Analysis of functions with a finite number of return values

Paul Zimmermann

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N°1625

Programme 2

Calcul symbolique, Programmation
et Génie logiciel

ANALYSIS OF FUNCTIONS WITH A FINITE NUMBER OF RETURN VALUES

Paul ZIMMERMANN

Février 1992

Analysis of functions with a finite number of return values

Paul Zimmermann [1]

Abstract. A class of functions with a finite number of return values is defined over combinatorial structures. We prove that the characterization of combinatorial structures by such a function is equivalent to a set of grammar productions, and we show how to derive these productions from the body of the function. In the field of automatic average case analysis, this result allows the use of functions, which makes the description of algorithms more natural and smaller.

Analyse de complexité moyenne de fonctions à nombre fini de valeurs

Résumé. On définit une classe de fonctions opérant sur des structures combinatoires, telle que chaque fonction a un ensemble image fini. On démontre que la caractérisation de structures combinatoires par de telles fonctions est équivalente à la donnée de productions d'une grammaire, et on indique comment obtenir ces productions à partir de la définition de la fonction. Dans le cadre de l'analyse en moyenne automatique d'algorithmes, ce résultat autorise l'utilisation de fonctions, ce qui rend la description des algorithmes plus aisée et concise.

[1]INRIA, Rocquencourt, 78153 Le Chesnay (France). This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

Analysis of functions with a finite number of return values

Paul Zimmermann[†]
zimmermann@inria.inria.fr

Abstract

A class of functions with a finite number of return values is defined over combinatorial structures. We prove that the characterization of combinatorial structures by such a function is equivalent to a set of grammar productions, and we show how to derive these productions from the body of the function. In the field of automatic average case analysis, this result allows the use of functions, which makes the description of algorithms more natural and smaller.

Keywords: combinatorial structures, automatic average case analysis, generating functions.

1 Introduction

Since Wegbreit's paper [11], different authors tried to formalize the analysis of algorithms: Ramshaw in his PhD Thesis [9], Cohen and Hickey [2, 7]. Recent works are more concerned in the design of systems that really perform an *automatic* analysis [8, 13, 5]. In the particular field of *average case* analysis, generating functions have proven to be a very useful tool [6]. Briefly, programs translate into equations for generating functions (Algebraic Analysis [12]) that contain all the required informations for average case analysis; in particular, some well-known analytic theorems give an asymptotic expansion of the average cost directly from the generating functions (Analytic Analysis [10]).

In this paper, we extend the Algebraic Analysis process to a wider class of programs than the one described in [12]. More precisely, we allow the use of (some kind of) functions, and we show that the resulting programs still translate into generating functions. This result is interesting, because it enables to write smaller (and usually more comprehensive) programs to describe the cost of an algorithm, for example in the Lambda-Upsilon-Omega system. Before we reveal the aim of this paper, let us just give a brief description of this system.

Lambda-Upsilon-Omega (or simply $\Lambda\Upsilon^{\Omega}$) is an assistant system, that analyzes in the average case some well-defined classes of algorithms [4, 5]. To describe an algorithm, one writes a program

[†]Domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex. This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

in a special purpose language called ADL (Algorithm Description Language). This program contains declarations for data types, procedures and complexity measures. For example, the set $\mathcal{N}$ of arithmetic expressions constructed with the integers $0, 1$, the binary operators $+, \times$ could be described in the following manner:

```
type N = zero | one | plus N N | times N N;
     zero, one = atom(0);
     plus, times = atom(1);
```

Here, the keyword **type** begins the data type declarations, the first line says that an element of $\mathcal{N}$ is either `zero` or `one`, or the Cartesian product of `plus` and two elements of $\mathcal{N}$, and so on. The second line defines `zero` and `one` as *atoms* (terminals in usual grammars) of size 0. The atoms `plus` and `times` are of size 1. The size is additive, thus the size of an arithmetic expression as defined by the above lines equals the number of atoms `plus` and `times` it contains. For example, the sequence

```
plus times zero times one times plus one one times plus one one one one
```

stands for the expression $(0 \times (1 \times ((1 + 1) \times ((1 + 1) \times 1)))) + 1$, whose size is 7.

To describe the cost of an algorithm, one writes some procedures that recursively go through the data structures. For example, one could count the number of $\times$ symbols that are just followed by a 0 in an arithmetic expression by the following ADL procedure

```
procedure P (n : N);
begin
   case n of
      zero : found_zero;
      one : found_zero;
      plus(i,j) : begin P(i); P(j) end;
      times(zero,j) : begin found_one; P(j) end;
      times(i,j) : begin P(i); P(j) end;
   end
end;
measure found_zero : 0;  found_one : 1;
```

In this procedure, the **case** ... **end** instruction distinguishes between different kinds of expressions, where the pattern `plus(i,j)` filters all sums, `times(zero,j)` filters all products whose first component is 0, and `times(i,j)` filters all remaining products.

Using algebraic analysis rules [12], the system translates the program into generating function[1] equations. These equations are then solved by a computer algebra system and we get:

$$N(z) = \frac{1 - \sqrt{1 - 16z}}{4z}, \quad \tau P(z) = \frac{1 - \sqrt{1 - 16z}}{4\sqrt{1 - 16z}}.$$

Using singularity analysis [10], the Analytic Analyzer extracts from these generating functions an asymptotic expansion of the average cost, here $n/8 + O(\sqrt{n})$ for expressions of size n. The interested reader will find a more complete introduction to the $\Lambda\Upsilon\Omega$ system in [5].

The capabilities of the ADL language, which is extensively described in [12, section 1.6], are powerful enough to describe rather complex problems, like Banach's famous matchbox

[1] If $\mathcal{A}$ is a set of combinatorial structures, the (counting) generating function of $\mathcal{A}$ is $A(z) = \sum_{a \in \mathcal{A}} z^{|a|}$, where $|\cdot|$ denotes the size function. If P is a procedure that takes inputs in $\mathcal{A}$, the generating function (of cost) associated to P is $\tau P(z) = \sum_{a \in \mathcal{A}} \tau P\{a\} z^{|a|}$, where $\tau P\{a\}$ is the cost of the evaluation of P on a. Thus the average cost of P over inputs of size n is simply $\tau P_n / A_n$ where f_n denotes the coefficient of z^n in the Taylor expansion of f around $z = 0$.

problem [4, Report 16]. But in some cases, the problem description becomes very intricate in ADL, although its formulation in an usual computer language is easy. Suppose for example that, among all arithmetic expressions of size n in $\mathcal{N}$, we want to know what proportion have an even value. One (tedious) solution would be to write an ADL grammar for even and odd expressions: an odd expression is for example either 1, or the sum of an even expression and an odd one, or the product of two odd expressions. An easier (and more natural) way would be to write a boolean function `is_even`, and then use this function in other parts of the program, for example **if** `is_even`(n) **then** `count1` **else** `count0`. But the ADL language as described in [12] does not allow any function.

We provide in this paper an extension of the ADL language with some kind of functions that compute properties over data structures. These functions are useful to describe algorithms in a natural manner. In this extension, the Algebraic Analysis is still automatic: the equations for generating functions are computable by a machine. This is a consequence of the Reduction Theorem (Section 3) that proves in addition that the class of equations produced is not modified by this extension. For example, we shall see how to obtain automatically the following result:

The probability P_n that an expression of size n in $\mathcal{N}$ has an even value is

$$P_0 = \frac{1}{2}, \quad P_1 = \frac{5}{8}, \quad P_2 = \frac{21}{32}, \quad P_3 = \frac{429}{640}, \quad \ldots, \quad P_n = \frac{1}{\sqrt{2}} + O(n^{-1/2}).$$

The paper is organized as follows: in Section 2, we define a class of functions with a finite number of return values over combinatorial structures like Cartesian products, lists and sets (Definition 2); in Section 3, we show that every such function translates into characteristic data type specifications (Theorem 1 or Reduction Theorem); in Section 4, we apply the results of Section 3 to the complexity analysis of programs with functions (Theorem 2) and we provide two examples.

2 Definitions

By specifying their grammar productions, we now define a class Ω of data structures (Section 2.1) and a class Π of functions on these structures (Section 2.2).

2.1 The class Ω

We consider the class Ω of grammars with productions among the three following classical kinds, where upper-case letters (A, B, C, ...) stand for non-terminals, and lower-case letters (a, b, c, ...) for terminals:

$A \to a$,

$A \to B \mid C$,

$A \to B \times C$,

and other productions that produce respectively lists, sets and multisets, that is sets with repetitions (the +-form of each constructor does not produce the empty structure):

$A \to \mathit{sequence}(B)$, $A \to \mathit{sequence}^+(B)$,

$A \to \mathit{set}(B)$, $A \to \mathit{set}^+(B)$,

$A \to \mathit{multiset}(B)$, $A \to \mathit{multiset}^+(B)$.

Definition 1 *We call Ω the class of grammars whose productions use union ($|$), product ($\times$), and the constructors sequence, sequence$^+$, set, set$^+$, multiset, multiset$^+$.*

Every such grammar derives a set of objects (even if these objects are no longer words, because of the set productions, we keep the word "language" for a set of objects). For example, the grammar

$$A \to a, \quad B \to b, \quad C \to A \mid B, \quad S \to \mathit{multiset}(C),$$

belongs to Ω, and the non-terminal S derives all multisets with letters a and b, including the empty multiset $\{\}$.

2.2 The class Π of functions on Ω

We now define a class, that we call Π, of programs on Ω. A program is a set of definitions of functions on Ω. These functions are constructed by means of *programming schemes*. We distinguish between two kinds of programming schemes: *general* schemes do not look at the inner object structure, whereas *descending* schemes depend on the object structure (product, set, sequence, ...).

2.2.1 General schemes

Our notation for a function definition is the following:

$$\langle\text{function name}\rangle(\langle\text{argument name}\rangle : \langle\text{argument type}\rangle) := \langle\text{function body}\rangle.$$

In the following lines, the names f, g, h denote functions of the class Π that is currently being defined. The first general scheme is the *return* scheme:

Scheme 1 (return) $f(a : A) := \langle \mathit{value} \rangle$,

that defines a constant function over inputs of type A. To combine the results of different functions on the same input, we use the *combine* scheme:

Scheme 2 (combine) $f(a : A) := g(a) \diamond h(a)$

where $\diamond$ stands for a binary operator between finite sets. This operator could be one of the logical operators *and*, *or* in the case where f, g, h return boolean values, that is true or false.

2.2.2 Descending schemes

Combining the general schemes defined above, we are only able to construct constant functions, because we have no branching scheme. The following schemes that depend on the data structure enable us to define more interesting functions.

Scheme 3 (descent in union)

$$A \to B \mid C \qquad f(a:A) := \textbf{ if } a \in B \textbf{ then } g(a) \textbf{ else } h(a)$$

This scheme enables us to return a different result according to the type of the function argument. If one compares the *union* constructor (|) to the record with variant declaration of the PASCAL language, then the *descent in union* scheme becomes very similar to a test of the record selector:

type $A =$ **record case** sel : *boolean* **of**
$true : (b : B);$
$false : (c : C);$
end ;

$A \to B \mid C$

if $a.sel = true$ **then** ... **else** ... **if** $a \in B$ **then** ... **else** ...

When a data structure is the product of several sub-objects, the following scheme enables one to define functions depending on one component only:

Scheme 4 (descent in product) $A \to B \times C \qquad f((b,c):A) := g(b).$

The last schemes to be introduced test whether a given property is true for all components (**forall**) or at least for one (**forone**) in a list, set or multiset, denoted below by the generic constructor Φ. Therefore, these schemes are available only for boolean functions.

Scheme 5 (descent in list, set, multiset [forall])

$$A \to \Phi(B) \qquad f(a:A) := \textbf{ forall } b \textbf{ in } a \textbf{ do } g(b)$$

Scheme 6 (descent in list, set, multiset [forone])

$$A \to \Phi(B) \qquad f(a:A) := \textbf{ forone } b \textbf{ in } a \textbf{ do } g(b)$$

The meaning of the **forall** scheme is: if all components b are such that $g(b) =$ true, or if there is no component, then f returns true, otherwise it returns false. In the case of the **forone** scheme, if $g(b) =$ true for at least one component b, then f returns true, otherwise it returns false; if there is no component, then f returns false.[2]

Definition 2 *A* boolean program *is a self-contained set of function specifications on* Ω *defined with the schemes 1–6 (return, combine, descent in union, product, list, set, multiset); we denote by* Π *the class of boolean programs.*

As every return value is associated to a constant function (by the return scheme), the number of possible return values of a boolean program is less than the number of functions that it contains. Therefore, functions in programs of Π have a finite number of return values.

[2]The **forall** and the **forone** schemes may be considered as two particular instances of a general iteration scheme of the form $d \diamond g(b_1) \diamond g(b_2) \diamond \cdots \diamond g(b_k)$, where d is a default value. With $d =$ true and $\diamond =$ and, we obtain the **forall** scheme; with $d =$ false and $\diamond =$ or, we get the **forone** scheme. With $\diamond =$ xor, we would get a scheme that depends on the parity of the number of b for which $g(b)$ is true.

3 Computing characteristic data type specifications

Consider a function f with input in a set $\mathcal{A}$ of objects corresponding to a non terminal A. If v is a possible value for the function f, let $\mathcal{A}_{f=v}$ be the subset of objects of $\mathcal{A}$ for which f returns v.

Definition 3 *A* characteristic data type specification *of f is a set of grammar specifications of the subsets $\mathcal{A}_{f=v}$ for all possible return values v for f.*

In this section, we prove that programs of Π have characteristic data type specifications in the class Ω, and how to compute them from function definitions and initial data type specifications.

Theorem 1 (Reduction Theorem) *Every program of* Π *translates into characteristic data type specifications for its functions. Moreover, these specifications are in the class* Ω.

Let us now prove Theorem 1: The proof divides into two steps: first, every function of Π translates into some specification in a class $\Omega_\cap$ (Lemma 1); secondly, these specifications of $\Omega_\cap$ are reduced to specifications of Ω (Lemma 2).

Let $\Omega_\cap$ be the class of specifications made from the constructors of Ω with in addition the *intersection* constructor $\cap$, such that $A \cap B$ derives the data structures derived by A *and* by B.

Lemma 1 *Every program of* Π *translates into data type specifications in* $\Omega_\cap$.

Proof: The lemma results from the following rules that compute from a function its characteristic data type specification (for the sake of simplicity, the letter v denotes a generic value of the function f, and the letter w denotes another value, *different* from v):

Rule 1 (return)

$$\frac{f(a:A) := v}{A_{f=v} \to A, \qquad A_{f=w} \to \emptyset}$$

Rule 2 (combine)

$$\frac{f(a:A) := g(a) \diamond h(a)}{A_{f=v} \to \bigcup_{\substack{v_1, v_2 \\ v_1 \diamond v_2 = v}} A_{g=v_1} \cap A_{h=v_2}}$$

Rule 3 (descent in union)

$$\frac{A \to B \mid C \qquad f(a:A) := \textbf{ if } a \in B \textbf{ then } g(a) \textbf{ else } h(a)}{A_{f=v} \to B_{g=v} \mid C_{h=v}}$$

Rule 4 (descent in product)

$$\frac{A \to B \times C \qquad f((b,c):A) := g(b)}{A_{f=v} \to B_{g=v} \times C}$$

Rule 5 (forall on a sequence)

$$\frac{A \to sequence(B) \qquad f(a:A) := \textbf{ forall } b \textbf{ in } a \textbf{ do } g(b)}{\begin{array}{c} A_{f=\text{true}} \to sequence(B_{g=\text{true}}) \\ A_{f=\text{false}} \to sequence(B_{g=\text{true}}) \times B_{g=\text{false}} \times sequence(B) \end{array}}$$

Rule 6 (forall on a set)

$$\frac{A \to set(B) \qquad f(a:A) := \textbf{ forall } b \textbf{ in } a \textbf{ do } g(b)}{\begin{array}{c} A_{f=\text{true}} \to set(B_{g=\text{true}}) \\ A_{f=\text{false}} \to set(B_{g=\text{true}}) \times set^{+}(B_{g=\text{false}}) \end{array}}$$

The remaining rules are very similar: for the +-form of the *set* constructor, just replace *set* by set^+ in the production of $A_{f=\text{true}}$; for the *multiset* constructor, just replace *set* by *multiset* in the rules; and for the **forone** scheme, just exchange *true* and *false* in rules 5 and 6.

Once put in Chomsky normal form, all rules give productions of Ω, except rule 2 that produces an intersection, therefore the whole specification belongs to $\Omega_\cap$. ■

Lemma 2 *Each data type specification of $\Omega_\cap$ produced by the preceding rules reduces to an equivalent (producing the same data structures) specification of Ω.*

Proof: [sketch] This lemma is proven by the existence of an algorithm, called **Reduction**, that transforms specifications of $\Omega_\cap$ into specifications of Ω. This algorithm is detailed in [12] in the case of a similar class of functions (schemes 5 and 6 were not allowed).

The main ideas of this algorithm are the following: first, for each new intersection $U_1 \cap U_2$ to be computed, the non-terminals U_1 and U_2 have necessarily a common *ancestor-type*, that is a non-terminal U of the initial specification that derives at least all data structures derived by U_1 and U_2. The existence of this ancestor-type is proven by induction from rule 2. Secondly, the productions that define U_1 and U_2 are necessarily of the same type (terminal, union, product, set, ...). Thus it is possible to write a production for $U_1 \cap U_2$ in terms of intersection of non-terminals appearing in the right hand side of the productions of U_1 and U_2. Then one replaces everywhere $U_1 \cap U_2$ by a new non-terminal, and one proceeds. The algorithm halts because the number of intersections to be computed is bounded. One then obtains a specification without any intersection. ■

4 Application to complexity analysis

In this section, we apply our results to the complexity analysis of computer programs. On the one hand, Theorem 1 implies that all programs containing functions of the class Π could be written without functions, thus functions are not useful. On the other hand, the same theorem implies that every program with functions of Π could be rewritten *automatically* into a program without any function, for which several methods of complexity analysis are known (see for example [5]). Thus the rules of Section 3 are useful because functions help to write more understandable and usually much smaller programs.

In a program description, we use functions by means of the *conditional scheme*:

Scheme 7 (conditional)

$$P(a:A) := \textbf{ if } f(a) = v \textbf{ then } Q(a) \textbf{ else } R(a),$$

where f is a function of Π, and P, Q, R are procedures. When the value of f on a is v, $P(a)$ calls $Q(a)$, otherwise $R(a)$.

Theorem 2 (Automatic analysis) *The Algebraic Analysis of* ADL *programs with functions in the class* Π *and the conditional scheme is possible automatically (*id est *the equations satisfied by generating functions are automatically computable), and the corresponding average case costs are computable in polynomial time.*

This last theorem is in fact a corollary of Theorem 1: the first assertion is a direct consequence of the reduction property and of the algebraic analysis of programs without functions on Ω [12, theorem 4]; the second assertion comes from theorems 5 and 6 of [12].

We said in the introduction that the description of algorithms with functions is more natural and easier. As the following result proves it, the use of functions may also produce much smaller programs:

Theorem 3 *Using functions decreases exponentially the program size in some cases.*

Proof: Consider a program that takes as input an integer in unary notation

type integer = one | one integer;

and has cost 1 if this integer is divisible by some number p, and 0 otherwise. Without functions, we write p mutually recursive procedures $Q_0, Q_1, \ldots, Q_{p-1}$, where $Q_k(\texttt{one})$ has cost 1 if $k = p-1$, 0 otherwise, and $Q_k((\texttt{one}, \texttt{j}))$ calls $Q_{(k+1) \bmod p}(\texttt{j})$. This program, with Q_0 as main procedure, is a solution. Each procedure has a constant length, and there are p procedures, thus the program length is $\Omega(p)$. In the same manner, we could obtain a set of boolean functions of total length $O(p)$ that recognizes multiples of p.

Suppose now that p is the product of the k first prime numbers $p_1, \ldots, p_k$. Using functions, we would write a set of functions for each prime, where one function, say M_j, recognizes multiples of p_j, and a main function whose body would simply be

$$M_1(i) \text{ and } M_2(i) \text{ and } \ldots \text{ and } M_k(i).$$

This program would have a length of $O(p_1 + \cdots + p_k + k) = O(kp_k)$, which is exponentially smaller that $\Omega(p_1 \ldots p_k)$ without functions. ■

Rules 1 to 6 have already been included in an experimental version of the $\Lambda\Upsilon^{\Omega}$ system. Thus in the following examples, the complexity analysis is done automatically by a computer (namely a Sun 3/60).

4.1 Parity of arithmetic expressions

The purpose of this example is to show how characteristic data types specifications are automatically deduced from function definitions, following the rules of Lemma 1 and the algorithm **Reduction** of Lemma 2.

We consider the set $\mathcal{N}$ of arithmetic expressions defined in the introduction, and we are interested in the probability of an expression of size n (with n symbols $+$ or $\times$) to have an even value. For example, there are eight expressions of size 1,

$$0+0, \quad 0+1, \quad 1+0, \quad 1+1, \quad 0\times 0, \quad 0\times 1, \quad 1\times 0, \quad 1\times 1,$$

and five of them have an even value. Thus the probability for $n = 1$ is $P_1 = 5/8$. Now let us write a boolean function that takes as input an arithmetic expression, and decides recursively whether it is even or not:

```
function is_even (n : N) : boolean;
begin
   case n of
      zero : true;
      one : false;
      plus(i,j) : if is_even(i) then is_even(j)
                  else if is_even(j) then false else true;
      times(i,j): if is_even(i) then true else is_even(j)
   end
end;
```

Using the conditional scheme, we then write the following procedure that calls a dummy instruction `count` of cost 1 for each even expression. The average cost of this procedure is thus exactly the probability wanted.

```
procedure proba_is_even (n : N);
begin
   if is_even(n) then count
end;
measure count : 1;
```

When we analyze this program with the $\Lambda\Upsilon^{\Omega}$ system, we get the following answer:

```
% luo V1.4
Lambda-Upsilon-Omega V1.4 (Caml+Maple)

#printlevel:=2; analyze "parity";;      (* we only type this *)

Introducing the new type N_is_even
Introducing the new type N_not_is_even

Counting generating functions:
  N_is_even(z)=zero(z)+plus(z)*N_is_even(z)**2+plus(z)*N_not_is_even(z)**2
    +times(z)*N_is_even(z)*N(z)+times(z)*N_not_is_even(z)*N_is_even(z)
  N_not_is_even(z)=one(z)+2*plus(z)*N_is_even(z)*N_not_is_even(z)+times(z)
    *N_not_is_even(z)**2
```

These last two equations, that are in fact the algebraic translation of grammar productions generated by the system for even and odd expressions, enable us to find the following generating function of cost for the procedure `is_even`:

$$\tau P(z) = \frac{2-\sqrt{2}\ \sqrt{1+\sqrt{1-16z}}}{4z},$$

and the asymptotic expansion $P_n = 1/\sqrt{2} + O(1/\sqrt{n})$ follows by means of well-known methods of singularity analysis presented in [5, 3] and implemented in a computer algebra system by B. Salvy [10].

4.2 Partitions with odd summands

This other example illustrates the use of the **forall** and **forone** schemes for functions over lists, sets or multisets. The following Adl program that belongs to the class Π determines whether an integer partition contains only odd summands or not:

```
type partition = multiset(integer);
     integer = one | product(one,integer);
     one = atom(1);
function is_odd (i : integer) : boolean;
begin
   case i of
      one      : true;
      (one,j) : if is_odd(j) then false else true;
   end;
end;
function has_only_odd_summands (p : partition) : boolean;
begin
   forall i in p do is_odd(i)
end;
```

The automatic complexity analysis of this program by the $\Lambda\Upsilon\Omega$ system produces the following results:

```
Introducing the new type integer_is_odd
Introducing the new type integer_not_is_odd

Introducing the new type partition_has_only_odd_summands
Introducing the new type partition_not_has_only_odd_summands

Counting generating functions:
  partition_not_has_only_odd_summands(z)=MP(integer_is_odd(z))
    *(MP(integer_not_is_odd(z))-1)
  partition_has_only_odd_summands(z)=MP(integer_is_odd(z))
  integer_is_odd(z)=one(z)+one(z)*integer_not_is_odd(z)
  integer_not_is_odd(z)=one(z)*integer_is_odd(z)
```

$$\texttt{tau\_proba\_has\_only\_odd\_summands(z) = MP(- --------)} \quad \frac{z}{2 - 1 + z}$$

where MP is the operator associated to the *multiset* constructor: $\mathrm{MP}(\sum a_n z^n) = \prod \frac{1}{(1-z^n)^{a_n}}$.

5 Conclusion

In this paper, we have presented a model of combinatorial specifications with functions (Section 2) that is completely reducible to a simpler model (Theorem 1), which in turn is well-suited for automatic complexity analysis [5]. Hence we have not enlarged the class of complexity results that a computer is able to produce, but we provide new tools to make the program description easier and smaller, as shown by the examples of Section 4 and Theorem 3.

Our result is very similar to the well-known closure property of context-free languages under the action of finite automata [1]: the class Ω introduced here describes context-free languages extended with set constructors (*set, multiset*), and functions of Π behave like automata without memory. More precisely, the functions of Π are tree-automata over the derivation trees of data structures. Because of this analogy, we say that functions of Π compute *regular* properties over the combinatorial data structures of Ω.

In some sense, Π is the largest class that preserves the closure property of Theorem 1, with respect to the class Ω. Namely, if we allow an integer counter per function, then we will be able to write a function f that recognizes words of the form $a^n b^n c^i$, a function g that recognizes words of the form $a^i b^n c^n$, whence f and g will recognize the well-known language $\{a^n b^n c^n\}$, which does not belong to the class of languages produced by a specification of Ω.

Acknowledgement: I would like to thank Philippe Flajolet for his many fruitful suggestions concerning this paper.

References

[1] N. Chomsky and M. P. Schützenberger. The algebraic theory of context–free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programing and Formal Languages*, pages 118–161. North Holland, 1963.

[2] Jacques Cohen. Computer–assisted microanalysis of programs. *Communications of the ACM*, 25(10):724–733, 1982.

[3] P. Flajolet and A. M. Odlyzko. Singularity Analysis of Generating Functions. *SIAM Journal on Discrete Mathematics*, 3(2):216–240, 1990.

[4] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda–Upsilon–Omega: The 1989 Cookbook. Rapport de recherche 1073, Institut National de Recherche en Informatique et en Automatique, August 1989. 116 pages.

[5] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic Average–case Analysis of Algorithms. *Theoretical Computer Science*, 79(1):37–109, February 1991.

[6] P. Flajolet and J-M. Steyaert. A complexity calculus for recursive tree algorithms. *Mathematical Systems Theory*, 19:301–331, 1987.

[7] T. Hickey and J. Cohen. Automating program analysis. *Journal of the ACM*, 35:185–220, 1988.

[8] D. Le Métayer. Ace: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[9] L. H. Ramshaw. Formalizing the analysis of algorithms, 1979. Ph. D. Thesis, Stanford University June 1979. Also available as Tech. Rep. SL-79-5, Xerox Palo Alto Research Center, Palo Alto, Calif.

[10] B. Salvy. *Asymptotique automatique et fonctions génératrices.* Thèse de doctorat, École Polytechnique, 1991.

[11] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[12] P. Zimmermann. *Séries génératrices et analyse automatique d'algorithmes.* Thèse de doctorat, École Polytechnique, Palaiseau, 1991.

[13] W. Zimmermann. *Automatische Komplexitätsanalyse funktionaler Programme.* PhD thesis, Fakultät für Informatik der Universität Karlsruhe, June 1990. Also available in the collection *Informatik Fachberichte*, number 261, Springer Verlag.