mathcal{G} \times \mathcal{F} + \mathcal{F} \times \mathcal{G}$.

From (3) we obtain that the generating function $g(z)$ of annotated programs satisfies the following equation where f is the generating function of regular (non-annotated) NFJ program:

$$g(z) = z + 2g(z)^2 + 2g(z)f(z). \tag{4}$$

Again, this equation can be solved explicitly which yields $g(z) = \frac{1-2f(z)-\sqrt{\Delta(z)}}{4}$ where $\Delta(z) = (1-2f(z))^2 - 8z$ expands to $\frac{1}{9}\left(5 - 84z + 4\sqrt{1-12z}\right)$. Obtaining an explicit formula for the number g_n of annotated programs would require to extract the coefficient of degree n in the power series expansion of $g(z)$, which would be extremely tedious.

Instead we resort to singularity analysis. The function Δ is well-defined and decreasing in the interval $[0; \frac{1}{12}]$ and $\Delta(\frac{1}{12}) = -\frac{2}{9} < 0$ so there exists a unique ρ_g in this interval such that $\Delta(\rho_g) = 0$. Let $u = \sqrt{1-12\rho_g}$, we have that $9\Delta(\rho_g) = 5 - 7(1-u^2) + 4u = 0$, which we can solve explicitly and yields $u = \frac{3\sqrt{2}-2}{7}$ and therefore $12\rho_g = \frac{27+12\sqrt{2}}{49} < 1$. This allows to write:

$$g(z) = \frac{1-2f(z)}{4} - h(z)\sqrt{1-\frac{z}{\rho_g}}$$

$$\text{where} \quad h(z) = \sqrt{\Delta(z)\left(1-\frac{z}{\rho_g}\right)^{-1}} \quad \text{is analytic in } \left\{z \in \mathbb{C} \mid |z| < \frac{1}{12}\right\}$$

Therefore the transfer theorem from [5, Thm. VI.3 p. 390] applies to $g(z)$ and its n-th coefficient g_n satisfies $g_n \sim \frac{h(\rho_g)}{2\sqrt{\pi}} n^{-\frac{3}{2}} \rho_g^{-n}$. Furthermore, we have that $h(\rho_g) = \sqrt{-\rho_g \Delta'(\rho_g)}$ which can be computed explicitly. This allows us to conclude the proof since the average number of global choices over programs of size n is $g_n/f_n \sim 6h(\rho_g) \cdot (12\rho_g)^n$. □

2.2.1. The impact of symmetries

Remark: this sub-section questions and refines the result form Theorem 2. It is rather technical and independent from the rest of the paper. It can thus safely be skipped at the first reading.

A natural question to ask, regarding Theorem 2 and its proof, is "how would commutativity and associativity affect this result?". We can indeed see in Table 1 that even for small sizes, many programs can be obtain from one another by flipping the operands of the parallel or the choice operator for instance, and it is clear from the semantics of NFJ that doing so does not change the state-space of programs. It is actually possible to answer this question using similar tools as before, although the proofs become significantly more technical. In this sub-section we state the analogue of Theorem 2 in a refined model which takes the aforementioned symmetries into account and we give the key ideas to prove it. The full details of the proof are given in Appendix A.

We now consider all three constructions of the language to be associative so that for instance $(a \parallel (b \parallel c)) = ((a \parallel b) \parallel c)$, which we will now write $(a \parallel b \parallel c)$. Moreover we consider the parallel composition operator and the choice operator to be commutative so that $(a \parallel (b; c)) = ((b; c) \parallel a)$ for instance. This has the consequence that there are less NFJ programs of given size in this model than in the simpler one, and that programs with many symmetries will be given less importance when computing the *average* number of global choices. We establish the following result.

Theorem 3. *The average number of global choices for programs with n atomic actions, taken up to commutativity and associativity is equivalent to $A \cdot B^n$ when $n \to \infty$ where $B \approx 1.11275$.*

It is extremely common in analytic combinatorics that going from *ordered* to *unordered* collections (in our context: from non-commutativity to commutativity) does not change the form the result but only the constants appearing in it. So the fact that the average number of global choices in this new model has the same behaviour is not surprising. However it is interesting that the new

growth rate of the number of global choices is this close to one we obtained in Theorem 2, and that it is slightly smaller. It is also possible to approximate the constant A but is extremely tedious and of little interest.

Taking these symmetries into account in the counting process requires to write new specifications for the class $\mathcal{F}$ of NFJ programs and for the class $\mathcal{G}$ of annotated NFJ programs, that is the set of pairs (P, P') where $P \in \mathcal{F}$ and P' is a global choice of P. Associativity is expressed using n-ary rather than binary operators whereas commutativity requires to introduce two operators that we have not seen so far, the multi-set operator and an unusual "replication" operator.

Specification of the set of program. Let $\mathcal{F}$ denote the class of NFJ programs, taken up to associativity and commutativity, and let $\mathcal{F}_;$, $\mathcal{F}_\|$ and $\mathcal{F}_+$ denote the sub-classes of $\mathcal{F}$ containing programs whose outermost constructors are respectively a sequence, a parallel composition and a choice.

The class $\mathcal{F}_;$ for instance contains all programs of the form $(P_1; P_2)$ for any two programs P_1 and P_2. But there is an ambiguity issue with this representation when one of P_1 or P_2 is itself of the form $(P_3; P_4)$, because there are at least two possible ways to represent the program. To circumvent this issue, we "flatten" nested sequence operators and say that an element of $\mathcal{F}_;$ is of the form $P_1; P_2; \ldots; P_k$ where $k \geq 2$ and for all $1 \leq i \leq k$ we have $P_i \notin \mathcal{F}_;$. This yields the specification $\mathcal{F}_; = \textsc{Seq}_{\geq 2}(\mathcal{F} \setminus \mathcal{F}_;)$ or, equivalently, $\mathcal{F}_; = \textsc{Seq}_{\geq 2}(\mathcal{Z} + \mathcal{F}_\| + \mathcal{F}_+)$, where $\textsc{Seq}_{\geq 2}(\cdot)$ denotes a sequence of at least 2 elements and is formally defined below.

Definition 5. *Given a combinatorial class $\mathcal{A}$ with no element of size 0, we define the class of sequences of elements of $\mathcal{A}$ as:*

$$\textsc{Seq}(\mathcal{A}) = \bigcup_{j \geq 0} \mathcal{A} \times \mathcal{A} \times \cdots \times \mathcal{A} \quad \textit{(j times)}$$

For $k \geq 0$, we denote by $\textsc{Seq}_{\geq k}(\mathcal{A})$, the restriction of the above union to $j \geq k$.

Proposition 1. *Let $\mathcal{A}$ be a combinatorial class with no element of size 0 and let $A(z)$ denote its generating function. The generating function of $\textsc{Seq}_{\geq k}(\mathcal{A})$ is given by*

$$\frac{A(z)^k}{1 - A(z)}$$

For $\mathcal{F}_+$ and $\mathcal{F}_\|$ we handle associativity the same way, but this does not solve the commutativity issue. The idea to express commutativity is to see the operands of a choice (resp. parallel composition) as a *multi-set* of programs rather than a sequence since the order in which they are written does not matter. Note that we do need a multi-set and not a set since two programs that are equal (up to renaming of the actions) may be used as two branches of a choice or may be composed in parallel. In combinatorics multi-sets are specified using the $\textsc{MSet}(\cdot)$ operator, defined below, which allows to write $\mathcal{F}_+ = \textsc{MSet}_{\geq 2}(\mathcal{F} \setminus \mathcal{F}_+)$ for instance. The treatment of this kind of operators relies on what is known as *Pólya theory* and is covered by the book [15].

Definition 6. *Given a combinatorial class $\mathcal{A}$ with no element of size 0, we define the class of multi-sets of elements of $\mathcal{A}$, denoted by $\textsc{MSet}(\mathcal{A})$, as the quotient of $\textsc{Seq}(\mathcal{A})$ by the following equivalence relation:*

$$(a_1, a_2, \ldots, a_k) \sim (a'_1, a'_2, \ldots, a'_\ell) \Leftrightarrow (k = \ell) \wedge \exists \sigma \in \mathfrak{S}_k, \forall\ 1 \leq i \leq k, a_{\sigma(i)} = a'_i$$

where $\mathfrak{S}_k$ *denotes the set of permutations of* $[\![1;k]\!]$.

The size of an element of $\textsc{MSet}(\mathcal{A})$ *is defined as the sum of the sizes of its components (this is independent of the ordering). Moreover, for* $k \geq 0$*, the class* $\textsc{MSet}_{\geq k}(\mathcal{A})$ *is defined as the sub-class of* $\textsc{MSet}(\mathcal{A})$ *whose elements have at least* k *components, that is* $\textsc{MSet}_{\geq k}(\mathcal{A}) = \textsc{Seq}_{\geq k}(\mathcal{A})/\sim$.

Proposition 2. *Let* $\mathcal{A}$ *be a combinatorial class with no element of size* 0 *and let* $A(z)$ *denote its generating function. The generating function of the multi-set* $\textsc{MSet}(\mathcal{A})$ *is given by*

$$\exp\left(\sum_{j\geq 1}\frac{A(z^j)}{j}\right)$$

Putting together the two constructions introduced above, we get a specification for NFJ expressing associativity and commutativity.

$$\begin{aligned}\mathcal{F} &= \mathcal{Z} + \mathcal{F}_; + \mathcal{F}_\| + \mathcal{F}_+\\ \mathcal{F}_; &= \textsc{Seq}_{\geq 2}(\mathcal{F}\setminus\mathcal{F}_;)\\ \mathcal{F}_\| &= \textsc{MSet}_{\geq 2}(\mathcal{F}\setminus\mathcal{F}_\|)\\ \mathcal{F}_+ &= \textsc{MSet}_{\geq 2}(\mathcal{F}\setminus\mathcal{F}_+)\end{aligned} \tag{5}$$

From this specification, one can derive a system of equations on the generating functions of these four classes and the singularity analysis of these functions yields the following theorem. Its proof is detailed in Appendix A.1.

Theorem 4. *The asymptotic number* f_n *of NFJ programs, up to associativity and commutativity, satisfies*

$$f_n \underset{n\to\infty}{=} \gamma n^{-\frac{3}{2}}\rho^{-n}(1+O(n^{-1}))$$

where $\rho \approx 0.13793576712500258$ *and* $\gamma \approx 0.12607642812680533$.

The value of $\rho^{-1} \approx 7.25$ here has to be compared with the growth rate 12 which we obtained in Theorem 1. A heuristic argument which partly explains this value is that each *syntactic* program of size n is made of $n-1$ binary operators and that about $\frac{2}{3}$ of them are choices or parallel composition. Moreover, it is unlikely that the two operands of a binary operator are isomorphic so each of these $\frac{2}{3}(n-1)$ commutative operators induce a symmetry. In total this accounts for about $2^{\frac{2}{3}n}$ symmetries. In turns out that $12/2^{\frac{2}{3}} \approx 7.56$ which is somewhat close to ρ^{-1}, the rest of the difference correspond to the associativity and the rough approximations of this argument.

Specification of the set of annotated program. Like in the proof of Theorem 2, in order to count the average number of global choices of programs of size n, one first counts the number of pairs (P, P') where P is a program of size n and P' is one of its global choices. We call such a pair an *annotated* program and we define its size as the size of its first component P.

Let $\mathcal{G}$ denote the class of annotated programs of size n and let $\mathcal{G}_;$, $\mathcal{G}_+$ and $\mathcal{G}_\|$ denote the classes of annotated programs whose outermost operator are respectively a sequential composition, a choice or a parallel composition. The equations defining $\mathcal{G}$, $\mathcal{G}_;$ and $\mathcal{G}_\|$ are straightforward to obtain as they follow closely their non-annotated counterparts $\mathcal{F}$, $\mathcal{F}_;$ and $\mathcal{F}_\|$:

$$\begin{aligned}\mathcal{G} &= \mathcal{Z} + \mathcal{G}_; + \mathcal{G}_\| + \mathcal{G}_+\\ \mathcal{G}_; &= \textsc{Seq}_{\geq 2}(\mathcal{G}\setminus\mathcal{G}_;)\\ \mathcal{G}_\| &= \textsc{MSet}_{\geq 2}(\mathcal{G}\setminus\mathcal{G}_\|)\end{aligned} \tag{6}$$

The case of the choice however, requires more work as we need to express the fact that one of the branches of the choice is executed and the others are not. In order to specify this, we reason on the executed branch of the choice and on the sub-set of the other branches which are isomorphic to it. The executed branch (P_0, P_0') belongs to $(\mathcal{G} \setminus \mathcal{G}_+)$ for the same reason the branches of regular (non-annotated) choice programs $\mathcal{F}_+$ belonged to $(\mathcal{F} \setminus \mathcal{F}_+)$ in the previous section. Moreover, among the other branches, $k \geq 0$ branches $P_1, P_2, \ldots, P_k$ may be isomorphic to P_0, that is to say that all of the P_i are copies of P_0 with different atom names. Although the choice operator is associative in this section, we have to specify that the program $P = P_0 + P_1 + P_2 + \cdots + P_k$ has $(k+1)$ distinct choices (one of each P_i) according to our specification. One way to achieve this at the specification level, is to artificially *partition* the branches of P into two sets. We fix an arbitrary ordering of the P_i and we distinguish between the P_i that are "before" P_0 and those that are "after" P_0. Thus, an annotated choice program in $\mathcal{G}_+$ is composed of an annotated branch $(P_0, P_0') \in \mathcal{G} \setminus \mathcal{G}_+$, a possibly empty set of copies of P_0 considered to be before P_0, a possibly empty set of copies of P_0 considered to be after P_0 and a possibly empty multi-set of other branches (different from P_0). As a consequence we have:

$$\mathcal{G}_+ = \bigcup_{(P_0,P_0')\in\mathcal{G}\setminus\mathcal{G}_+} \textsc{MSet}(\{P_0\}) \times \{(P_0, P_0')\} \times \textsc{MSet}(\{P_0\}) \times \textsc{MSet}(\mathcal{F} \setminus \mathcal{F}_+ \setminus \{P_0\}) \setminus \{(P_0, P_0')\}\,.$$

Note that the removal of the term $\{(P_0, P_0')\}$ on the right captures the fact that a choice term cannot be reduced to one single branch. Also note that the term $\textsc{MSet}(\{P_0\}) \times \textsc{MSet}(\mathcal{F} \setminus \mathcal{F}_+ \setminus \{P_0\})$ can be simplified to $\textsc{MSet}(\mathcal{F} \setminus \mathcal{F}_+)$. This can be interpreted by saying that the copies of P_0 considered to be after P_0 can be grouped together with the multi-set of other branches (different from P_0) so that it forms one single multi-set of non-annotated programs. Furthermore, the remaining terms can be simplified too by observing that the j remaining copies of P_0 can be grouped together with (P_0, P_0') and that the $(j+1)$-tuples of the form $(P_0, P_0, \ldots, P_0, (P_0, P_0')) \in \mathcal{F}^j \times \mathcal{G}$ have the same size as the $(j+1)$-tuples of the form $((P_0, P_0'), (P_0, P_0'), \ldots, (P_0, P_0')) \in \mathcal{G}^{j+1}$ and are trivially in bijection with them. We introduce a "replication" operator which can specify this kind of tuples.

Definition 7. *Let $\mathcal{A}$ be a combinatorial class with no element of size 0, we define the class of replicas of elements of $\mathcal{A}$ as*

$$\textsc{Repl}(\mathcal{A}) = \bigcup_{j\geq 1} \left\{ \underset{j \text{ components}}{(x, x, \ldots, x)} \mid x \in \mathcal{A} \right\} \tag{7}$$

Proposition 3. *Let $\mathcal{A}$ be a combinatorial class with no element of size 0 and let $A(z)$ denote the generating function of $\mathcal{A}$. The generating function of $\textsc{Repl}(\mathcal{A})$ is given by*

$$\sum_{j\geq 1} A(z^j)$$

Finally, we can also note that $\textsc{MSet}(\mathcal{F} \setminus \mathcal{F}_+) = \mathcal{F}_+ + (\mathcal{F} \setminus \mathcal{F}_+) + \mathcal{E}$ from (5). Hence we get the simpler specification of $\mathcal{G}_+$ given by:

$$\mathcal{G}_+ = \textsc{Repl}(\mathcal{G} \setminus \mathcal{G}_+) \times (\mathcal{F} + \mathcal{E}) \setminus (\mathcal{G} \setminus \mathcal{G}) \tag{8}$$

Here again, the specifications (6) and (8) translate into a system of equations on the generating functions of $\mathcal{G}$, $\mathcal{G}_+$, $\mathcal{G}_;$, $\mathcal{G}_\parallel$ and $\mathcal{F}$. The analysis of theses functions lead to Theorem 3 and is detailed in Appendix A.3.

This concludes the study of the class NFJ itself and of its number of global choices. The outcome of this sub-section is that we can still obtain an equivalent the average number of global choices in programs of size n when programs are considered up to commutativity and associativity. Unsurprisingly, this equivalent is of the same form as in the simple case without taking the symmetries into account, though the constants are different. Although it is more satisfactory to take these symmetries into account for quantifying the average number of global choices, this result comes at the expense of a considerably more technical analysis. We now turn to the counting and random sampling of executions until the end of Section 2.

2.3. The combinatorial toolset part 2: executions as partial increasing labellings

In order to study the set of executions of a program, and in particular in order to count it, we need to give it a combinatorial interpretation too. As mentioned earlier, the idea is to see an execution as a labelling of the actions of the programs. Before formalising this approach, we present a graphical representation of NFJ programs that will help us picture these labellings. This representation yields graphs similar to those of Figure 2, though with a little more detail so as to remain generic.

Definition 8 (Control graph)**.** *To every NFJ program we associate a control graph with three kinds of nodes: actions* (a)*, fork-join nodes* ($\circ$) *and choices nodes* ($+$)*. The control graph* $G(P)$ *of a program* P *is inductively defined as follows:*

$$G(a) = \quad a$$

$$G(P \parallel Q) = \begin{array}{ccc} & \circ & \\ G(P) & & G(Q) \\ & \circ & \end{array}$$

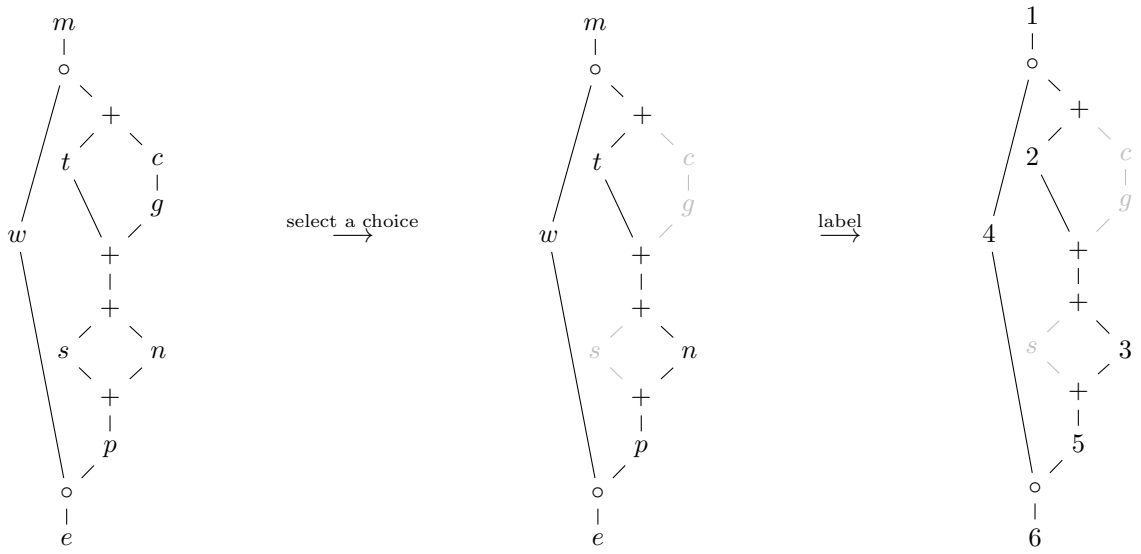
$$G(P;Q) = \begin{array}{c} G(P) \\ | \\ G(Q) \end{array}$$

$$G(P+Q) = \begin{array}{ccc} & + & \\ G(P) & & G(Q) \\ & + & \end{array}$$

As mentioned above, we can see an execution of an NFJ program as a two-step process. First, select a *global choice*, that is select which branch of each choice should be run, and second, label the actions of this global choice using the integers from $[\![1; n]\!]$ according to the order in which they are fired. The integer n is the number of actions in the global choice here. Figure 3 pictures the same example as in Figure 2 using this representation. In the middle picture, one branch of each $+$ node has been selected and the other has been discarded (coloured in light grey). In the rightmost picture, the remaining actions of the graph have been labelled such that, whenever there is an edge between two actions, the action on the upper end of the edge has a smaller label than the action at the bottom. We call this an increasing labelling of the graph since every path from the top of the graph to the bottom is increasingly labelled.

The key idea to study the executions of NFJ programs is to see the set of possible executions of a single program as its own combinatorial class. To this end we will need a *labelled* variant of the formalism presented above. Informally, a labelled combinatorial class is a special case of combinatorial class in which the size n of an object corresponds to a number of "atoms" in the object (typically graph nodes or tree nodes) and where each of these atoms are assigned a unique label from the set $[\![1; n]\!]$. A typical example of labelled class is the set of permutations which can

Figure 3: Two-step decomposition of an execution of program $P = m; (w \parallel ((t + (c; g)); (s + n); p)); e$



be seen as a linear arrangement of n atoms labelled from 1 to n. In our case, the labelled class of interest is that of the executions of a program, which are seen as a labelled global choice.

More formally, labelled combinatorial classes can be seen as sets of pairs of a regular "unlabelled" object and a permutation representing a labelling.

Definition 9 (Labelled combinatorial class)**.** *Let $\mathfrak{S} = \bigcup_{n\geq 0} \mathfrak{S}_n$ denote the set of all finite permutations. A combinatorial class $\mathcal{C}$ is a* labelled *combinatorial class if it is of the form $\mathcal{C} \subseteq C \times \mathfrak{S}$ for some set C and if for all element (c, σ) of $\mathcal{C}$ the size $|(c, \sigma)|$ of the element is the unique n such that $\sigma \in \mathfrak{S}_n$.*

As for unlabelled classes, the study of such a class can be made more systematic when one is able to *specify* it. There exists several operators of the symbolic method which are specific to labelled classes, in addition to those presented in Table 2. Here we will only need two, the *labelled product* and the *ordered product*. The labelled product (denoted by $\star$) of two objects is defined as the set of all possible interleavings of their respective labellings. Formally, we first defined the labelled product of two objects:

$$(a, \sigma_a) \star (b, \sigma_b) = \{((a, b), \sigma) \mid \sigma \text{ is an interleaving of } \sigma_a \text{ and } \sigma_b\}$$

$$\text{that is} \quad \begin{cases} \sigma \in \mathfrak{S}_{|a|+|b|} \\ \forall i, j \leq |a|, \sigma(i) < \sigma(j) \Leftrightarrow \sigma_a(i) < \sigma_a(j) \\ \forall i, j > |a|, \sigma(i) < \sigma(j) \Leftrightarrow \sigma_b(i) < \sigma_b(j) \end{cases}$$

Said differently, the restriction of σ to $[\![1; |a|]\!]$ is a labelling of a using a subset of the labels $[\![1; |a|+|b|]\!]$. Similarly, the restriction of σ to $[\![1 + |a|; |a| + |b|]\!]$ is the labelling of b using the remaining labels. The labelled product of two classes is then defined as a union over all possible pairs:

$$\mathcal{A} \star \mathcal{B} = \bigcup_{a\in\mathcal{A}, b\in\mathcal{B}} a \star b$$

Although the definition is a bit technical, it captures a natural idea in the context of concurrency since it mimics the interleaving semantics of the parallel composition operator. As an example, consider two programs P and Q and two possible executions of these programs e_P and e_Q. Since the rules Lpar and Rpar commute in the semantics given at the beginning of this section, it is easy to see that any interleaving of e_P and e_Q is a valid execution of $P \parallel Q$ and that any execution of $P \parallel Q$ is the interleaving of some executions of P and Q. Hence, if $\mathcal{P}$ and $\mathcal{Q}$ denote the set of all possible executions of P and Q, seen as labelled objects, then $\mathcal{P} \star \mathcal{Q}$ is the set of possible executions of $P \parallel Q$.

The *ordered product* $\mathcal{A} \boxbslash \mathcal{B}$ of two labelled classes $\mathcal{A}$ and $\mathcal{B}$ has a simpler definition, although it is less common. Unlike the labelled product, it does not appear in [5]. Resources on this operator and its unlabelled counterpart can be found in [16]. It is defined by

$$\mathcal{A} \boxbslash \mathcal{B} = \left\{ ((a,b),\sigma) \mid (a,\sigma_a) \in \mathcal{A}, (b,\sigma_b) \in \mathcal{B}, \sigma(i) = \begin{cases} \sigma_a(i) & \text{if } i \leq |a| \\ \sigma_b(i-|a|)+|a| & \text{otherwise} \end{cases} \right\}.$$

The ordered product simply shifts the labelling σ_b of its second component b so that it does not overlap with the labelling σ_a of its first component a. More eloquently, in an ordered product, the first component always has the smallest labels and the second component has the largest. In the context of concurrency, this captures the semantics of the sequential composition operator.

In fact, all the constructions of the NFJ language can be mapped to one of the combinatorial constructions we have seen so far. Hence, we can define a *combinatorial specification* of the set of the executions of any program P by induction on the syntax. However, in order to add more information in the specification, which will become useful later for random generation, we introduce a last notion, markers. A marker is a combinatorial class that contains only one object of size 0 and which purpose is only to distinguish one position in the objects or one subset of the objects (e.g. those that contains the markers by opposition to those that do not). For instance, if we consider again two NFJ programs P and Q and their respective sets of possible executions $\mathcal{P}$ and $\mathcal{Q}$, the set of the executions of $(P+Q)$ can be specified by $\mathcal{Y}_\ell \star \mathcal{P} + \mathcal{Y}_r \star \mathcal{Q}$ where $\mathcal{Y}_\ell$ (resp. $\mathcal{Y}_r$) is a marker class marking the executions of $(P+Q)$ taking the P branch (resp. the Q branch). In a sense this is a *more precise* specification than $\mathcal{P} + \mathcal{Q}$ since it carries more information.

The function S mapping a program to the specification of its executions is inductively defined in Table 3. As a convenience, we assume that all the choices in the program are given a unique identifier i (this is pictured by $+_i$ in all the formulas) so that we can assign them two marker classes $\mathcal{Y}_{i,\ell}$ and $\mathcal{Y}_{i,r}$ marking respectively the executions taking their left branch and their right branch.

The generating functions given in the third column of Table 3 is the generating function of the executions of the program. It is actually a function of several variables: the main variable z counting the number of atoms in the program and the marker variables ($y_{i,\ell}$, $y_{i,r}$, ...) marking the different choices. It generalises the generating functions with one variable introduced above so that if $\vec{y}$ is a product of $y_{i,\ell}$ and $y_{i,r}$ variables (where each variable may appear at most once), then the coefficient in front of $z^n\vec{y}$ in the series is the number of execution of size n of the program such that for all i, $y_{i,\ell}$ (resp. $y_{i,r}$) appears in $\vec{y}$ if and only if the left (resp. right) branch of choice number i is taken. In short, the markers encode the local choices while the atomic class $\mathcal{Z}$ encodes the number of actions of the execution.

The operation denoted by $P(z) \circledcirc Q(z)$ is called the coloured product and is introduced in [16]. The symbol $\circledcirc$ originally denotes an operation on combinatorial specification and we overload it here to denote an operation on generating functions.

Table 3: Recursive combinatorial specification of the set of executions of a program and its corresponding generating function

Language construction P	Specification $S(P)$	Generating function $P(z)$
a	$\mathcal{Z}$	z
$P \parallel Q$	$S(P) \star S(Q)$	$P(z) \circledcirc Q(z)$
$P; Q$	$S(P) \boxbslash S(Q)$	$P(z)Q(z)$
$P +_i Q$	$\mathcal{Y}_{i,\ell} \times S(P) + \mathcal{Y}_{i,r} \times S(Q)$	$y_{i,\ell}P(z) + y_{i,r}Q(z)$

where $\sum_{n\geq 0} a_n z^n \circledcirc \sum_{n\geq 0} b_n z^n = \sum_{n\geq 0} \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} z^n$.

Remark 1. *The reader familiar with combinatorics might find it odd that we use* ordinary *generating functions (OGF) rather than exponential ones (EGF) which are generally more suitable for labelled classes. The reason behind this choice is that we are actually facing operators from both worlds here. The ordered product* $\boxbslash$ *expresses in the labelled terms an operation which behaves better in the unlabelled world, and it indeed gives a nice formula in terms of OGF but not in terms of EGF. On the other hand the labelled product* $\star$ *is intrinsically labelled and behaves well only in terms of EGF. Since there is no obvious choice here between the two, we opt for ordinary generating functions for implementation reasons. They require integer arithmetic whereas EGFs would require to deal with rational numbers, which would be less efficient in practice.*

The generating function $P(z)$ of $S(P)$ captures insightful counting information about the program. For instance, the total number of executions of P is obtained by substituting 1 for every variable in $P(z)$. Finer-grained information can also be obtained. For example, given an integer i, the generating function of the subset of the execution of P taking the left branch at choice i is obtained by substituting 1 for $y_{i,\ell}$ and 0 for $y_{i,r}$. The number of such executions can then be obtained by substituting 1 for the remaining variables.

As an example, for the beverage vending machine $P = m; (w \parallel ((t +_1 (c; g)); (s +_2 n); p)); e$, whose control graph is pictured in Figure 3, we get the specification $S(P) = \mathcal{Z} \boxbslash (\mathcal{Z} \star ((\mathcal{Y}_{1,\ell} \star \mathcal{Z} + \mathcal{Y}_{1,r} \star (\mathcal{Z} \boxbslash \mathcal{Z})) \boxbslash (\mathcal{Y}_{2,\ell} \star \mathcal{Z} + \mathcal{Y}_{2,r} \star \mathcal{Z}) \boxbslash \mathcal{Z})) \boxbslash \mathcal{Z}$. From this specification we get the following generating function by applying the rules of the symbolic method described in Table 3: $P(z) = 4y_{1,\ell}(y_{2,\ell}+y_{2,r})z^6+5y_{1,r}(y_{2,\ell}+y_{2,r})z^7$. The number of executions taking the left branch of choice 1, that is choosing tea over coffee, is obtained by substituting 0 for $y_{1,r}$ and 1 for all the remaining variables. This yields 8 whereas the number of executions taking the right branch of choice is 10. This tiny example already shows that sampling executions by choosing one branch of each choice with probability $\frac{1}{2}$ and scheduling the rest of the actions introduces some bias in the generation. While this bias is harmless on such a small example, it can be dramatic in terms of coverage for larger programs as we demonstrate in Section 4.

2.4. Statistical analysis

We now tackle the problem of exploring the state-space of a given process through random generation. To this end we describe a *uniform* sampler of executions which relies on the counting information contained in the generating function of the program. Our random sampler thus requires the computation of this function as a pre-processing step, which can be done in polynomial time and space. We thus do *not* need to need the explicit, costly construction of the state-space.

2.4.1. Preprocessing: the generating function of executions

As explained in the previous section, the symbolic method gives a systematic way of computing the generating function of the class $S(P)$ of the executions of a program P. However, some care must be taken on the memory representation of this function. Fortunately for us, since the state-space is finite, the generating function of the executions of a program is a polynomial and not an infinite power series, but it has multiple variables encoding the different local choices. We also saw in Theorem 2 that this number of global choices was exponential so fully expanding the generating function of $S(P)$ would yield an exponential number of terms which constrains us to seek a more compact representation.

A more suitable representation is to only expand on the z variable, that is we represent the generating function of $S(P)$ as a dense polynomial in z whose coefficients are arithmetic expressions stored as trees sharing some common sub-structures. More precisely, an arithmetic expression is a binary tree whose internal nodes store a flag indicating whether the node corresponds to an addition or a multiplication, and whose leaves are either a pair of the form (i, s) indicating a $y_{i,s}$ variable or an integer. Moreover, the implementation of these coefficients must use hash-consing (see [17]). That is to say that when an expression (resulting from previous computations) is used several times, it should not by copied but referenced by a pointer. Note that this is different from optimal compaction where common sub-terms are systematically compacted. Finally, the generating function of $S(P)$ is stored as an array of such coefficients such that the coefficient at position i is the coefficient of degree i in z. An example of such a polynomial is pictured in Figure 4.

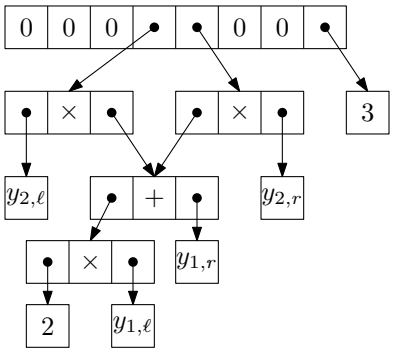


Figure 4: The compact tree-based memory representation of $y_{2,\ell}(2y_{1,\ell} + y_{1,r})z^3 + y_{2,r}(2y_{1,\ell} + y_{1,r})z^4 + 3z^7$. Note that the common sub-term $(2y_{1,\ell} + y_{1,r})$ is shared by two expressions.

A straightforward application of the symbolic method leads to Algorithm 1 for computing the generating function of the executions of a program. In the algorithm, the multiplications and additions of two coefficients are implemented as the allocation of a new tree node whose two children are the two operands.

The coloured product $\circledcirc$ used in the parallel composition case can be implemented similarly to the "text-book" polynomial multiplication using the formula given in the algorithm. Note however that computing each binomial coefficient individually incurs a non-negligible costs since they require big-integer arithmetic and cannot be obtained in constant time. This cost can be reduced significantly by re-using the value of the last-computed binomial coefficient at each iteration step using the formula $\binom{n}{k+1} = \binom{n}{k}\frac{n-k}{k+1}$. The whole procedure is presented in Algorithm 2 and the trick used to compute the binomial coefficient faster is at line 7. This technique allows to lower the cost

Algorithm 1 Computation of the generating function of the executions of an NFJ program

Input: An NFJ program P
Output: The generating function of $S(P)$

function GFUN(P)
  if $P = a$ **then return** z
  else if $P = (Q \parallel R)$ **then return** GFUN $(Q) \circledcirc$ GFUN (R)
  else if $P = (Q; R)$ **then return** GFUN $(Q) \cdot$ GFUN (R)
  else if $P = (Q +_i R)$ **then return** $y_{i,\ell}$GFUN(Q) + $y_{i,r}$GFUN(R)

where $\sum_{n\geq 0} a_n z^n \circledcirc \sum_{n\geq 0} b_n z^n = \sum_{n\geq 0} \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} z^n$.

of computing one binomial coefficient to only one multiplication and one division of a big integer by a small integer fitting in a machine word.

Algorithm 2 Computation of the coloured product of two polynomials

Input: Two polynomials P and Q of respective degrees d_P and d_Q
Output: The coloured product of P and Q

```
1:  function ⊙(P, Q)
2:      R ← array of length d_P + d_Q + 1
3:      for n from 0 to d_P + d_Q + 1 do
4:          b ← 1
5:          c ← P[0] · Q[n]
6:          for k from 1 to n do                ▷ Invariant: upon entering the loop b = (n choose k−1)
7:              b ← b · (n − k + 1)/k
8:              c ← c + P[k] · Q[n − k] · b
9:          R[n] ← c
10:     return R
```

Theorem 5 (Complexity of Algorithm 1)**.** *Algorithm 1 can be implemented in complexity $O(|P|^2)$ in terms of memory allocations and arithmetic operations on big integers, where $|P|$ denotes the number of atomic actions in P.*

Proof. For all programs P, let $C(P)$ denote the number of arithmetic operations on *coefficients* performed by GFUN(P). All the binary operators of NFJ have a cost that is at most of the order of one polynomial multiplication. Moreover, note that the degree of the generating function of P (which, recall, is a polynomial) is at most $|P|$. Hence, there exists a constant $\alpha \geq 1$ such that for all P, Q we have $C(P; Q), C(P + Q), C(P \parallel Q) \leq 2\alpha|P||Q| + C(P) + C(Q)$.

We prove by induction on the syntax of programs that $C(P) \leq \alpha|P|^2$. This is trivially true when $P = a$ and if $\bullet$ denotes any of the three other operators, we have $C(P \bullet Q) \leq 2\alpha|P||Q| + \alpha(|P|^2 + |Q|^2) \leq \alpha(|P| + |Q|)^2 = \alpha|P \bullet Q|^2$.

Note that it is crucial to use hash-consing for this results to hold. Said differently, one arithmetic operation on expressions must only consist in the allocation of a new tree node and must *not* perform any deep copy. □

2.5. Random sampling of executions

Our random sampling algorithm builds on ideas from the so-called *recursive method*, which is due to Nijenhuis and Wilf [18]. We use a two-step process to generate uniform executions. First, we select one of the global choices of the program with probabilities depending on the number of labellings of each global choice. Second, once a global choice has been selected, we draw one of the possible labellings of this choice uniformly at random. In other words, at the first step we *bias* our generation of a global choice so that the overall process remains uniform in terms of executions.

In order to sample a global choice, we first choose the *size* of the choice to be sampled. To this end, recall that the coefficient of degree k of the generating function encodes all executions of length k and that they correspond to global choices of size k. Besides, the coefficient of degree k of the generating function can be seen as the generating function of the size-k executions of the program. Thus, substituting 1 for every variable occurring in it yields the total number of size-k executions. The size of the global choice to be sampled must therefore be chosen with a probability proportional to the full evaluation of its corresponding coefficient. This is implemented in Algorithm 3. For example, for the beverage vending machine, we have $P(z) = 4y_{1,\ell}(y_{2,\ell}+y_{2,r})z^6+5y_{1,r}(y_{2,\ell}+y_{2,r})z^7$ which has two coefficients. Substituting 1 for all y variables yields $P(z) = 8z^6 + 10z^7$, which tells us that the global choice to be sampled must have size 6 with probability $\frac{8}{8+10} = \frac{4}{9}$ and size 7 with probability $\frac{5}{9}$.

Algorithm 3 Random sampling of a coefficient of the generating function of P

Input: The generating function $P(z)$ of the executions of a program P

```
function SAMPLE_SIZE(P(z))
    for c · z^d ∈ P(z) do
        W_d ← EVAL_COEFF (c)
    m ← RAND_UNIF ([1; ∑ W_d])
    d ← the minimum index d s.t. m ≤ ∑_{d'≤d} W_{d'}
    return d
```

Now, recall that a coefficient is an arithmetic expression encoded as a tree whose internal nodes are sums ($+$) or products ($\times$). Once a coefficient has been selected, we traverse this coefficient recursively, from top to bottom and, select only one child of each sum node we encounter, and collect y variables along the way. The idea is to construct a global choice from these y variables since each one of them encodes a local choice. More precisely, at each sum node $e_1 + e_2$ in the traversal, we compute the total number of executions, $e_1(1)$ and $e_2(1)$, encoded by both terms and we choose the term i (for $i \in \{1, 2\}$) with probability $\frac{e_i(1)}{e_1(1)+e_2(1)}$. Conversely, at each product node we traverse recursively both children since they contribute to two "parts" of the same executions whereas the children of a sum node contribute to two disjoint sets of executions. In the end, the set of y variables that we have seen in the process are interpreted as follows: $y_{i,\ell}$ corresponds to choosing the left branch of choice i and $y_{i,r}$ corresponds to choosing the right branch. This is described in more detail in Algorithm 4 which returns a list of y variables.

The function EVAL_COEFF used in both Algorithms 3 and 4 fully evaluates a expression by substituting 1 for all its variables. It is given in Algorithm 5 for the sake of completeness.

Finally, once the local choices returned by Algorithm 4 are applied to the program, that is to say that all the unused branches are removed, there remains to sample a uniform execution of the remaining choice-free program. This has been covered in [14] which proposes, in particular, an

Algorithm 4 Random sampling of a global choice of a given size

Input: An arithmetic expression e
Output: A list of local choices

```
function SAMPLE_CHOICE(e)
    if e = e_1 + e_2 then
        p_1 ← EVAL_COEFF (e_1)
        p_2 ← EVAL_COEFF (e_2)
        if BERNOULLI (p_1/(p_1+p_2)) then return SAMPLE_CHOICE (e_1)
        else return SAMPLE_CHOICE (e_2)
    else if e = e_1 × e_2 then return CONCAT (SAMPLE_CHOICE (e_1), SAMPLE_CHOICE (e_2))
    else if e = y_{i,s} then return [y_{i,s}]
    else if e = e'/n then return SAMPLE_CHOICE (e')
    else if e = n then return []                                   ▷ empty list
```

Algorithm 5 Full evaluation of an expression

Input: An arithmetic expression e encoded as a tree
Output: An integer

```
function EVAL_COEFF(e)
    if e = e_1 + e_2 then return EVAL_COEFF (e_1) + EVAL_COEFF (e_2)
    else if e = e_1 × e_2 then return EVAL_COEFF (e_1) · EVAL_COEFF (e_2)
    else if e = y_{i,s} then return 1
    else if e = n then return n
```

algorithm that is optimal in terms of random bits. We do not recall this algorithm here. The complete procedure for sampling a uniform execution in P is given in Algorithm 6. Naturally, the pre-processing step at line 2 must only by done once if one wishes to sample several executions for the same program.

Algorithm 6 Full procedure for the uniform sampling of executions in NFJ

Input: an NFJ program P
Output: a uniform execution of P

```
1: function SAMPLE_EXEC(P)
2:     P(z) ← GFUN (P)
3:     k ← SAMPLE_SIZE (P(z))
4:     e ← the coefficient of degree k of P(z)
5:     y⃗ ← SAMPLE_CHOICE (e)
6:     P' ← apply the global choice y⃗ to P
7:     return SAMPLE_CHOICEFREE (P')          ▷ See [14] for SAMPLE_CHOICEFREE
```

If we set aside the complexity of the pre-processing step, which has already been covered in Theorem 5, most of the remaining cost of the generation is hidden in the SAMPLE_SIZE and SAMPLE_CHOICE functions. First, these functions need to evaluate the coefficients of the generating functions. Since these evaluations, as well as the evaluations of some of their sub-terms, are to be re-used later in the generation process, they must be cached using a dynamic programming

approach. This implies that the memory layout presented in Figure 4 should be adapted to reserve some space for one big-integer in each tree node. Second, these functions need to traverse a whole coefficient and to draw Bernoulli random variables. Since the evaluation and caching part must only be done once too, it can be performed during the pre-processing step. Moreover, each node of each coefficient incurs one arithmetic operation on big integers, so the complexity of this part of the algorithm in terms of arithmetic operations is of the same order as the space complexity of Algorithm 1. In addition, the traversal of a coefficient requires a similar number of memory accesses and a linear number of calls to the generator of Bernoulli variables. Theorem 6 summarises these remarks and is the main result of this section.

Theorem 6 (complexity of the random sampling algorithm)**.** *Sampling uniform random executions of a program P requires:*

- *a pre-processing step of complexity $O(|P|^2)$ in terms of memory allocations and arithmetic operations on big integers;*
- *the generation of a linear number of Bernoulli variables and $O(|P|^2)$ memory accesses.*

Moreover, all the big integers at play here are bounded by $|P|!$ so their binary size is bounded by $|P| \log_2 |P|$.

Proof. The complexity of the pre-processing and of the random generation have already been discussed and only the binary size of the integers remains to prove. All the integers we manipulate in the algorithms are bounded by the maximum possible number of executions of a program. A straightforward induction shows that for any programs P and Q, the total number of executions of $(P; Q)$ and of $(P + Q)$ is upper-bounded by the total number of executions of $(P \parallel Q)$. Hence, the maximum possible number of executions of a program of size n is obtained when the program is made only of atomic actions and parallel compositions, which corresponds to $n!$. □

2.6. Experimental study

In order to assess experimentally the efficiency of our method, we put into use the algorithms presented here and demonstrate that they can handle systems with a significantly large state space. We generated a few NFJ programs at random using a Boltzmann random generator. All the polynomial operations and coefficients were implemented in OCaml.

Note that we did not optimize our code for efficiency nor ran extensive benchmarking, hence the numbers we give should be taken as a rough estimate of the performance of our algorithms. For the sake of reproducibility, the source code of our experiments is available in the companion repository[2]at `https://gitlab.com/ParComb/libnfj`.

Table 4 reports the runtime of the preprocessing phase (Algorithm 1), the runtime of the random sampler (Algorithm 6) and the number of executions of various programs. For the runtime of the counting algorithm, every measurement was performed 7 times and we reported the median of these 7 values. For the random sampler, every measure was performed 101 times and for each one we report the median of these values as well as the interquartile range (IQR)[3], which gives an idea

[2]All the benchmarks were run on a standard laptop with an Intel Core i7-8665U and 32G of RAM running Ubuntu 20.10 with kernel version 5.8.0-48-generic. We used OCaml version 4.08.1 and GMP version 6.2.0.

[3]The interquartile range of a set of measures is the difference between the third and the first quartiles. Compared with the value of the median, it gives a rough estimate of the dispersion of the measures.

Table 4: Quick benchmark of the counting and random sampling functions of executions for loop-free programs

size	# executions	mem. size	GFUN	UNIFEXEC	IQR
100	$1.168 \cdot 2^{108}$	65.30K	0.000091s	0.010ms	0.001ms
200	$1.956 \cdot 2^{199}$	235.12K	0.000245s	0.022ms	0.001ms
500	$1.249 \cdot 2^{645}$	2.21M	0.004563s	0.091ms	0.007ms
1000	$1.012 \cdot 2^{903}$	5.92M	0.011524s	0.135ms	0.008ms
2000	$1.354 \cdot 2^{2381}$	50.40M	0.076030s	0.429ms	0.093ms
3000	$1.682 \cdot 2^{6331}$	591.75M	0.987996s	1.562ms	0.309ms
5000	$1.464 \cdot 2^{10085}$	1.92G	2.959532s	3.239ms	0.413ms

of the dispersion of the measures. We use these metrics rather than the mean and the variance to reduce the importance of extremal values and give a precise idea of what runtime the user should expect when running our sampler. The time reported is the CPU time. The state-space column indicates the total number of executions. Finally, the mem. size column reports the amount of memory occupied by the generating function of executions computed by GFUN.

3. Extending the model with loops

This section is devoted to extending our model with loops. This is a significant improvement in terms of expressiveness, but has major implications on the state-space of programs.

First, programs may now have an infinite number of executions, and as a consequence, there is no *uniform* distributions over their executions any more. To circumvent this issue, we turn to the uniform generation of executions of a given length n where n is given as an input of the problem. Such a sampler can also be used, in conjunction with a particular procedure to select a length at random according to some particular distribution, for instance to sample uniform executions of length at most n.

A second, more significant, consequence of adding loops is that it interacts with the non-deterministic choices, as a choice may occur inside a loop and thus may be duplicated multiple times as we unroll the loop. Thus, the notion of global choice we have defined in the previous section, allowing us to decide of *all* the choices at once and then executing the rest of the program, does not extend well in the presence of loops. In a way, by introducing loops, we trade the clean separation we had between the non-determinism and the interleaving semantics of NFJ for more expressiveness.

As a consequence, we must take another approach to random sampling in this section. We will use the structure of the program to guide the generation rather than the structure contained in a multi-variate generating function as before.

3.1. Non-deterministic fork-join processes with loops

We start by extending the model from the previous section with a loop construction expressing that a program may be executed any number of times. Also, the empty program 0 used in the semantics of Section 2 as a convenience, is now usable in the syntax. The complete updated grammar of NFJ terms is given below. Note that only the two last lines are new.

$$
\begin{array}{lcll}
P, Q & ::= & P \parallel Q & \text{parallel composition} \\
& | & P; Q & \text{sequential composition} \\
& | & P + Q & \text{non-deterministic choice} \\
& | & a \in \mathcal{A} & \text{atomic action} \\
& | & P^\star & \text{loop} \\
& | & 0 & \text{empty program}
\end{array}
$$

As in the previous section, programs are considered up to alpha-equivalence and atomic actions are assumed to occur only once within a term. Again, since we only model the control-graphs of concurrent processes, the loop construction expresses that the body of the loop may be executed any number of times but does not state under which condition we exit the loop. Informally, the loop $P^\star$ may have either zero iteration, in which case it behaves as 0, or at least one, in which case it behaves as $(P; P^\star)$. We introduce the empty program in the grammar here, not only as a convenience, but also because it provides a slight gain of expressiveness, as it allows to write program such as $(0 + P)$ which express optional computations.

The semantics of programs must be updated to express the behaviour of loops. We first define a nullable predicate which indicates whether a program may terminate without firing any action. We can start the next iteration of a loop only if the current iteration is nullable.

$$
\begin{array}{ll}
\text{nullable}(P \parallel Q) = \text{nullable}(P) \wedge \text{nullable}(Q) & \text{nullable}(0) = \top \\
\text{nullable}(P; Q) = \text{nullable}(P) \wedge \text{nullable}(Q) & \text{nullable}(a) = \bot \\
\text{nullable}(P + Q) = \text{nullable}(P) \vee \text{nullable}(Q) & \text{nullable}(P^\star) = \top
\end{array}
$$

The reduction relation $P \xrightarrow{a} P'$ introduced in the first section is then extended to loops. Note that we also need to modify the reduction rule for the sequential composition. Since we now have non-empty programs which may terminate without firing any action (the nullable programs), we now want to allow the right-hand-side of a sequence to start its execution whenever the left-hand-side is nullable. The full new list of reduction rules is given below in Figure 5.

$$
\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}\ (\text{Lpar}) \qquad
\frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}\ (\text{Rpar}) \qquad
\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q}\ (\text{Lseq})
$$

$$
\frac{\text{nullable}(P) \quad Q \xrightarrow{a} Q'}{P; Q \xrightarrow{a} Q'}\ (\text{Rseq}) \qquad
\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}\ (\text{Lchoice}) \qquad
\frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}\ (\text{Rchoice})
$$

$$
\frac{}{a \xrightarrow{a} 0}\ (\text{act}) \qquad
\frac{P \xrightarrow{a} P'}{P^\star \xrightarrow{a} P'; P^\star}\ (\text{loop})
$$

Figure 5: Semantic of NFJ with loops

We call "execution step" a *proof-tree* built from the above rules and we define an execution as a sequence of such steps leading to a nullable term.

Definition 10 (Execution)**.** *An execution of an NFJ program P_0 is a sequence of steps of the form $P_0 \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \dots \xrightarrow{a_n} P_n$, such that* nullable$(P_n)$ *holds, and where for all i, $P_{i-1} \xrightarrow{a_i} P_i$ is a* proof-tree*, that is it contains all the applied rules and not simply its conclusion.*

We refer to the set of all possible executions of a program as its state-space*.*

As an example the program $a^{\star\star}$ has two executions of length 2, both firing a twice. One corresponds to the case where the outer loop is only unrolled once (i.e. the (*loop*) rule is only applied once) but the inner loop twice. The other corresponds to the case where the outer loop is unrolled twice and the two occurrences of the inner loop once. The first step of both executions is the same and is depicted below:

$$\dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} 0}\,(\text{act})}{a^\star \xrightarrow{a} 0; a^\star}\,(\text{loop})}{a^{\star\star} \xrightarrow{a} (0; a^\star); a^{\star\star}}\,(\text{loop})$$

Then the second step of the two executions are the following. On the left, the inner loop fires a second a while, on right, the first iteration of the inner loop terminates (we apply the (Rseq) rule at the top level) and a second iteration of the outer loop starts.

$$\dfrac{\dfrac{\text{nullable}(0) \qquad \dfrac{\dfrac{}{a \xrightarrow{a} 0}\,(\text{act})}{a^\star \xrightarrow{a} 0; a^\star}\,(\text{loop})}{0; a^\star \xrightarrow{a} 0; a^\star}\,(\text{Rseq})}{(0; a^\star); a^{\star\star} \xrightarrow{a} (0; a^\star); a^{\star\star}}\,(\text{Lseq})$$

$$\dfrac{\text{nullable}(0; a^\star) \qquad \dfrac{\dfrac{\dfrac{}{a \xrightarrow{a} 0}\,(\text{act})}{a^\star \xrightarrow{a} 0; a^\star}\,(\text{loop})}{a^{\star\star} \xrightarrow{a} (0; a^\star); a^{\star\star}}\,(\text{loop})}{(0; a^\star); a^{\star\star} \xrightarrow{a} (0; a^\star); a^{\star\star}}\,(\text{Rseq})$$

We will take the following program as a running example for the rest of the section: $P_0 = ((a + (b \parallel c))^\star \parallel (d + 0))^\star; (e + (f \parallel g))$. This program has one length-1 execution firing only e and four length-2 executions respectively firing fg, gf, ae and de.

3.2. Combinatorial interpretation

From a combinatorial point of view, the introduction of loops in the language has two levels of implication.

Syntax. At the syntactic level first, it is still possible to interpret the set of NFJ programs as a combinatorial class but this requires to choose a more adequate notion of size. Recall that a combinatorial class is given by a set of objects and a size function such that there is a finite number objects of size n for all integer n. The number of actions contained in a program is not eligible as a size function since there is an infinite number of (syntactic) programs with one action, for instance $a, a^\star, a^{\star\star}$, etc. A more adequate notion of size is the number of constructors used to build a program, that is

$$\begin{aligned} &|P; Q|_c = |P + Q|_c = |P \parallel Q|_c = 1 + |P|_c + |Q|_c \\ &|P^\star|_c = 1 + |P|_c \\ &|a|_c = |0|_c = 1. \end{aligned} \tag{9}$$

We use the notation $|P|_c$ to denote this new size function in order to avoid confusion with the number of atoms of a program $|P|$. Using, this size function, the class $\mathcal{F}$ of NFJ programs can be specified by

$$\begin{aligned}\mathcal{F} &= \mathcal{Z} \times \mathcal{F} \times \mathcal{F} \\ &+ \mathcal{Z} \times \mathcal{F} \times \mathcal{F} \\ &+ \mathcal{Z} \times \mathcal{F} \times \mathcal{F} \\ &+ \mathcal{Z} \\ &+ \mathcal{Z} \times \mathcal{F} \\ &+ \mathcal{Z}.\end{aligned} \tag{10}$$

Two differences must be noted compared to Section 2. First, the size function is different so that both the atomic action and the empty program have size 1 (hence the two $\mathcal{Z}$), and each constructor "costs" one $\mathcal{Z}$ since they are counted in the size. The second different is that we have to more terms, one for the empty program $\mathcal{Z}$, and one for loop terms $\mathcal{Z} \times \mathcal{F}$. From this specification, we get that the generating function f of NFJ programs satisfies $f(z) = z(2 + f(z) + 3f(z)^2)$. This equation can be solved explicitly and we get

$$f(z) = \frac{1 - z - \sqrt{(1-z)^2 - 24z^2}}{6z} \tag{11}$$

By studying the behaviour of this function near its main singularity $\rho = (1 + 2\sqrt{6})^{-1}$, we can obtain the asymptotic number of program of size n using the transfer theorem from [5]. See the proof of Theorem 1 in the previous section for a similar but more detailed proof. This leads to the following result.

Theorem 7. *The number of NFJ programs (with loops) of size n is equivalent to $Cn^{-\frac{3}{2}}\rho^{-n}$ when $n \to \infty$, where $\rho^{-1} = 1 + 2\sqrt{6} \approx 5.898979$ and $C = \dfrac{1}{18\sqrt{\pi}}\sqrt{4 + \sqrt{2/3}} \approx 0.068789$.*

This can be compared with Theorem 1 using the property that a binary tree with n leaves has $n-1$ internal nodes. This implies that if a program is built only using the constructors from the previous section and has n atomic actions, then it is made of $2n-1$ constructors. Thus, when n denotes the number of *constructors*, the exponential factor in Theorem 1 becomes $\sqrt{12}^n$ (for odd values of n) and $\sqrt{12} \approx 3.464 < 5.899$.

Executions. As in the previous section, we define a specification $S(P)$ of the class of the executions of a program P. Some differences must be noted with the previous section. First, we need a new operator, the *ordered set* operator, modelling a sequence of increasingly labelled objects, which expresses the semantics of the loop. Like the *ordered product* $\boxdot$, this is an uncommon operator, which has been studied in [16]. Another difference is that the presence of the loop and the empty program makes possible for non-trivial programs to have the empty execution as a valid execution. Although this seems innocuous, it forces us to handle carefully some special cases in our specification to avoid counting the same execution twice. Finally, the generating function of the executions of $S(P)$ which we refer to as the generating function of the executions of P is not a polynomial any more, but is an infinite (but convergent) formal power series. The recursive definition of $S(P)$ and its generating function are given in Table 5. A detailed explanation of the different constructions is given below.

Table 5: Simplified recursive combinatorial specification of the set of executions of a program and its corresponding generating function

Language construction P	Specification $S(P)$	Generating function $P(z)$
0	$\mathcal{E}$	1
a	$\mathcal{Z}$	z
$P \parallel Q$	$S(P) \star S(Q)$	$P(z) \circledcirc Q(z)$
$P; Q$	$S(P) \boxbslash S(Q)$	$P(z)Q(z)$
$P + Q$ when nullable$(P) \wedge$ nullable(Q)	$S(P) + (S(Q) \setminus \mathcal{E})$	$P(z) + Q(z) - 1$
$P + Q$ otherwise	$S(P) + S(Q)$	$P(z) + Q(z)$
$P^\star$ when nullable(P)	$\textsc{Set}^{\boxbslash}(S(P) \setminus \mathcal{E})$	$(1 - (P(z) - 1))^{-1}$
$P^\star$ otherwise	$\textsc{Set}^{\boxbslash}(S(P))$	$(1 - P(z))^{-1}$

where $\sum_{n\geq 0} a_n z^n \circledcirc \sum_{n\geq 0} b_n z^n = \sum_{n\geq 0} \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} z^n$.

The empty program 0 and the atomic action a have only one execution, of length 0 and 1 respectively. This is modelled combinatorially by the neutral class $\mathcal{E}$ — the class containing only one element of size 0 — and the atom class $\mathcal{Z}$ — the class with only one element of size 1.

As before, the executions of $P \parallel Q$ are made of any interleaving of one execution of P and one execution of Q. For instance if $P = a + (b; c)$ and $Q = d^\star$, then P admits for instance an execution firing b and then c (denoted by bc for short) and Q admits an execution firing two ds (denoted by dd for short). Then all the 6 possible interleavings of these executions are executions of $P \parallel Q$: *bcdd*, *bdcd*, *bddc*, *dbcd*, *dbdc* and *ddbc* (again, we only denote the executions by their firing sequences for conciseness). Although the interpretation of these interleavings using increasing labellings is less obvious than is the previous section, the notion at play here is still well captured by the labelled product of combinatorics, denoted by $\star$.

The executions of $P; Q$ are given by an execution of P followed by an execution of Q. So for instance, using the same example programs P and Q as above, *bcdd* is an execution of $(P; Q)$ but not *dbcd*. So they can be seen as a pair of an execution of P and an execution of Q, which is still expressed using the ordered product $\boxbslash$.

The set of executions of $P + Q$ is the union of the executions of P and Q. Moreover this union is "almost" disjoint in the sense that the only execution that these programs may have in common is the empty execution, hence the two cases in the definition. Combinatorially, the fact that nullable(P) holds corresponds to the fact that the class of its executions contains one object of size 0, the empty execution. It is in fact important that we can express this in terms of *disjoint* unions because they fit in the framework of analytic combinatorics whereas arbitrary unions are more difficult to handle[4].

Finally, the executions of $P^\star$ are sequences of executions of P or, equivalently, sequences of *non-empty* executions of P. This second formulation leads to a non-ambiguous specification as the unique class $\mathcal{P}'$ satisfying $\mathcal{P}' = \mathcal{E} + \mathcal{P}_+ \boxbslash \mathcal{P}'$, where $\mathcal{P}_+$ denotes the non-empty executions of P.

[4]Grammar descriptions involving non-disjoint unions are referred to as "ambiguous" and lack most of the benefits, if not all, of the symbolic method, essentially because some objects may be counted multiple times when applying the method.

This implicitly defined class $\mathcal{P}'$ is denoted $\textsc{Set}^{\boxed{\star}}(\mathcal{P}_+)$ and is called the *sequence* of $\mathcal{P}_+$. Once again we must distinguish whether nullable(P) holds or not in the definition of $\mathcal{P}_+$ to avoid ambiguities and thus double-counting.

The S function described above maps each program to a combinatorial specification of its executions. As an example, for our example program P_0 introduced above, we have $S(P_0) = \textsc{Set}^{\boxed{\star}}(\textsc{Set}^{\boxed{\star}}(\mathcal{Z} + (\mathcal{Z} \star \mathcal{Z})) \star (\mathcal{Z} + \mathcal{E}) \setminus \mathcal{E}) \boxed{\star} (\mathcal{Z} + (\mathcal{Z} \star \mathcal{Z}))$. The generating function of the executions of a program, i.e. of the class $S(P)$, constitutes a condensed summary of the counting information of its state space. Our uniform random sampler of executions for NFJ programs with loops will use the generating function of each sub-term of P to generate an execution of P.

Before diving into the description of our random sampler, we want to give another example of application of analytic combinatorics, by showing how a few manipulations on polynomials can lead to interesting algorithmic applications and precise quantitative results. We already showed in Theorems 1 and 2 how to get the asymptotic number of programs and the average number of global choices of loop-free programs using this kind of techniques. Here, we study the generating function $P_0(z)$ of the example program P_0 given above, which we recall here for convenience: $P_0 = [(a + (b \parallel c))^\star \parallel (d + 0)]^\star; [e + (f \parallel g)]$. Let $P_0(z) = \sum_{n\geq 0} p_n z^n$ denote the expansion in power series of the generating function of $S(P_0)$ and recall that its n-th coefficient p_n is the number of executions of P_0 of length n. By applying the rules from Table 5 we obtain that:

$$\begin{aligned} P_0(z) &= \left[(1 - z - 2z^2)^{-1} \circledcirc (z + 1)\right]^{-1} \cdot [z + 2z^2] \\ &= \frac{(2z+1)(2z-1)^2(z+1)^2 z}{1 - 4z - 4z^2 + 6z^3 + 8z^4} \end{aligned}$$

The second line of the above formula is obtained by applying the calculus rule[5] $z \circledcirc A(z) = z\frac{d(zA(z))}{dz}$. From this formula we derive two applications. First, from the denominator of this rational expression we deduce that for all $n > 6$ we have $p_n - 4p_{n-1} - 4p_{n-2} + 6p_{n-3} + 8p_{n-4} = 0$. The obtained recurrence formula can be used to compute the number of executions of length n of P_0 in linear time. On the analytic side, $P_0(z)$ being a rational function, we can do a *partial fraction decomposition* to obtain $P_0(z)$ as a sum of four terms of the form $C_i(1 - z\rho_i^{-1})$ (plus a polynomial). Each of these terms expands as $\sum_{n\geq 0} C_i \rho_i^{-n} z^n$, hence the number of executions of P_0 of length n satisfies $p_n = C \cdot \rho^{-n} \cdot (1 + o(1))$ for some constants C and ρ and with an exponentially small error term hidden in the $o(1)$. In this case we have $\rho \approx 0.221987$, $C \approx 0.146871$ and the error term is of the order of 0.327950^n. Table 6 compares the values of p_n — computed using the aforementioned linear algorithm — and of the proposed approximation for a few values of n. One can see that already for small values of n, the relative error of this approximation is rather low.

3.3. Statistical analysis algorithms

We now study the problem of exploring the state-space of a given program through random generation. In the presence of loops, programs can have an infinite number of executions and there is thus no uniform distribution over executions. However, programs still have a finite number of executions of a given length, so we propose a uniform sampler of executions of fixed length. This sampler can be used to target executions of specific length n or in, conjunction with a procedure to

[5] This is the only "non-standard" computation rule we use in this example. All the rest is usual polynomial manipulations. General rules for computing $A(z) \circledcirc B(z)$ are beyond the scope of this article.

Table 6: Value of p_n, of its approximation $C \cdot \rho^{-n}$ and of the relative error $|p_n - C \cdot \rho^{-n}|/p_n$ for small values of n

n	6	7	8	9	10	11	...	30
p_n	1226	5528	24904	112196	505424	2276832	...	5985551205783341568
approx	1227	5529	24907	112199	505429	2276839	...	5985551205783353055
rel. err.	8.16e-4	1.81e-4	1.2e-4	2.67e-5	9.89e-6	3.07e-6	...	1.92e-15

sample a size, to sample a uniform execution of length bounded by n for instance. In Section 4, we will see an alternative approach for exploring the state-space, based on the generation of execution *prefixes*.

3.3.1. Preprocessing: the generating function of executions

As explained above, the symbolic method gives a systematic way of computing the generating function of the class of the executions of a program P from its specification $S(P)$. The computations rules are given in Table 5. Since we are in presence of infinite spate-spaces, these generating functions are not polynomials any more and become (convergent) formal power series. For the algorithms to remain practical, we only compute the $(n+1)$ first terms of these series, hence allowing us to sample uniform executions of length k for all $k \leq n$. Algorithm 7 implements the computation of the n first terms of the generating functions of *all* sub-terms of a program. The resulting (partial) generating functions must be stored in the tree representation of the program.

Algorithm 7 Computation of the generating function of the executions of an NFJ program, and all its sub-terms, up to degree n

Input: An NFJ program P and a positive integer n.
Output: The first $n+1$ terms of the generating function of P

```
function GFUN(P, n)
    if P = 0 then return 1
    else if P = a then return z
    else if P = Q ∥ R then return GFUN (Q, n) ⊙ GFUN (R, n) mod z^{n+1}
    else if P = Q; R then return GFUN (Q, n) · GFUN (R, n) mod z^{n+1}
    else if P = Q + R then
        q(z) ← GFUN (Q, n),    r(z) ← GFUN (R, n)
        return q(z) + r(z) − q(0)r(0)
    else if P = Q^⋆ then
        q(z) ← GFUN (Q, n)
        return (1 − (q(z) − q(0)))^{−1} mod z^{n+1}
```

The coloured product $\odot$ used in the parallel composition case can be implemented using the "naive" algorithm as in the previous section. There is a more efficient approach here though, because we are using integer coefficients rather than expressions. The idea is to use the combinatorial Laplace and Borel transforms to express this operation as a regular product. This is achieved by the formula $A \odot B = \mathcal{L}(\mathcal{B}(A) \cdot \mathcal{B}(B))$, where the Borel transform $\mathcal{B}$ is defined by $\mathcal{B}(\sum_{n\geq 0} a_n z^n) = \sum_{n\geq 0} \frac{a_n}{n!} z^n$ and the Laplace transform $\mathcal{L}$ is defined by $\mathcal{L}(\sum_{n\geq 0} a_n z^n) = \sum_{n\geq 0} n! a_n z^n$. More on the coloured product, the Borel and Laplace transform and their applications can be found in [16].

To be implemented efficiently using only integer rather than rational arithmetic, the coefficients of the result of the Borel transform should share $n!$ as a common denominator where n is the degree of the polynomial and the division by $n!$ should be postponed to the last moment. So if $A(z) = \sum_{k=0}^{n} a_k z^k$ and $B(z) = \sum_{k=0}^{m} b_k z^k$, then

$$\mathcal{B}(A) = \frac{1}{n!} \sum_{k=0}^{n} \frac{n!}{k!} a_k z^k = \frac{1}{n!} \tilde{A}(z) \qquad \mathcal{B}(B) = \frac{1}{m!} \sum_{k=0}^{m} \frac{m!}{k!} b_k z^k = \frac{1}{m!} \tilde{B}(z)$$

and

$$A(z) \circledcirc B(z) = \frac{1}{n!m!} \mathcal{L}(\tilde{A}(z) \cdot \tilde{B}(z)).$$

Thus, the coloured product can be implemented as in Algorithm 8 where the polynomial multiplication at line 13 is where most of the computational cost lies. The advantage of this approach is that it leaves the choice of the polynomial multiplication algorithm open and we can thus benefit from existing fine-tuned implementations of the algorithms from the literatures. The FLINT library [19] for instance provides such algorithms.

Algorithm 8 Fast implementation of the coloured product for integer polynomials

Input: Two integer polynomials $A(z)$ and $B(z)$ of respective degrees n and m and stored as arrays of integers
Output: The coloured product $A(z) \circledcirc B(z)$ of A and B

```
function COLPROD(A, B)
    Ã, B̃ ← copies of A and B
    f ← 1
    for k from n down to 1 do
        Ã[k] ← Ã[k] · f
        f ← f · k
    Ã[0] ← Ã[0] · f
    g ← 1
    for k from m down to 1 do
        B̃[k] ← B̃[k] · g
        g ← g · k
    B̃[0] ← B̃[0] · g
    R ← Ã · B̃
    for k from 0 to m + n do R[k] ← R[k]/(f · g)
    return R
```

The computation of $(1 - (q(z) - q(0)))^{-1}$ at the last line of Algorithm 7 can be carried out efficiently using the so-called Newton method (see [20, p. 259] and [21] for instance). The algorithm consists in computing the sequence $S_{i+1}(z) \leftarrow S_i(z) + S_i(z) \cdot ((q(z) - q(0)) \cdot S_i(z) - (S_i(z) - 1))$, starting fvom $S_0(z) = 1$, and until the $(n+1)$ first coefficients of $S_i(z)$ are the same as those of $(1 - (q(z) - q(0)))^{-1}$. The intuition behind this formula comes from the field of numerical analysis where the Newton method is used to get fast-converging approximations of real constants.

The key feature of this approximation scheme is that the error term, that is the difference between $S_i(z)$ and its limit $(1 - (q(z) - q(0)))^{-1}$, is squared at each iteration. As a consequence, the number of correct terms, in $S_i(z)$ doubles at each iteration and only a logarithmic number of

iterations of the formula is necessary to compute the $(n+1)$ first terms of the solution. To make this argument more formal, let $\tilde{q}(z) = q(z) - q(0)$, let $S(z) = (1 - \tilde{q}(z))^{-1}$ and let $E_i(z)$ denote the error term of $S_i(z)$, that is $E_i(z) = S(z) - S_i(z)$. By observing that $S(z) = 1 + \tilde{q}(z)S(z)$, we have that

$$\begin{aligned} S_{i+1}(z) &= S_i(z) + S_i(z)(S_i(z)\tilde{q}(z) - (S_i(z) - 1)) \\ &= S_i(z) + S_i(z)(S(z) - 1 - E_i(z)\tilde{q}(z) - S(z) + 1 + E_i(z)) \\ &= S(z) - E_i(z) + (S(z) - E_i(z))E_i(z)(1 - \tilde{q}(z)) \\ &= S(z) - (1 - \tilde{q}(z))E_i(z)^2 \\ \text{hence } E_{i+1} &= (1 - \tilde{q}(z))E_i(z)^2 \end{aligned}$$

Since $E_0(0) = 0$, we have that the first term of the expansion of $E_0(z)$ is zero and by induction the 2^i first terms of $E_i(z)$ are zero. As a consequence, the 2^i first terms of $S_i(z)$ are the same as the 2^i first terms of $S(z)$. Algorithm 9 implements this scheme. Note that in the formula at line 5, it is only necessary to compute the two products up to degree $2i$.

Algorithm 9 Computing the $n+1$ first terms of the quasi inverse of a polynomial using the Newton method

Input: A polynomial $q(z)$
Output: The $n+1$ first terms of $(1 - q(z) + q(0))^{-1}$ as a polynomial of degree n

```
1: function QINV(q)
2:     S(z) ← 1
3:     i ← 1
4:     while i < n + 1 do
5:         S(z) ← S(z) + S(z)(S(z)(q(z) − q(0)) − (S(z) − 1))
6:         i ← 2i
7:     return S(z)
```

Assume we have a so-called multiplication function $M : \mathbb{N} \to \mathbb{N}$, that is a function such that

- the number of arithmetic operations that are necessary to compute the product of two integer polynomials of degree at most n is at most $M(n)$;
- for all $n_1, n_2 \in \mathbb{N}$ we have $M(n_1) + M(n_2) \leq M(n)$.

It is generally accepted that such a function exists [22]. The following lemma expresses the complexity of Algorithm 9 as a function of $M(n)$.

Lemma 1. *Algorithm 9 can be implemented to compute the quasi-inverse* $(1 - (q(z) - q(0)))^{-1}$ *of* $q(z)$ *in* $O(M(n))$ *arithmetic operations on integers.*

Proof. At each iteration of the loop, two multiplication of polynomials of degree $2i$ are performed (the terms of higher degrees can be safely ignored). Thus, the total cost of the multiplications is $2M(n)+2M(n/2)+2M(n/4)+2M(n/8)\cdots \leq 2M(2n) = O(M(n))$. The additions only contribute to a lower order term. □

Theorem 8. *Let P be an NFJ program and let $|P|_c$ denote its syntactic size as defined in* (9)*. Algorithm 7 can be implemented to compute the first n coefficients of the generating function of the executions of P in $O(|P|_c M(n))$ operations on big integers, where $M(n)$ is the complexity of the multiplication of two polynomials of degrees at most n.*

Proof. The proof of Theorem 8 follows from the above discussion: each constructor incurs one polynomial operation among addition, multiplication, coloured product and quasi-inverse and all of them can be carried out in $O(M(n))$. □

An important question which complements Theorem 8 is that of the cost of one arithmetic operation. Since this cost is a function of the binary size of the integers, this can be reformulated into: what is the size of the integers at play? Theorem 9 gives an upper bound on these coefficients. This upper bound is expressed using the *height* of a program, that is its maximum number of nested operators, which is recursively defined by

$$\begin{aligned} &h(a) = h(0) = 0 \\ &h(P \parallel Q) = h(P + Q) = h(P; Q) = 1 + \max(h(P), h(Q)) \\ &h(P^\star) = 1 + h(P). \end{aligned}$$

Theorem 9. *Let P be an NFJ program and let $n \geq 0$. The number p_n of length-n executions of P is at most $2^{h(P)n}$ and its binary size $\lceil \log_2(p_n) \rceil$ is thus bounded by $h(P)n$.*

Proof. This upper bound can be proven by induction on P.

- It is trivially true for the base cases $P = 0$ and $P = a$.
- The number of length-n executions of $(P; Q)$ is upper-bounded by the number of length-n executions of $(P \parallel Q)$, which is itself bounded, by induction hypothesis, by

$$\sum_{k=0}^{n} \binom{n}{k} 2^{h(P)k + h(Q)(n-k)} = (n+1) 2^{\max(h(P),h(Q))n} \leq 2^{(1+\max(h(P),h(Q)))n}.$$

- For $n \geq 1$, the number of length-n executions of $(P + Q)$ is bounded by induction by $2^{h(P)n} + 2^{h(Q)n} \leq 2^{h(P+Q)n}$.
- Finally, if p_i denote the number of executions of length i of P, then the number of executions of $P^\star$ is given by

$$\begin{aligned} \sum_{k=1}^{n} \sum_{\substack{i_1, i_2, \ldots, i_k > 0 \\ i_1 + i_2 + \cdots + i_k = n}} p_{i_1} p_{i_2} \cdots p_{i_k} &\leq \sum_{k=1}^{n} \sum_{\substack{i_1, i_2, \ldots, i_k > 0 \\ i_1 + i_2 + \cdots + i_k = n}} 2^{h(P)(i_1 + i_2 + \cdots + i_k)} \\ &= \sum_{k=1}^{n} \sum_{\substack{i_1, i_2, \ldots, i_k > 0 \\ i_1 + i_2 + \cdots + i_k = n}} 2^{h(P)n} \\ &= 2^{h(P)n} \cdot 2^{n-1} \leq 2^{h(P^\star)n} \end{aligned}$$

□

To give a rough idea of the performance that can be achieved by Algorithm 7, we computed the generating function of P_0 up to degree $n = 10000$ — and thus its number of executions of length k for all $k \leq 10000$ — in less that 4s on a standard PC. A more detailed benchmark of Algorithm 7 is given in Section 3.4.

3.3.2. Random sampling of executions

In order to sample a uniform execution directly from the syntax of the program, we use the so-called "recursive method", as introduced in [23] and integrated into the analytic combinatorics framework in [24]. It operates in a similar fashion to the symbolic method, that is by induction on the specification, by combining the random samplers of the sub-structures with simple rules depending on the grammar construction. For the sake of clarity we represent executions as sequences of atomic actions in the presentation of the algorithm. This encoding does not contain all the information that defines an execution, typically it does not reflect in which iteration of a loop an atomic action is fired for instance. However it makes the presentation clearer and the algorithm can be easily adapted to a more faithful encoding. Our uniform random sampler of executions is described in Algorithm 10 and the detailed explanations about the different constructions are given below.

Algorithm 10 Uniform random sampler of executions of given length

Input: A program P and an integer n such that P has length n executions.
Output: A list of atomic actions representing an execution

1: **function** UnifExec(P, n)
2: **if** $n = 0$ **then return** the empty execution
3: **else if** $P = a$ **then return** a
4: **else if** $P = Q + R$ **then**
5: **if** Bernoulli $(\frac{q_n}{q_n + r_n})$ **then return** UnifExec (Q, n)
6: **else return** UnifExec (R, n)
7: **else if** $P = Q \parallel R$ **then**
8: draw $k \in [\![0; n]\!]$ with probability $\binom{n}{k} q_k r_{n-k}/p_n$
9: **return** shuffle (UnifExec (Q, k), UnifExec $(R, n - k)$)
10: **else if** $P = Q; R$ **then**
11: draw $k \in [\![0; n]\!]$ with probability $q_k r_{n-k}/p_n$
12: **return** concat (UnifExec (Q, k), UnifExec $(R, n - k)$)
13: **else if** $P = Q^\star$ **then**
14: draw $k \in [\![1; n]\!]$ with probability $q_k p_{n-k}/p_n$
15: **return** concat (UnifExec (Q, k), UnifExec $(P, n - k)$)

The lower case letters p_n, q_k, r_{n-k} etc. indicate the number of executions of length n, k, $n - k$ of programs P, Q and R.

Choice. The simplest rule of the recursive method is that of the disjoint union used at line 4 of Algorithm 10. If q_n and r_n denote the number of length-n executions of Q and R, then a uniform random length-n execution of $P = Q + R$ is a uniform length-n execution of Q with probability $q_n/(q_n + r_n)$ and a uniform length-n execution of R otherwise. One way to draw the Bernoulli variable is to draw a uniform random big integer x in $[\![0; q_n + r_n[\![$ and to return `true` if and only if $x < q_n$. As an example, consider the programs $Q = (a + (b \parallel c))$ and $R = d^\star$. We count

that Q has two executions of length two: bc and cb and R has only one: dd. Hence, to sample a length-2 execution in $(Q + R)$, one must perform a recursive call on Q with probability $2/3$ and on R with probability $1/3$.

Parallel composition. The other rules build on top of the disjoint union case. For instance, the set of length-n executions of $P = Q \parallel R$ can be seen as $\mathcal{Q}_0 \star \mathcal{R}_n + \mathcal{Q}_1 \star \mathcal{R}_{n-1} + \cdots + \mathcal{Q}_n \star \mathcal{R}_0$ where $\mathcal{Q}_k$ (resp. $\mathcal{R}_k$) denotes the set of length-k executions of Q (resp. R). By generalising the previous rule to disjoint unions of $(n + 1)$ terms, and using the fact that the number of elements of $\mathcal{Q}_k \star \mathcal{R}_{n-k}$ is $q_k r_{n-k}\binom{n}{k}$, one can select in which one of these terms to sample by drawing a random variable which is k with probability $q_k r_{n-k}\binom{n}{k}/p_n$. Then it remains to sample a uniform element of $\mathcal{Q}_k$, a uniform element of $\mathcal{R}_{n-k}$ and a uniform shuffling of their labellings among the $\binom{n}{k}$ possibilities. This is described at line 7 of Algorithm 10. We do not detail the implementation of the shuffling function here, an optimal algorithm in terms of random bits consumption, can be found in [14]. As an example, consider the same programs as above: $Q = (a + (b \parallel c))$ and $R = d^\star$. The number of length-3 executions of $(Q \parallel R)$ is $1 \cdot 1 \cdot \binom{3}{1} + 2 \cdot 1 \cdot \binom{3}{1} = 9$ using the decomposition $Q_1 \star R_2 + Q_2 \star R_1$. Say $k = 1$ is selected (with probability $1/3$), then the recursive calls to $(Q, 1)$ and $(R, 2)$ necessarily return a and dd and the SHUFFLE procedure must choose a shuffling uniformly between add, dad and dda.

Sequential composition. The case of the sequential composition is similar (see line 10 of Algorithm 10). We use the same kind of decomposition, using the Cartesian product $\times$ in place of the labelled product $\star$. This has the consequence of removing the binomial coefficient in the formula for the generation of the k random variable. Once k is selected, we generate an execution of Q_k, an execution of R_{n-k} and we concatenate the two.

Loop. Finally, the case of the loop is a slight adaptation of the case of the sequential composition using the fact that the executions of $Q^\star$ are the executions of $(0 + Q; Q^\star)$. However, care must be taken to avoid issues related to double-counting. More specifically, when sampling an execution of $(Q; Q^\star)$ we must not choose an execution of length 0 for the left-hand-side Q. This is related to the same reason we had to specify the executions of $Q^\star$ as all the sequences of *non-empty* executions of Q. This is presented at line 13 of Algorithm 10, note that $k > 0$. As an example, for sampling a length-3 execution in $(a + (b; c))^\star$, one may select $k = 1$ with probability $2/3$, which yields abc or aaa depending on the recursive call to $(Q^\star, 2)$ or $k = 2$, with probability $1/3$, which yields bca.

Generation of random variables. We did not give details on how to generate the random variable k for the parallel, sequential and loop case. In has been showed in [24, 25, 26] that good performance can be achieved by using the so-called boustrophedonic order. For instance, in the case of the sequential composition $P = (Q; R)$, the idea is to generate a random integer x in the interval $[\![0; p_n[\![$ and to find the minimum number ℓ such that the sum of ℓ terms $q_0 r_n + q_n r_0 + q_1 r_{n-1} + q_{n-1} r_1 + q_2 r_{n-2} + \cdots$ (taken in this particular order) is greater than x. Then k is such that the last term of this sum is $q_k r_{n-k}$. The key idea of this algorithm is that the worst case of this algorithm corresponds to choosing k close to $n/2$ which yields a divide and conquer scheme.

Theorem 10. *Using the boustrophedonic order, the complexity of the random generation of an execution of length n in P in terms of arithmetic operations on big integers is $O(n \cdot \min(\ln(n), h(P)))$ where $h(P)$ refers to the* height *of P.*

Contrary to the classical context of random generation that we have in analytic combinatorics (like in [24, 26, 25]), the grammar enumerating the executions to be sampled is not a constant but rather a parameter of the problem. Hence its size cannot be considered constant and the complexity analysis needs to be carefully crafted to take this variable into account.

Proof. The $O(n\ln(n))$ bound follows from Theorem 11 of [24]. We obtain the other bound by refining the result of Theorem 12 from [25].

The combinatorial classes we are considering are built from the $\star, \times, +$ and $\textsc{Seq}(\cdot)$ operators without recursion, they hence fall under the scope of *iterative classes* for which Molinero proved a linear complexity in n. However the proof given in [25] does not give an explicit bound for the multiplicative constants, which actually depends on the size of the grammar and which we cannot consider constant in our context. Let $C(P, n)$ denote the cost of $\textsc{UnifExec}(P, n)$ in terms of arithmetic operations on big integers. We show that $C(P, n) \leq \alpha n h(P)$ by induction for some constant α to be specified later.

- The base cases have a constant cost.
- The case of the choice only incurs a constant number c of arithmetic operations in addition to the cost of the recursive calls. Hence $C(Q+R, n)$ is bounded by $c+\alpha\max(C(Q, n), C(R, n)) \leq c+\alpha n\max(h(Q), h(R)) = c+\alpha n(h(Q+R)-1)$ by induction. Thus, if $\alpha \geq c$, then $C(Q+R, n) \leq \alpha n h(Q + R)$.
- The parallel composition case incurs a number of arithmetic operations of the form $c' \cdot \min(k, n - k)$ where k is the random variable generated using the boustrophedonic order technique. Hence $C(Q \parallel R, n)$ is bounded by $c'\min(k, n-k) + C(Q, k) + C(R, n-k)$ and by induction by $c'\min(k, n-k) + \alpha k h(Q) + \alpha(n-k)h(R) \leq \alpha n h(Q \parallel R) + c'\min(k, n-k) - \alpha n$. The last term on the right is bounded by 0 if $\alpha \geq c'$.
- Sequential composition is treated using the same argument as for parallel composition.
- Finally, the loop must be handled by reasoning "globally" on the total number of unrollings. Say the loop $Q^\star$ is unrolled r times. Then its cost is bounded by $c'\sum_{i=1}^{r}\min(k_i, k_{i+1} + \cdots + k_r) + \sum_{i=1}^{r+1} C(Q, k_i)$. The first sum is bounded by $c'n$ and the second is bounded by induction by $\sum_{i=1}^{r+1}\alpha k_i h(Q)$ which simplified to $\alpha n h(Q)$. Hence, reusing the bound $\alpha \geq c'$ and the fact that $h(Q^\star) = 1 + h(Q)$, we get $C(Q^\star, n) \leq \alpha n h(Q^\star)$ which terminates the proof.

□

3.4. Experimental study

In order to assess experimentally the efficiency of our method, we put into use the algorithms presented here and demonstrate that they can handle systems with a significantly large state space. We generated a few NFJ programs at random using a Boltzmann random generator. In its basic form, the Boltzmann sampler would generate a high number of loops and a large number of subterms of the form $P+0$ in the programs which we believe is not realistic so we tuned it using [27] so that the number of both types of nodes represent only 10% of the size of the program in expectation. We rely on the FLINT library (Fast Library for number theory [19]) to carry all the computations on polynomials except for the coloured product and the quasi-inversion using Newton iteration

which we implemented ourselves. The former was not provided natively by the library and the latter was feasible using FLINT's primitives but slow compared to the Newton method.

Note that besides the choice of the algorithms, we did not optimize our code for efficiency nor ran extensive benchmarking, hence the numbers we give should be taken as a rough estimate of the performance of our algorithms. For the sake of reproducibility, the source code of our experiments is available on the companion repository[6]at `https://gitlab.com/ParComb/libnfj`.

Table 7 reports the runtime of the preprocessing phase (Algorithm 7), the runtime of the random sampler (Algorithm 10) and the number of executions of length n of various programs of various values of n. For the runtime of the counting algorithm, every measurement was performed 7 times and we reported the median of these 7 values. For the random sampler, every measure was performed 101 times and for each one we report the median of these values as well as the interquartile range (IQR)[7], which gives an idea of the dispersion of the measures. We use these metrics rather than the mean and the variance to reduce the importance of extremal values and give a precise idea of what runtime the user should expect when running our sampler. The time reported is the CPU time as measured by C's `clock` function. The state-space column indicates the number of executions of length n. Finally, the mem. size column reports the amount of memory occupied by the generating functions of executions computed by GFUN.

size	len	# executions	mem. size	GFUN	UnifExec	IQR
100	500	$1.370 \cdot 2^{1119}$	898.98K	0.015s	0.241ms	0.020ms
100	1000	$1.690 \cdot 2^{2234}$	3.32M	0.053s	0.521ms	0.059ms
500	500	$1.071 \cdot 2^{1589}$	2.84M	0.087s	0.359ms	0.069ms
500	1000	$1.093 \cdot 2^{3102}$	10.38M	0.538s	1.094ms	0.133ms
1000	500	$1.096 \cdot 2^{2374}$	8.75M	0.289s	0.496ms	0.068ms
1000	1000	$1.579 \cdot 2^{4756}$	33.19M	1.645s	1.842ms	0.177ms
2000	500	$1.336 \cdot 2^{2273}$	10.42M	0.355s	0.914ms	0.293ms
2000	1000	$1.551 \cdot 2^{4624}$	39.20M	1.890s	1.119ms	0.128ms

Table 7: Quick benchmark of the counting and random sampling functions of executions

4. Execution prefixes generation

In this section, we describe a complementary tool to explore the state-space of a program: a uniform random sampler of execution *prefixes*. Execution prefixes offer a more fine-grained tool to explore the state-spaces as a sampler of prefixes can be combined with other tools or with custom heuristics to bias the random generation toward regions of interest of the state space. We see this algorithm as a building block to construct exploration strategies and the fact that it is *uniform* over prefixes of a given length implies that is still gives *control* over the distribution of the sampled values.

[6]All the benchmarks were run on a standard laptop with an Intel Core i7-8665U and 32G of RAM running Ubuntu 20.10 with kernel version 5.8.0-48-generic. We used FLINT version 2.6.3-2 and GMP version 6.2.0

[7]The interquartile range of a set of measures is the difference between the third and the first quartiles. Compared with the value of the median, it gives a rough estimate of the dispersion of the measures.

Note that sampling a uniform prefix of a given length is different from sampling a uniform execution of larger length and truncating it. For instance, the program $P = a^\star + (b+c)^\star$ has three execution prefixes of length 1, namely a, b and c but the probability that an execution of length n has a as a prefix is only $1/(1+2^n)$. The prefix a will thus statistically never appear among executions of large length. This trivial example illustrates that in order to achieve a good coverage on the set of prefixes of a given length of a program, a dedicated sampler is necessary. At the end of this section we will compare our sampler experimentally to another classical random sampling technique, called isotropic sampling, and used for instance in [10].

We start by defining formally the notion of prefix. Then we apply the methodology that we have developed in the previous sections to tackle the problem of the uniform generation of prefixes: specify the objects to be sampled, count them and use the counting information to sample. We also show a quantitative result on the number of prefixes of a program and its relation to the number of full executions.

Definition 11 (execution prefixes)**.** *An execution prefix of an NFJ program P is any, possibly empty, sequence of executions steps starting from P of the form $P \stackrel{a_1}{\Rightarrow} P_1 \stackrel{a_2}{\Rightarrow} P_2 \cdots \stackrel{a_n}{\Rightarrow} P_n$. Note that P_n is not necessarily* nullable *here.*

4.1. Specification of the prefixes

As for the uniform random generation of execution in the previous sections, we start by reformulating the problem into combinatorial terms. Just like we have described a combinatorial specification $S(P)$ of the class of the executions of a program P, we describe here how to compute a combinatorial specification $S_p(P)$ of the class of the execution prefixes of P. Interestingly, this specification can be seen as the specification of the executions of a new program $\mathrm{pref}(P)$ whose *full executions* are in bijections with the *execution prefixes* of P. More eloquently, for each program P, we can define a new program $\mathrm{pref}(P)$ such that $S_p(P) = S(\mathrm{pref}(P))$. The recursive rules used to compute $\mathrm{pref}(P)$ as well as $S_p(P)$ and its generating function are given in Table 8. The combinatorial interpretation of each rule is detailed below.

Program P	Prefix program $\mathrm{pref}(P)$	Specification $S_p(P)$	Generating function $\bar{P}(z)$
0	0	$\mathcal{E}$	1
a	$0 + a$	$\mathcal{E} + \mathcal{Z}$	$1 + z$
$P \parallel Q$	$\mathrm{pref}(P) \parallel \mathrm{pref}(Q)$	$S_p(P) \star S_p(Q)$	$\bar{P}(z) \circledcirc \bar{Q}(z)$
$P + Q$	$\mathrm{pref}(P) + \mathrm{pref}(Q)$	$S_p(P) + (S_p(Q) \setminus \mathcal{E})$	$\bar{P}(z) + \bar{Q}(z) - 1$
$P; Q$	$\mathrm{pref}(P) + (P; \mathrm{pref}(Q))$	$S_p(P) + S(P) \boxstar (S_p(Q) \setminus \mathcal{E})$	$\bar{P}(z) + P(z)(\bar{Q}(z) - 1)$
$P^\star$	$P^\star; \mathrm{pref}(P)$	$\mathcal{E} + S(P^\star) \boxstar (S_p(P) \setminus \mathcal{E})$	$1 + \frac{\bar{P}(z)-1}{1-(P(z)-P(0))}$

Table 8: Recursive rules for the computation of (1) the program $\mathrm{pref}(P)$ whose executions are in bijection with the execution prefixes of P, (2) the specification $S_p(P)$ of the execution prefixes of P and (3) its generating function $\bar{P}(z)$

First of all, remark that our definition of execution prefixes includes the empty prefix, having zero execution steps, as well as all the full executions of the program. So for all P, we must have $\mathcal{E} \subset S_p(P)$ and $S(P) \subset S_p(P)$.

For instance, the program a consisting of only one atomic action has two execution prefixes, the empty prefix and the prefix firing a. Hence, its set of prefixes is modelled by $\mathcal{E} + \mathcal{Z}$ which also corresponds to the executions of the program $0 + a$. The empty program only has the empty prefix.

Another simple case is that of the parallel composition. The execution prefixes of $P \parallel Q$ are exactly all the possible interleavings of a prefix of P and a prefix of Q. Hence, its set of prefixes is $S_p(P) \star S_p(Q)$ which corresponds to the executions of $\mathrm{pref}(P) \parallel \mathrm{pref}(Q)$. Similarly, the prefixes of $P + Q$ are simply the union of the respective prefixes of P and Q. Moreover, the intersection of these two sets always contains exactly one element, the empty prefix. Hence the prefixes of $P + Q$ are unambiguously specified by $S_p(P) + (S_p(Q) \setminus \mathcal{E})$, which corresponds to the executions of $\mathrm{pref}(P) + \mathrm{pref}(Q)$. Recall that combinatorial specifications need to be unambiguous for the symbolic method, that is the set of rules dictating how to compute the generating function of $S_p(P)$, to apply.

The case of the sequential composition is more interesting. We distinguish between the prefixes of executions of $P; Q$ which only fire actions from P, and those which have fired at least one action from Q. Said differently, the former correspond to the prefixes which always use the (Lseq) rule of the semantics to $P; Q$ and the latter correspond to those which use the (Rseq) rule at some point. The prefixes firing only actions from P are specified by $S_p(P)$ and the ones firing at least one element from Q are made of a full execution of P followed by a non-empty prefix from Q, that is $S(P) \boxtimes (S_p(Q) \setminus \mathcal{E})$. Note that it is necessary to only consider non-empty prefixes of Q in this second case, so as to ensure that the two specifications do not overlap. The program $\mathrm{pref}(P) + (P; \mathrm{pref}(Q))$ has the same executions.

Finally, the case of the loop is a generalisation of the above reasoning. All the prefixes of $P^\star$, at the exception of the empty prefix, are made of any sequence of *full* non-empty executions of P, corresponding to full iterations of the loop, followed by a non-empty prefix of P, corresponding to the last, possibly partial, iteration. Note that a sequence of non-empty executions of P is actually an execution of $P^\star$.

With this specification at hand, we can derive two kinds of results on the set of execution prefixes of a program. First, we show that the number of execution prefixes of length n of a program is of the same order as its number of executions of length n (provided it has at least one such execution). This means that prefixes are, in average, shared by many executions. Second, on the algorithmic side, we describe a uniform random sampler of execution prefixes in the same fashion as the random sampler of execution of Section 3.

4.2. Quantitative analysis

The number of execution prefixes of length n of a program is trivially lower-bounded by its number of executions of length n. A natural question to ask is how many more prefixes than executions a program has. In this sub-section we quantify the number of prefixes of programs precisely and we prove that, in most cases, the number of prefixes of length n of a program is asymptotically of the same order as its number of executions of length n. We describe the different possible configurations.

The results of this section build on the following technical result, which states that the generating functions of the executions and of the prefixes have the same asymptotic behaviour.

Theorem 11. *Let P be an NFJ program containing at least one loop. Let $P(z)$ denote its generating function of executions and let $\bar{P}(z)$ denote its generating function of execution prefixes. We have that $P(z)$ and $\bar{P}(z)$ have the same radius of convergence $0 < \rho \leq 1$. Furthermore, there exist three constants $\bar{C} \geq C > 0$ and $\alpha \in \mathbb{N}^*$ such that near ρ we have $P(z) \sim \frac{C}{(\rho - z)^\alpha}$ and $\bar{P}(z) \sim \frac{\bar{C}}{(\rho - z)^\alpha}$.*

Note that a program has an infinite number of executions if and only if it contains at least one loop (the pattern $0^\star$ being forbidden). We thus only reason on programs having an infinite

state-space here. Since the generating function $P(z)$ (resp. $\bar{P}(z)$) is rational, its coefficients can by written as a linear combination of geometric terms of the form r^n, with polynomial coefficients, where r is a pole of $P(z)$ (resp. $\bar{P}(z)$).

$$[z^n]P(z) = \sum_{i=1}^{k} p_i(n) r_i^n \qquad \deg(p_i) = \text{degree of the pole } r_i \text{ in } P(z)$$

$$[z^n]\bar{P}(z) = \sum_{i=1}^{\ell} \bar{p}_i(n) \bar{r}_i^n \qquad \deg(\bar{p}_i) = \text{degree of the pole } \bar{r}_i \text{ in } \bar{P}(z)$$

From Theorem 11, we also know that the dominant terms in both sequences are of the same order $n^{\alpha-1}\rho^{-n}$. So the behaviour of the number of executions *and* and the number of prefixes of length n should be of the order of $n^{\alpha-1}\rho^{-n}$. But, because these functions might have several poles of modulus ρ, there might be periodic compensations in these sequences.

For instance, the program $P = (a \parallel b)^\star$ only has executions of even length so that its number of executions is $p_n = \mathbb{1}_{\{2|n\}}\sqrt{2}^n$. However, its number of prefixes is $\bar{p}_n = \mathbb{1}_{\{2|n\}}\sqrt{2}^n + 2\mathbb{1}_{\{2|n-1\}}\sqrt{2}^{n-1}$, which is thus of the order of p_n only for even value of n. A precise description of the possible behaviours of such sequences is given in [5, Theorem V.3] but we only focus on a corollary here, which is more insightful in our context.

Corollary 1. *Let P be an NFJ program with an infinite state-space (thus with at least one loop), let P_n denote its number of executions of length* at most *n and let $\bar{P}_n$ denote its number of executions prefixes of length* at most *n. There exist a constant $\lambda > 0$ such that we have $P_n \leq \bar{P}_n \leq \lambda P_n$. Moreover, if α and ρ denote the constants from Theorem 11, we have that $P_n = \theta(n^{\alpha-1}\rho^{-n})$ if $\rho > 1$ and $P_n = \theta(n^\alpha)$ otherwise.*

Considering the number of executions of length at most n has advantage of mitigating the periodic effects described above and thus describes more faithfully the growth rate of the state-space. This corollary establishes that, in the sense given above, a program has roughly the same number of execution prefixes as it has executions.

The rest of the sub-section is dedicated to the proof of Theorem 11. The proof of Theorem 11 is done by induction on the syntax of the program and is actually straightforward, except for the case of the coloured product, for which we must establish some analytical properties. Lemma 2 gives a calculus formula for the coloured product of a polynomial with any function and Lemmas 4 and 3 gives formulas for computing the coloured product of any two rational functions. We only characterise the properties of the coloured product over rational functions here since the generating function of the executions and the prefixes of an NFJ program are necessarily rational.

Lemma 2. *Let $A(z) = \sum_{n\geq 0} a_n z^n$ be a formal power series and $k \in \mathbb{N}$, we have $z^k \circledcirc A(z) = \frac{z^k}{k!}\frac{d^k}{dz^k}(z^k A(z))$, which holds both formally and analytically in the domain of convergence of A.*

Proof. By definition we have $z^k \circledcirc A(z) = \sum_{n\geq 0} a_n \binom{n+k}{k} z^{n+k}$. Moreover, we have that $\frac{\mathrm{d}^k}{\mathrm{d}z^k}(z^{n+k}) = (n+k)(n+k-1)\cdots(n+1)z^n = \binom{n+k}{k}k!z^n$, hence the result of the lemma. □

Using Lemma 2, we obtain that if A is a rational function, then $z^k \circledcirc A(z)$ has the same poles as A and these poles have the same degree as in A, incremented by k. In Lemmas 4 and 3 below we give two formulas allowing to compute the product of any two rational functions.

Lemma 3. *Let a be a complex number, we have that* $(1+az)^k \circledcirc \dfrac{1}{1-az} = \dfrac{1}{(1-az)^{k+1}}$.

Before going over the proof of this lemma, observe that this formula has a combinatorial interpretation, in terms of rational languages, when a is an integer. Let Σ be an alphabet with a letters and let $\Sigma_1, \Sigma_2, \ldots, \Sigma_k$ be k distinct copies of Σ. The left-hand-side of the equality given in the lemma is the generating function of the rational language L_1 obtained as the shuffle of $(\epsilon + \Sigma_1).(\epsilon + \Sigma_2)\ldots(\epsilon + \Sigma_k)$ with $\Sigma^\star$. The right-hand-side of this equality is the generating function of the language $L_2 = \Sigma_1^\star.\Sigma_2^\star \ldots \Sigma_k^\star.\Sigma^\star$. The equality of the generating functions is explained by the following bijection between the two languages. A word w in L_1 can be uniquely decomposed as $w = w_{i_1}u_{i_1}w_{i_2}u_{i_2}\cdots w_{i_\ell}u_{i_\ell}w'$ where $0 \leq \ell \leq k$, $w_{i_1}, \ldots, w_{i_\ell}, w' \in \Sigma^\star$ and $u_{i_j} \in \Sigma_{i_j}$ for all j. Since Σ_{i_j} is a copy of Σ, w_{i_j} can be mapped to a unique word w'_{i_j} in Σ_{i_j} and thus w can be mapped to a unique word $w'_{i_1}u_{i_1}w'_{i_2}u_{i_2}\cdots w'_{i_\ell}u_{i_\ell}w'$ in L_2. Furthermore, all the words of L_2 can be obtained in such a way.

The general case is given by a computational proof.

of Lemma 3. By definition we have

$$(1+az)^k \circledcirc \frac{1}{1-az} = \sum_{n\geq 0}\sum_{j=0}^{n}\binom{n}{j}\binom{k}{j}a^j a^{n-j}z^n = \sum_{n\geq 0}\left(\sum_{j=0}^{n}\binom{n}{j}\binom{k}{k-j}\right)(az)^n$$

And by Vandermonde's identity $\sum_{j=0}^{n}\binom{n}{j}\binom{k}{k-j} = \binom{n+k}{k}$, which yields

$$\sum_{n\geq 0}\left(\sum_{j=0}^{n}\binom{n}{j}\binom{k}{k-j}\right)(az)^n = \sum_{n\geq 0}\binom{n+k}{k}(az)^n = \left(\frac{1}{1-az}\right)^{k+1}$$

□

Lemma 3 allows to decompose a pole of any degree as a polynomial and a simple pole. We need one last identity allowing to compute the coloured product of two simple poles.

Lemma 4. *Let a and b be two complex numbers, we have* $\dfrac{1}{1-az} \circledcirc \dfrac{1}{1-bz} = \dfrac{1}{1-(a+b)z}$.

Again, this identity has a simple combinatorial interpretation when the numbers a and b are positive integers. In this case, consider two disjoint alphabets Σ and Σ' of respective cardinality a and b. The left-hand-side of this equality is the generating function of the shuffle of the languages $\Sigma^\star$ and $\Sigma'^\star$ and the right-hand-side is the generating function of the language $(\Sigma \cup \Sigma')^\star$, both languages being trivially equal. The proof in the general case is computational.

Proof of Lemma 4. We have by definition

$$\frac{1}{1-az} \circledcirc \frac{1}{1-bz} = \sum_{n\geq 0}\sum_{k=0}^{n}\binom{n}{k}a^k b^{n-k}z^n = \sum_{n\geq 0}(a+b)^n z^n = \frac{1}{1-(a+b)z}$$

Note that, in particular, $\frac{1}{1-az} \circledcirc \frac{1}{1+az} = 1$. □

Using the identities presented above, one can compute the coloured product of any two poles as follows:

$$\begin{aligned}\left(\frac{1}{1-az}\right)^k \circledcirc \left(\frac{1}{1-bz}\right)^\ell &= \left((1+az)^{k-1} \circledcirc (1+bz)^{\ell-1}\right) \circledcirc \frac{1}{1-(a+b)z} \\ &= \sum_{j=1}^{k+\ell-1} \frac{\lambda_{k,\ell,j}(a,b)}{(1-(a+b)z)^j}\end{aligned} \tag{12}$$

where the $\lambda_{k,\ell,j}(a,b)$ are computable coefficients, in particular, $\lambda_{k,\ell,k+\ell-1}(a,b) = \frac{a^{k-1}b^{\ell-1}}{(a+b)^{k+\ell-2}}\binom{k+\ell-2}{k-1}$.

More generally, using the partial fraction decomposition of two rational functions, one can compute their coloured product using (12). This shows that rational functions are stable by coloured product, thus proving that the generating function of the executions and the generating function of the prefixes of a program are rational.

In the particular case where both functions are generating functions, this also gives us that, if their respective dominant singularities are ρ_1 and ρ_2 of degrees α_1 and α_2, then the dominant singularity of their coloured product is $(\rho_1^{-1} + \rho_2^{-1})^{-1}$ and is of degree $\alpha_1 + \alpha_2 - 1$. We will use this fact in the proof of Theorem 11.

Proof of Theorem 11. We prove by induction that, if P is an NFJ program with an infinite state-space, then its generating function of executions $P(z)$ and its generating function of prefixes $\bar{P}(z)$ have the same radius of convergence ρ and satisfy $\bar{P}(z) \sim \lambda P(z)$, for some constant $\lambda > 0$, when $z \to \rho$. Recall that $P(z)$ and $\bar{P}(z)$ are rational.

Parallel composition. If $P = (Q \parallel R)$ and P has an infinite state-space, then at least one of Q or R has an infinite state-space too. Assume, without loss of generality, that this is the case for Q. Then, by induction hypothesis, $Q(z)$ and $\bar{Q}(z)$ have the same radius of convergence ρ_1 and we have $\bar{Q}(z) \sim \lambda_1 Q(z) \sim \frac{\bar{C}_1}{(1-z/\rho_1)^{\alpha_1}}$, for some positive constants λ_1, $\bar{C}_1$ and α_1 when $z \to \rho_1$.

- If R has a finite state-space, it is easy to see that $R(z)$ and $\bar{R}(z)$ are polynomials and that they have the same degree k. Then, using Lemma 2, one can prove that P has the same poles as Q with the same degrees increased by k. Thus, the radius of convergence of $P(z)$ and $\bar{P}(z)$ is ρ_1 and near ρ_1 we have $\bar{P}(z) \sim \lambda P(z) \sim \frac{\bar{C}}{(1-z/\rho_1)^{\alpha_1+k}}$ for some positive constants λ and $\bar{C}$.

- If R has an infinite state-space too, then by induction hypothesis $R(z)$ and $\bar{R}(z)$ have the same radius of convergence ρ_2 and we have near ρ_2 $\bar{R}(z) \sim \lambda_2 R(z) \sim \frac{\bar{C}_2}{(1-z/\rho_2)^{\alpha_2}}$ for some positive constants λ_2, $\bar{C}_2$ and α_2. By using Lemma 4 and Lemma 3 to compute the partial fraction decomposition of $P(z)$ and $\bar{P}(z)$, we can show that they have the same radius of convergence $\rho = (\rho_1^{-1} + \rho_2^{-1})^{-1}$ and that near ρ we have $\bar{P}(z) = \lambda_1\lambda_2 P(z) \sim \frac{\bar{C}}{(1-z/\rho)^{\alpha_1+\alpha_2-1}}$ for some computable constant $\bar{C}$.

Non-deterministic choice. If $P = (Q + R)$ and P has an infinite state-space, then at least one of Q or R have an infinite state-space too. Similarly as before, we assume without loss of generality that this is the case for Q and we apply the induction hypothesis to Q with the same notations.

- First consider the case where either R has a finite state-space, or R has an infinite state-space but with $\rho_2 > \rho_1$ or with $\rho_1 = \rho_2$ and $\alpha_2 < \alpha_1$. In this case of the radius of convergence of $\bar{P}(z)$ and $P(z)$ is ρ_1 and near ρ_1 we have $\bar{P}(z) \sim \bar{Q}(z) \sim \lambda Q(z) \sim \lambda P(z)$ since $R(z) = o(Q(z))$.

- The symmetric case is similar by commutativity, so it only remains to handle the case where $\rho_1 = \rho_2$ and $\alpha_1 = \alpha_2$. In this case we have $\bar{P}(z) \sim \frac{\bar{C}_1+\bar{C}_2}{(1-z/\rho_1)^{\alpha_1}}$ and $P(z) \sim \frac{\lambda_1^{-1}\bar{C}_1+\lambda_2^{-1}\bar{C}_2}{(1-z/\rho_1)^{\alpha_1}}$, which allows to conclude.

Sequential composition. If $P = (Q; R)$ and P has an infinite state-space, then at least one of Q and R have an infinite state-space too. We have that $\bar{P}(z) = \bar{Q}(z) + Q(z)(\bar{R}(z) - 1)$ and $P(z) = Q(z)R(z)$. Again, we use the same notations as above.

- We first consider the case where Q has an infinite state-space and either the state-space of R is finite or is infinite but is such that $\rho_2 > \rho_1$. Thus, the dominant singularity of $P(z)$ and $\bar{P}(z)$ is ρ_1 and when $z \to \rho_1$ we have $\bar{P}(z) \sim (\lambda_1 + \bar{R}(\rho_1) - 1)Q(z)$ and $P(z) \sim R(\rho_1)Q(z)$. As a consequence $\bar{P}(z) \sim \frac{\lambda_1+\bar{R}(\rho_1)-1}{R(\rho_1)}P(z)$.
- In the symmetric case, that is either the state-space of Q is finite or is infinite but such that $\rho_1 > \rho_2$, we have that the radius of convergence of $\bar{P}(z)$ and $P(z)$ is ρ_2. Besides, near ρ_2 we have $\bar{P}(z) \sim Q(\rho_2)\bar{R}(z) \sim Q(\rho_2)\lambda_2 R(z)$ and $P(z) \sim Q(\rho_2)R(z)$. Thus $\bar{P}(z) \sim \lambda_2 P(z)$.
- Finally, if both Q and R have an infinite state-space and $\rho_1 = \rho_2$, then the radius of convergence of $P(z)$ and $\bar{P}(z)$ is ρ_1 and near ρ_1 we have $\bar{P}(z) \sim Q(z)\bar{R}(z) \sim Q(z)\lambda_2 R(z) \sim \frac{\lambda_1^{-1}\bar{C}_2\bar{C}_1}{(1-z/\rho_1)^{\alpha_1+\alpha_2}}$ and $P(z) \sim Q(z)R(z)$, which allows to conclude.

Loops. If $P = Q^\star$, then we have $P(z) = (1 - (Q(z) - Q(0)))^{-1}$ and there is a unique $\rho > 0$ such that $Q(\rho) - Q(0) = 1$. If Q has an infinite state-space, then we have that ρ is smaller than the radius of convergence of $Q(z)$ (and $\bar{Q}(z)$, by induction hypothesis). In a neighbourhood of ρ we have $Q(z) - Q(0) = 1 - (\rho - z)Q'(\rho) + o(\rho - z)^2$, with $Q'(\rho) > 0$, and thus $P(z) \sim \frac{Q'(\rho)^{-1}}{\rho - z}$. Finally, observe that $\bar{P}(z) = 1 + P(z)(\bar{Q}(z) - 1) \sim P(z)(\bar{Q}(\rho) - 1)$ near ρ.

Base cases. There is nothing to prove for $P = 0$ or $P = a$. Informally, the "real" base cases of this induction, that is the cases where P has an infinite state-space but all its sub-terms have a finite one, is the case where $P = Q^\star$ and $Q \neq 0$ contains no loop. □

4.3. Uniform random sampling of prefixes

We now tackle the problem of sampling a uniform prefix of a given length n, of a given program. We first describe in Algorithm 11 how to compute the generating functions of the prefixes of all the sub-terms of a given NFJ program recursively. As for the generating function of the executions, the algorithm must store each resulting generating function in its corresponding AST node. Moreover, we assume that Algorithm 7 has already been called on a program P before running Algorithm 7 on it, so that the generating functions of the *executions* of the necessary sub-terms of P are available.

Random sampling is then straightforward, the methodology is the same as in the previous section. Algorithm 12 describes a uniform random sampler of prefixes based on the generating functions of prefixes computed in Algorithm 11.

The complexity analysis of Algorithm 12 is the same as that of Algorithm 6 in the previous section, and arrives to the same conclusion. We do not repeat it here.

Algorithm 11 Computation of the generating function of the prefixes of a program up to degree n.

function PREFGFUN(P, n)
  if $P = 0$ **then return** 1
  else if $P = a$ **then return** $1 + z$
  else if $P = Q + R$ **then return** PREFGFUN (Q, n) + PREFGFUN $(R, n) - 1$
  else if $P = Q \parallel R$ **then return** PREFGFUN $(Q, n) \circledcirc$ PREFGFUN (R, n)
  else if $P = Q; R$ **then**
    $q(z) \leftarrow$ GFUN (Q, n) ▷ should have been pre-computed
    return PREFGFUN (Q, n) + $q(z) \cdot$ (PREFGFUN $(R, n) - 1$)
  else if $P = Q^\star$ **then**
    $p(z) \leftarrow$ GFUN (P, n) ▷ should have been pre-computed
    return $1 + p(z) \cdot$ (PREFGFUN $(Q, n) - 1$)

Algorithm 12 Uniform random sampling of prefixes of a given length

Input: An NFJ program P and a length n
Output: A uniform prefix of execution of P of length n

function UNIFPREF(P, n)
  if $n = 0$ **then return** the empty prefix
  else if $P = a$ **then return** a
  else if $P = Q + R$ **then**
    if BERNOULLI $(\frac{\bar{q}_n}{\bar{q}_n + \bar{r}_n})$ **then return** UNIFPREF (Q, n)
    else return UNIFPREF (R, n)
  else if $P = Q \parallel R$ **then**
    draw $k \in [\![0; n]\!]$ with probability $\binom{n}{k} \bar{q}_k \bar{r}_{n-k} / \bar{p}_n$
    return SHUFFLE (UNIFPREF (Q, k), UNIFPREF $(R, n - k)$)
  else if $P = Q; R$ **then**
    if BERNOULLI $(\frac{\bar{q}_n}{\bar{p}_n})$ **then return** UNIFPREF (Q, n)
    else
      draw $k \in [\![0; n]\!]$ with probability $q_k \bar{r}_{n-k} / (p_n - \bar{q}_n)$
      return CONCAT (UNIFEXEC (Q, k), UNIFPREF $(R, n - k)$)
  else if $P = Q^\star$ **then**
    draw $k \in [\![0; n - 1]\!]$ with probability $p_k \bar{q}_{n-k} / p_n$
    return CONCAT (UNIFEXEC (P, k), UNIFPREF $(Q, n - k)$)

Table 9: Quick benchmark of the counting and random sampling functions of execution prefixes

$\lvert P\rvert_c$	n	# prefixes	mem. size	prefGfun	UnifPrefix	IQR
100	500	$1.841 \cdot 2^{1128}$	1.77M	0.025s	0.250ms	0.013ms
100	1000	$1.123 \cdot 2^{2244}$	6.67M	0.087s	0.635ms	0.662ms
500	500	$1.640 \cdot 2^{1611}$	5.75M	0.173s	0.372ms	0.024ms
500	1000	$1.034 \cdot 2^{3124}$	20.90M	1.002s	1.098ms	0.069ms
1000	500	$1.047 \cdot 2^{2462}$	17.84M	0.563s	0.526ms	0.041ms
1000	1000	$1.523 \cdot 2^{4844}$	67.14M	3.223s	1.962ms	0.191ms
2000	500	$1.685 \cdot 2^{2381}$	21.43M	0.673s	0.475ms	0.047ms
2000	1000	$1.098 \cdot 2^{4732}$	79.98M	3.630s	1.155ms	0.038ms

4.4. Experimental study

4.4.1. Performance evaluation

In order to assess experimentally the efficiency of our method, we put into use the algorithms presented here and demonstrate that they can handle systems with a significantly large state space. We generated a few NFJ programs as described in Section 3.4 and conducted a similar experiment on the same machine. We quickly recall the main points of our setup below.

Table 9 reports the runtime of the preprocessing phase (Algorithm 11), the runtime of the random sampler (Algorithm 12) and the number of prefixes of length n for various programs and various values of n. Here again, for the runtime of the counting algorithm, every measurement was performed 7 times and we reported the median of these 7 values. For the random sampler, every measure was performed 101 times and for each one we report the median of these values as well as the interquartile range (IQR)[8], which gives an idea of the dispersion of the measures. We use these metrics rather than the mean and the variance to reduce the importance of extreme values and give a precise idea of what runtime the user should expect when running our sampler. The time reported is the CPU time as measured by C's `clock` function. The state-space column indicates the number of *prefixes* of length n. The mem. size column reports the amount of memory occupied by the generating functions of the executions *and* that of the executions prefixes. Recall that both generating functions are necessary for the random sampling routine.

The take-away of this experiment is that (1) the preprocessing phase can be carried out for systems with a state-space of size $\approx 2^{18000}$ in a time of the order of the minute and that (2) once this is done, sampling a uniform prefix in this set is a matter of a few milliseconds.

4.4.2. Prefix covering

We present another experimentation here that highlights the importance of the uniform distribution for the purpose of state-space exploration. The problem is the following. We consider given an NFJ program and we sample random prefixes of a given length n of this program using two different algorithms:

- our random sampler which is *globally* uniform among all prefixes of length n;

[8]The interquartile range of a set of measures is the difference between the third and the first quartiles. Compared with the value of the median, it gives a rough estimate of the dispersion of the measures.

- a more "naive" sampler that repeatedly generates one execution step uniformly at random among the legal steps, until we get a length n prefix. This strategy is called *locally uniform* or *isotropic*.

The question is: in average, how many random prefixes must be generated in order to discover a given proportion of the possible prefixes? This question actually falls under the scope of the Coupon Collector Problem, which is treated in depth in [28]. Table 10 gives numerical answers for both exploration strategies for a random NFJ program of size 25 and for a target coverage of 20% of the possible prefixes.

Prefix length	1	2	3	4	5
# prefixes	11	18	30	60	128
Isotropic	2.1	4.45	11.17	35.09	$1.28 \cdot 10^{14}$
Uniform	2.1	3.18	6.57	13.26	27.69
Gain	0%	40%	70%	165%	$4.61 \cdot 10^{14}$%

Table 10: Expected number of prefixes to be sampled to discover 20% of the prefixes of a random program of size 25 with either the isotropic or the uniform method

Expectedly the uniform strategy is faster but what is interesting to see is that the speed-up compared to the isotropic method grows extremely fast. The more the state-space grows, the more the uniform approach is unavoidable.

Unfortunately, the formula given in [28] for the isotropic case involves the costly computation of power-sets which makes it impractical to give values for larger programs and prefix lengths. However, these small-size results already establish a clear difference between the two methods. It would be interesting to have theoretical bounds to quantify this explosion or to investigate more efficient ways to compute these values but this falls out of the scope of this article.

Conclusion

In this article, we have presented a framework, based on combinatorial specifications and on analytic combinatorics, allowing to study a class of concurrent programs with non-determinism, loops, and a fork-join style of synchronisation. Using this framework, we obtained two types of results. First, on the analytical side, we established quantitative properties of fork-join programs, related to their number of global choices (in the loop-free fragment) and on their typical number of executions prefixes. Second, and more importantly, we described efficient uniform random samplers of executions and execution prefixes allowing the explore the state-space of programs *without* explicitly constructing it. These algorithms thus provide a tractable way to tackle the state explosion problem.

We believe there are two logical directions to go from there. First, although we have made a significant step forward in terms of expressiveness in this article, work remains to be done to be able to model real-life programs. A direction we wish to explore is to relax the kind of synchronisation we allow to drift away from the fork-join model and handle a larger class of programs. As a second step, we wish to apply the techniques described here and implement a statistical model checker relying on uniform random generation.

References

[1] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, J. Oudinet, S. Peyronnet, Coverage-biased random exploration of large models and application to testing, International Journal on Software Tools for Technology Transfer 14 (1) (2012) 73–93.

[2] R. Grose, S. A. Smolka, Monte carlo model checking., in: TACAS, Vol. 3440, Springer, 2005, pp. 271–286.

[3] O. Bodini, A. Genitrini, F. Peschanski, A have proven in Quantitative Study of Pure Parallel Processes, Electronic Journal of Combinatorics 23 (1) (2016) P1.11, 39 pages.

[4] O. Bodini, A. Genitrini, F. Peschanski, The Combinatorics of Non-determinism, in: IARCS Annual Conference FSTTCS, Vol. 24, 2013, pp. 425–436.

[5] P. Flajolet, R. Sedgewick, Analytic Combinatorics, Cambridge University Press, 2009.

[6] O. Bodini, M. Dien, A. Genitrini, F. Peschanski, The Combinatorics of Barrier Synchronization, in: International Conference Petri Nets, Springer, 2019, pp. 386–405.

[7] D. Krob, J. Mairesse, I. Michos, On the average parallelism in trace monoids, in: 19th Annual STACS, 2002, pp. 477–488.

[8] S. Abbes, J. Mairesse, Uniform generation in trace monoids, in: 40th International Symposium MFCS, 2015, pp. 63–75.

[9] N. Basset, J. Mairesse, M. Soria, Uniform sampling for networks of automata, in: 28th International Conference on Concurrency Theory (CONCUR 2017), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[10] R. Grosu, S. A. Smolka, Monte carlo model checking, in: 11th International Conference TACAS, 2005, pp. 271–286.

[11] J. Oudinet, A. Denise, M.-C. Gaudel, R. Lassaigne, S. Peyronnet, Uniform monte-carlo model checking, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2011, pp. 127–140.

[12] A. Darrasse, K. Panagiotou, O. Roussel, M. Soria, Boltzmann generation for regular languages with shuffle, in: GASCOM 2010 - Conference on random generation of combinatorial structures, Montréal, Canada, 2010.

[13] A. Genitrini, M. Pépin, F. Peschanski, Statistical analysis of non-deterministic fork-join processes, in: International Colloquium on Theoretical Aspects of Computing, Springer, 2020, pp. 83–102.

[14] O. Bodini, M. Dien, A. Genitrini, F. Peschanski, Entropic Uniform Sampling of Linear Extensions in Series-Parallel Posets, in: 12th International Symposium CSR, 2017, pp. 71–84.

[15] G. Polya, R. C. Read, Combinatorial enumeration of groups, graphs, and chemical compounds, Springer Science & Business Media, 2012.

[16] O. Bodini, M. Dien, A. Genitrini, F. Peschanski, The Ordered and Colored Products in Analytic Combinatorics: Application to the Quantitative Study of Synchronizations in Concurrent Processes, in: 14th SIAM Meeting ANALCO, 2017, pp. 16–30.

[17] E. Goto, Monocopy and associative algorithms in an extended lisp, Tech. rep., University of Tokyo (1974).

[18] A. Nijenhuis, H. Wilf, Combinatorial Algorithms: For Computers and Hard Calculators, 2nd Edition, Academic Press, Inc., USA, 1978.

[19] W. Hart, F. Johansson, S. Pancratz, FLINT: Fast Library for Number Theory, version 2.5.2, `http://flintlib.org` (2013).

[20] J. Von Zur Gathen, J. Gerhard, Modern computer algebra, Cambridge university press, 2013.

[21] C. Pivoteau, B. Salvy, M. Soria, Algorithms for combinatorial structures: Well-founded systems and Newton iterations, Journal of Combinatorial Theory, Series A 119 (2012) 1711–1773.

[22] A. Bostan, F. Chyzak, M. Giusti, R. Lebreton, G. Lecerf, B. Salvy, É. Schost, Algorithmes efficaces en calcul formel, Published by the authors, 2017.

[23] A. Nijenhuis, H. S. Wilf, Combinatorial algorithms, Computer science and applied mathematics, Academic Press, New York, NY, 1975, [Second edition (1978) available on the author's website].

[24] P. Flajolet, P. Zimmermann, B. Van Cutsem, A calculus for the random generation of labelled combinatorial structures, Theoretical Computer Science 132 (1-2) (1994) 1–35.

[25] X. Molinero, Ordered generation of classes of combinatorial structures, Ph.D. thesis, Universitat Politècnica de Catalunya (Oct 2005).

[26] C. Martínez, X. Molinero, An experimental study of unranking algorithms, in: C. C. Ribeiro, S. L. Martins (Eds.), Experimental and Efficient Algorithms, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 326–340.

[27] M. Bendkowski, O. Bodini, S. Dovgal, Polynomial tuning of multiparametric combinatorial samplers, in: 2018 Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO), SIAM, 2018, pp. 92–106.

[28] P. Flajolet, D. Gardy, L. Thimonier, Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search, D. A. Math. 39 (3) (1992) 207–229.

[29] M. Drmota, Random Trees: An Interplay between Combinatorics and Probability, Springer-Verlag, 2009.

Appendix A. Breaking symmetries

This appendix contains the proof details of Theorem 3 from Section 2.

Appendix A.1. Asymptotic enumeration

We follow here a similar proof to that of Theorem 2. That is we explicit the behaviour of the generating function $f(z)$ of NFJ programs near its dominant singularity and we deduce the number of programs using the transfer theorem from [5, Thm. VI.3 p. 390]. There is a major difference from Section 2 here though, which is that we do not look for an explicit solution to the functional equations. Instead we use complex analysis to describe the behaviour of the solution near its dominant singularity which is actually enough to get precise asymptotic results.

The translation of specification (5) in terms of generating functions yields the following system by the symbolic method. Note that by symmetry, $\mathcal{F}_+$ and $\mathcal{F}_\parallel$ have the same number of elements of each size so their respective generating functions $f_\parallel$ and f_+ are equal and we only use the former.

$$\begin{aligned} f(z) &= z + f_;(z) + 2f_\parallel(z) \\ f_;(z) &= \frac{1}{1-(f(z)-f_;(z))} - 1 - (f(z) - f_;(z)) \\ f_\parallel(z) &= \exp\left(\sum_{j\geq 1} \frac{f(z^j) - f_\parallel(z^j)}{j}\right) - 1 - (f(z) - f_\parallel(z)) \end{aligned}$$

To simplify this system, we first observe that $f_;$ can be expressed as a function of f only, more precisely $f_;(z) = f(z)^2(1+f(z))^{-1}$. Hence $f_;$ has the same radius of convergence as f. For $f_\parallel$ however no such simplification is possible and we resort to a classical argument for this kind of operators. First note that f and $f_\parallel$ have the same radius of convergence ρ and that $\rho < 1$. This can be proved for instance by observing that $\mathcal{Z} \times \mathcal{F}_; \subset \mathcal{F}_\parallel$ and $(\mathcal{Z} + \mathcal{F}_\parallel)^2 \subset \mathcal{F}_\parallel$ and by enumerating the programs belonging to the subset of $\mathcal{F}$ described by $\mathcal{F}'_; = \mathcal{Z} \times \mathcal{F}'_;$ and $\mathcal{F}_\parallel = (\mathcal{Z} + \mathcal{F}_;)^2$. As a consequence, we have that for all $j > 1$, the radius of convergence of $f(z^j)$ (and thus $f_\parallel$) is at least $\sqrt{\rho} > \rho$. Hence, the exponential in the above formula can be split in two:

$$\begin{aligned} f_\parallel(z) &= e^{f(z)-f_\parallel(z)} \exp^{\zeta_1(z)} -1 - (f(z) - f_\parallel(z)) \\ \text{where} \quad \zeta_1(z) &= \sum_{j\geq 2} \frac{f(z^j) - f_\parallel(z^j)}{j} \quad \text{is analytic in } \{z \mid |z| < \sqrt{\rho}\}. \end{aligned}$$

As a consequence, we get that $f_\parallel(z) = \ln \frac{1}{1+f(z)} + f(z) + \zeta_1(z)$ and finally $f(z) = \zeta_2(z) + \phi(f(z))$ where ϕ is the analytic function defined below and $\zeta_2(z) = z + 2\zeta_1(z)$. Note that ϕ has no terms of degree 0 and 1, this will be important in the following.

$$\phi(u) = \frac{u^2}{1+u} + 2\ln\frac{1}{1+u} + u = \sum_{n\geq 2} (-1)^n \left(1 + \frac{2}{n}\right) u^n$$

In order to get the asymptotic behaviour of f near its singularity, we first study the functional equation $y(x) = x + \phi(y(x))$ which admits a unique analytic solution in a neighbourhood of 0. By

unicity of the solution, we get that $f(z) = y(\zeta_2(z))$ and that the radius of convergence ρ of f is the unique $z > 0$ such that $\zeta_2(z)$ is equal to the radius of convergence of y.

We solve the equation $y = x + \phi(y)$ by applying the Theorem 2.19 from [29] after a simple change of variable so that it fits the hypotheses of the Theorem. Note that we actually need the weaker version of the theorem exposed in Remark 2.20 from the same book because our function ϕ has negative coefficients in its expansion. By introducing $\tilde{y} = \frac{y}{x} - 1$ we can reformulate the equation as in:

$$\tilde{y} = \frac{1}{x}\phi(x(1+\tilde{y})) = F(x, \tilde{y}) \tag{A.1}$$

The hypotheses of the Remark 2.20 of [29] apply to this equation. In particular we find that $x_0 = y_0 - \phi(y_0)$ and $\tilde{y}_0 = \frac{\phi(y_0)}{x_0}$ where $y_0 = \frac{\sqrt{3}-1}{2}$ is the unique solution of $\partial_{\tilde{y}} F(x, \tilde{y}) = 1$ on the positive real axis. Hence, by [29] there exists a unique solution $\tilde{y}$ to this equation, it is analytic for $|x| < x_0$ and furthermore there exists two functions $\tilde{h_1}$ and $\tilde{h_2}$ which are analytic around $x = x_0$ and such that locally around $x = x_0$ we have:

$$\tilde{y}(x) = \tilde{h_1}(x) - \tilde{h_2}(x)\sqrt{1 - \frac{x}{x_0}}$$

$$\text{besides,} \quad \tilde{h_1}(x_0) = \tilde{y}_0 \quad \text{and} \quad \tilde{h_2}(x_0) = \sqrt{\frac{2x_0 \partial_x F(x_0, \tilde{y}_0)}{\partial_{\tilde{y}^2} F(x_0, \tilde{y}_0)}} = \sqrt{\frac{2(y_0\phi'(y_0) - \phi(y_0))}{x_0^2 \phi''(y_0)}}.$$

As a consequence f is analytic in $|\zeta_2(z)| < x_0$, its dominant singularity ρ is the unique $z > 0$ such that $\zeta_2(z) = x_0$ and there exists two analytic functions h_1 and h_2 such that locally around $z = \rho$ we have

$$f(z) = h_1(z) - h_2(z)\sqrt{1 - \frac{z}{\rho}} \tag{A.2}$$

$$= y_0 - x_0 \tilde{h_2}(x_0)\sqrt{\frac{\rho \zeta_2'(\rho)}{x_0}}\sqrt{1 - \frac{z}{\rho}} + O(\rho - x). \tag{A.3}$$

Finally, the transfer theorem (see [5, Thm. VI.3 p. 390]) allows us to deduce the asymptotic behaviour of the sequence f_n, counting the number of NFJ programs of size n, when n tends to the infinity.

Appendix A.2. Numerical estimation of the constants

Some arguments exposed above are not constructive. In particular the constant ρ is defined as the solution of an equation involving $f(z^j)$ and $f_{\|}(z^j)$ for all $j \geq 2$, for which we have no expression. In addition, the value of γ depends on the value of ρ and $\zeta_2'(\rho)$, this function being implicitly defined. It is however possible to numerically evaluate these constants with extremely good precision using a method explained in [5, Section VII.5], which we detail in this section.

The method is actually based on a simple idea. One start by computing the first terms of the expansion in power series of f and $f_{\|}$ up to some degree m, this yields two polynomials $f^{[m]}$ and $f_{\|}^{[m]}$ of degree m. Then one uses this expansion to approximate ζ_2 by $\zeta_2^{[m]}(z) = z + 2\sum_{j=2}^{m} j^{-1}(f^{[m]}(z^j) - f_{\|}^{[m]}(z^j))$ and we solve numerically the equation $\zeta_2^{[m]}(z) = x_0 = \ln\left(\frac{2+\sqrt{3}}{2}\right) - \frac{3\sqrt{3}-5}{2}$. Note that the function $\zeta_2^{[m]}$ in increasing on the positive real axis so the equation $\zeta^{[m]}(z) = x_0$ can be numerically

solved by dichotomy. Finally, we let m grow until the approximation stabilises. Since the solution ρ of $\zeta_2(z) = x_0$ lies inside the disc of convergence of ζ_2, the approximation of $\zeta_2(z)$ by $\zeta_2^{[m]}(z)$ converges quickly and the approximation of ρ is extremely precise even for small values of m.

There is one subtlety though in the computation of the first terms of the expansion of $f_{\|}$ and $f_{;}$. We did not give a formula for computing the first terms of $\textsc{MSet}(\mathcal{A})$ given the first terms of $\mathcal{A}$. The usual approach to obtain such a relation is to exploit the formula for the derivative of the generating function of $\textsc{MSet}(\mathcal{A})$ which has a more convenient expression. In our case, we want to the first terms of $f_{\|}(z) = \exp(f(z) - f_{\|}(z) + \zeta_1(z)) - 1 - (f(z) - f_{\|}(z))$ whose derivative is:

$$\begin{aligned} f'_{\|} &= (f' - f'_{\|} + \zeta'_1)\exp(f - f_{\|} + \zeta_1) - (f' - f'_{\|}) \\ &= (f' - f'_{\|} + \zeta'_1)(f + 1) - (f' - f'_{\|}) \\ &= (f' - f'_{\|} + \zeta'_1) \cdot f + \zeta'_1 \\ \text{with} \quad \zeta'_1(z) &= \sum_{j \geq 2} z^{j-1}(f'(z^j) - f'_{\|}(z^j)) \end{aligned}$$

One can then obtain a recurrence relation for the coefficient of degree n of $f'_{\|}$ by extracting the term of degree n from both sides of the last equality. Since $f(0) = 0$ and since the sum starts at $j = 2$ in ζ'_1, only coefficients of f' and $f'_{\|}$ of degree less than n appear on the right-hand-side of the relation.

Getting a recurrence relation to compute the n-th term of $f_{;}$ is more straightforward as there is no exponential to deal with. One can for instance use the following formula (note that both operands of the product are null at 0):

$$f_{;}(z) = f(z)(f(z) - f_{;}(z))$$

Using the approach described above, one quickly gets a good approximation of ρ and $\zeta'_2(\rho)$, this second value being necessary to compute γ. As a rough indication of the speed of convergence of this approximation scheme, with our implementation using double precision floats, the sequence $m \mapsto \rho^{[m]}$ is stationary after $m = 16$ as we hit the limit of representatble numbers.

Appendix A.3. Counting global choices

In terms of generating functions, the specifications (6) and (8) of the *annotated* NFJ programs translate into the following system of equations. Note that the equations satisfied by $g_{;}$ and $g_{\|}$ are similar to the equations satisfied by $f_{;}$ and $f_{\|}$ so we do not repeat the explanations.

$$\begin{aligned} g(z) &= z + g_{;}(z) + g_{\|}(z) + g_{+}(z) \\ g_{;}(z) &= \frac{g(z) - g_{;}(z)}{1 - (g(z) - g_{;}(z))} = \frac{g(z)^2}{1 + g(z)} \\ g_{\|}(z) &= \exp\left(\sum_{j \geq 1} \frac{(g - g_{\|})(z^j)}{j}\right) - 1 - (g - g_{\|})(z) = g(z) + \ln\frac{1}{1 + g(z)} + \sum_{j \geq 2} \frac{(g - g_{\|})(z^j)}{j} \\ g_{+}(z) &= \left(\sum_{j \geq 1} g(z^j) - g_{+}(z^j)\right)(1 + f(z)) - (g(z) - g_{+}(z)) \end{aligned}$$

Let ρ_g denote the radius of convergence of g. Each program has at least one global choice so there is at least as many element of size n in $\mathcal{G}$ as in $\mathcal{F}$ and thus $\rho_g \leq \rho$. Using the same argument as above, one can show that $g(z^j)$ is analytic in a disc of radius larger than ρ_g whenever $j \geq 2$. Hence, the rightmost term in the expression of $g_{\|}$, which we denote by $\zeta_3(z) = \sum_{j\geq 2} \frac{(g-g_{\|})(z^j)}{j}$, is analytic in a disc of radius larger than ρ_g. For the same reasons, $\zeta_4(z) = \sum_{j\geq 2} g(z^j) - g_+(z^j)$ is analytic in the same disc and we can write $g_+ = \zeta_4 + \frac{f}{1+f} g$.

Finally, by merging all equations together, we get

$$\begin{aligned} g(z) &= \zeta_5(z) + (1+f(z))\psi(g(z)) \\ \text{with} \quad \psi(z) &= \frac{z^2}{1+z} + z + \ln\frac{1}{1+z} = \sum_{n\geq 2} \frac{n+1}{n}(-z)^n \\ \text{and} \quad \zeta_5(z) &= (1+f(z))(z+\zeta_3(z)+\zeta_4(z)). \end{aligned}$$

As for f, the key to get the precise behaviour of g near its main singularity, and therefore to get an approximation scheme for ρ_g, is to show that the functional equation $y = y(x,u) = x + u\psi(y)$ has a unique solution, which we are able to describe, and to conclude by unicity that $g(z) = y(\zeta_5(z), 1+f(z))$. To this end, we use an extension of Theorem 2.21 from [29]. This requires to apply the simple change of variables $\tilde{y} = \frac{y-x}{ux}$ and $\tilde{u} = \frac{u}{1+f(\rho_g)}$ so that the equation fulfils the requirements of the theorem (except for the non-negativity of the coefficients of ψ). The equation then becomes:

$$\tilde{y} = F(x,\tilde{y},\tilde{u}) = \frac{1}{x}\psi(x(u_1\tilde{u}\tilde{y}+1)) \quad \text{where } u_1 = 1+f(\rho_g). \tag{A.4}$$

Remark 2.20 from [29] (which is related to Theorem 2.19 in the book) can actually be adapted to Theorem 2.21. So in order to prove the existence of an analytic continuation of "square-root" type for the unique solution of (A.4), it is enough to show that there exists a pair $(x_1, \tilde{y_1})$ in the domain of convergence of F such that

$$\begin{aligned} \tilde{y_1} &= F(x_1, \tilde{y_1}, 1) \\ 1 &= \partial_{\tilde{y}} F(x_1, \tilde{y_1}, 1) \\ 0 &\neq \partial_x F(x_1, \tilde{y_1}, 1) \\ 0 &\neq \partial^2_{\tilde{y}} F(x_1, \tilde{y_1}, 1) \end{aligned}$$

In our case, we have $\partial_{\tilde{y}} F(x,\tilde{y},\tilde{u}) = u_1\tilde{u}\psi'(x(u_1\tilde{u}\tilde{y}+1))$ and $\psi'(z) = 2 - \frac{1}{1+z} - \frac{1}{(1+z)^2}$. First we solve $u\psi'(z) = 1$ which yields a unique positive solution $y_1(u) = \frac{2}{\sqrt{1+4(2-u^{-1})}-1} - 1$ which is a decreasing function of u and thus satisfies $y_1(1) > y_1(u) \geq y_1(u_1) \geq y_1(1+f(\rho))$ for $1 < u \leq u_1$. Note that we have $y_1(1) = \frac{\sqrt{5}-1}{2} \approx 0.618$ and $y_1(1+f(\rho)) = y_1(\frac{1+\sqrt{3}}{2}) \approx 0.366$.

Thus, a solution $(x_1, \tilde{y_1})$ of the above system necessarily satisfies $x_1(u_1\tilde{y_1}+1) = y_1(u_1)$. Then we solve $\tilde{y_1} = F(x_1, \tilde{y_1}, \tilde{u})$ by injecting the later equality in the definition of F which yields $x_1(u) = y_1(u) - u\psi(y_1(u))$ in the general case and thus $x_1 = y_1 - u_1\psi(y_1(u_1))$. The two non-nullity conditions are then easily checked since $\partial_x F(x_1, y_1, 1) = (u_1 x_1)^{-1}$ and $\partial^2_y F(x_1, y_1, 1) = x_1 u_1^2 \psi''(y_1(u_1)) > 0$.

We thus obtain that equation (A.4) has a unique solution $\tilde{y}$ which is analytic near $x = 0$ and $u = 1$. Furthermore, this functions admits a representation of the following form near $x = x_1$ and $u = 1$ where $\tilde{h_3}$ and $\tilde{h_4}$ are analytic functions:

$$\tilde{y}(x,\tilde{u}) = \tilde{h_3}(x,\tilde{u}) - \tilde{h_4}(x,\tilde{u})\sqrt{1 - \frac{x}{x_1(u_1\tilde{u})}}$$

From the unicity of the solution, we get that $g(z) = y(\zeta_5(z), 1+f(z))$ and thus, for some analytic functions $h_3(z)$ and $h_4(z)$, we have:

$$g(z) = h_3(z) - h_4(z)\sqrt{1 - \frac{\zeta_5(z)}{x_1(1+f(z))}} \tag{A.5}$$

The singularity ρ_g of g is thus the minimum positive real number in $[0;\rho]$ such that $\zeta_5(z) = x_1(1+f(z))$. Numerically we get that $\rho_g \approx 0.12 < \rho$ so in a neighbourhood of ρ_g we have

$$g(z) = h_3(z) - h_5(z)\sqrt{1 - \frac{z}{\rho_g}}$$

$$\text{where} \quad h_5(z) = h_4(z)\sqrt{\left(1 - \frac{\zeta_5(z)}{x_1(1+f(z))}\right)\left(1 - \frac{z}{\rho_g}\right)^{-1}} \quad \text{is analytic near } \rho_g.$$

As a consequence, the number g_n of annotated programs is equivalent to $\gamma n^{-\frac{3}{2}}\rho_g^{-n}$ for some constant $\gamma_g > 0$. The numerical evaluation of the constants ρ_g and γ_g is similar to the evaluation of ρ and γ so it is not repeated here. Numerically, we get $\rho_g \approx 0.122854753$.