Lambda-Upsilon-Omega : an assistant algorithms analyzer

Philippe Flajolet, Paul Zimmermann, Bruno Salvy

To appear in Proc. AECC'6, Rome, July 1988, Lect. Notes in Comp. Sc.

LAMBDA-UPSILON-OMEGA: AN ASSISTANT ALGORITHMS ANALYZER

Philippe Flajolet
INRIA, Rocquencourt
78150 Le Chesnay (France)

Bruno Salvy
INRIA and Ecole Polytechnique
91405 Palaiseau (France)

Paul Zimmermann
INRIA, Rocquencourt

Abstract. Lambda-Upsilon-Omega, $\Lambda\Upsilon\Omega$, is a system designed to perform automatic analysis of well-defined classes of algorithms operating over "decomposable" data structures.

It consists of an 'Algebraic Analyzer' System that compiles algorithms specifications into generating functions of average costs, and an 'Analytic Analyzer' System that extracts asymptotic informations on coefficients of generating functions. The algebraic part relies on recent methodologies in combinatorial analysis based on systematic correspondences between structural type definitions and counting generating functions. The analytic part makes use of partly classical and partly new correspondences between singularities of analytic functions and the growth of their Taylor coefficients.

The current version $\Lambda\Upsilon\Omega_0$ of $\Lambda\Upsilon\Omega$ implements as basic data types, term trees as encountered in symbolic algebra systems. The analytic analyzer can treat large classes of functions with explicit expressions. In this way, $\Lambda\Upsilon\Omega_0$ can generate in the current stage about a dozen non-trivial average case analyses of algorithms like: formal differentiation, some algebraic simplification and matching algorithms. Its analytic analyzer can determine asymptotic expansions for large classes of generating functions arising in the analysis of algorithms.

The outline of a design for a full system is also discussed here. The long term goal is to include a fairly rich set of data structuring mechanisms including some general recursive type definitions, and have the analytic analyzer treat wide classes of functional equations as may be encountered in combinatorial analysis and the analysis of algorithms.

1. Introduction

Ideally, a system for automatic program analysis should take as input a procedure or function specification

I1. `procedure Quicksort {instructions} end;`

I2. `procedure Diff {instructions} end;`

for sorting or computing symbolic derivatives, and produce an "analysis" of the program. We concern ourselves here with *average–case* analysis and optimization of programs, and we would like the system to output something like

O1. `Time for Quicksort on random inputs of size n is`
`11.67 n ln(n) - 1.74 n + O(ln(n))`

O2. `Time for Diff on random inputs of size n is`

$$8\,\frac{n\,(-240+37\,42^{1/2})}{(-13+2\,42^{1/2})\,42^{1/2}}+O(1)$$

These two analyses will naturally depend on *type specifications* that are companion to (I1) and (I2), a description of a *complexity model* (e.g. an '`if`' takes 5 units of time), and a description of a (random input) *statistical model*. For Quicksort, we could have some way of specifying that all permutations of n are taken equally likely, while for Diff we could decide that all expression trees of size n with the proper type are equally likely.

We shall describe here a system whose current state performs automatic analysis of a whole *class* of algorithms in the realm of symbolic manipulation algorithms and contains a good deal of what is needed in order to analyze permutation algorithms like (I1). Result (O1) is taken directly from [Knuth 1973], but (O2) was literally produced by our system.

Our system is called $\Lambda\Upsilon\Omega$. The name $\Lambda\Upsilon\Omega$ (Lambda-Upsilon-Omega) comes from the Greek word $\lambda\acute{\upsilon}\omega$ which means (amongst other things) 'I solve', and it is from this verb that "analysis" derives. Implementation was started in mid 1987. We shall describe here its overall design principles as well as the state of the current implementation $\Lambda\Upsilon\Omega_0$. There are two major components in $\Lambda\Upsilon\Omega$:

- An *Algebraic Analyser System*, ALAS, that accepts algorithms specifications in a suitably restricted programming language. That part produces *type descriptors* and *complexity descriptors* in the form of *generating functions*.
- An *Analytic Analyser System*, ANANAS, that accepts generating functions (for type descriptors and complexity descriptors) and tries to determine automatically an asymptotic expansion of its Taylor coefficients.

The algebraic component is currently implemented in Lisp, though ML is also considered for later implementations. In its present form, it permits to analyze a class of symbolic term (tree) manipulation programs and comprises about 500 Lisp instructions (in the Le_Lisp dialect). The analytic component is already a fairly large set of symbolic "algebra" routines written in Maple and comprising about 3000 instructions.

Both components encapsulate a fair amount of mathematical expertise at a rather abstract level.

- The algebraic system is based on research in combinatorial analysis developed mostly during the 1970's regarding correspondence between structural definitions of combinatorial objects and generating functions, together with some new extensions to program schemes.
- The analytic system is based on some recent developments of late 19th and early 20th century complex asymptotic analysis concerning the correspondence between singularities or saddle points of functions and the asymptotic order of coefficients in Taylor expansions.

The $\Lambda\Upsilon\Omega$ system has of course no claim of being universal, since program termination is in general undecidable. Its interest lies in consideration of a restricted class of purely "functional" procedures that operate through recursive descent over a large class of "decomposable" structures defined by powerful type structuring mechanisms. Such a class contains algorithms and data structures, like binary search trees, unbalanced heaps for priority queues, quicksort, digital tries and radix exchange sort, merge sort, several

versions of hashing, pattern matching, recursive parsing. Our long term objective is to have a system that will perform automatically analysis of a non-negligible fractions of these algorithms as well as many other of the same style. The current system implements a complexity calculus on term trees along the lines of [Flajolet, Steyaert 1987] and the analytic analyzer is already appreciably more general.

For an interesting alternative approach to automatic complexity analysis, the reader is referred to [Hickey, Cohen 1988] and references therein.

2. A Sample Session

A typical $\Lambda\Upsilon\Omega$ session† starts with by calling a script, which (using Unix virtual tty's) initiates a joint Lisp and Maple session. We then load ALAS, apply it to the program to be analyzed. This generates a set of equations over generating functions, that are passed to Maple initialized with ANANAS.

The example considered is a program that computes symbolic derivatives (without simplification) of expressions (terms, trees) built from the operator set

$$1^{(0)},\ x^{(0)},\ \exp^{(1)},\ +^{(2)},\ *^{(2)},\ \div^{(2)}$$

with superscripts denoting arities. The key steps are

1. The recursive definition of the type 'term' is reflected by a quadratic equation for its generating function $t(z)$.
2. The recursive structure of the Diff procedure is reflected by a linear equation for its complexity descriptor $\tau diff$ (generating function of average costs).

These two steps are completed automatically by ALAS, which also uses a small Maple procedure to derive explicit expressions. At the next stage, ANANAS is used on those generating functions:

3. Both $t(z)$ and $\tau diff(z)$ are recognized as having singularities at a finite distance of a so-called 'algebraico-logarithmic' type. Local singular expansions are then determined (through Maple's Taylor capability).
4. Using general theorems from complex asymptotic analysis, singular expansions can be transformed automatically into asymptotic expansions of the coefficients. This is achieved by means of the versatile '`equivalent`' command of ANANAS.

Dividing the asymptotic form of the coefficients of $[z^n]$ in $t(z)$ and $\tau diff(z)$, we obtain the *asymptotic* average complexity of symbolic differentiation in an either algebraic or floating point form. The same device may be used to analyse the variant DiffCp of Diff that proceeds by copying subexpressions instead of sharing them.

In this way we obtain average case analyses which we summarized here, compared to the obvious best case and worst case results.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
`Diff` [sharing]	$O(n)$	$c.n + O(1)$	$O(n)$
`DiffCp` [copy]	$O(n)$	$c.n^{3/2} + O(n)$	$O(n^2)$

The *order* of the cost for Diff was to expected. The $O(n^{3/2})$ result for DiffCp is harder to guess, and it is related to the behaviour of the average path length in trees as discussed in [Knuth 1968] or [Meir, Moon 1978].

† The necessary concepts will be developed in Sections 3 (Algebraic System) and 4 (Analytic System). The script that follows has been slightly edited and a few commands have been decomposed for the sake of readability.

```
Script started on Fri Jan 29 12:08:24 1988
% maplelisp    [Initialize Lisp and Maple]
; Le-Lisp (by INRIA) version 15.21  (25/Dec/87)  [sun]
    |\^/|
._|\|   |/|_. INRIA - Rocquencourt
 \  MAPLE  /  Version 4.1 --- May 1987
 <____ ____>  For on-line help, type  help();
      |

[Load Algebraic Analyzer, Alas]
? (load luo)
Function to analyse : (diff copy1 distrib size simpl count1 filter
copy evalf diffcp)
? diff [Diff with subexpression sharing]
(de diff (x)
    (caseroot   *    ('+    (maketree    '+
    [etc...]
Analysis of diff on trees formed with ...
   symbol x of arity 0 etc...
Calling analyse ...
? (maple)
*** Initial system ***   [Equations Produced by Alas]
                2                    2
 phi = 2 + 3 z  + z, t = 2 + 3 z t  + z t,
                    2
   tau_diff = 21 z t  + 6 z t tau_diff + 4 z t + z tau_diff + 2 z
*** Explicit expression(s) ***   [etc...]

[Start Analytic Analyzer, Ananas]
> tn:=equivalent(t); [Asymptotic number of inputs: 16 sec on sun3/60]
                          1/4            1/2 1/2
                       42    (13 - 2 42   )
tn := - 1/6 ----------------------------------------------
                          1/2   3/2   1/2           1/2 n
             (- 13 + 2 42   ) n    Pi    (13 - 2 42   )
                    1/2 (- n)
          (13 - 2 42   )
   + O(--------------------)
                   5/2
                  n
>  [etc...]    [average diff time]
                                                  1/2
                                   n (- 240 + 37 42   )
                av_diffn := 8 ------------------------ + O(1)
                                              1/2    1/2
                                   (- 13 + 2 42   ) 42
> evalf(av_diffn);    [Floating point eval]
                                  6.82942 n + O(1)
> suspend_ maple;
? (analyse 'diffcp)   [Diff with copy of arguments]
> [etc ...]
> evalf(av_diffcpn);
                                        3/2
                                16.6226 n    + O(n)
> suspend_maple;
? (end)    [The whole thing!]
Que Le-Lisp soit avec vous.
% ^D
script done on Fri Jan 29 12:44:51 1988
```

Figure 1. A $\Lambda\Upsilon^{\Omega}$ session showing the automatic analysis of symbolic differentiation.

3. The Algebraic Analyzer System

3.1 Combinatorial Principles

The algebraic part of our system – ALAS – relies on recent research in combinatorial analysis. Till the mid twentieth century, the field of combinatorial enumerations was mostly conceived as an art of obtaining *recurrences* for the counting of combinatorial structures, with generating functions entering as an *ad hoc* solution device in more complex cases. The books by Riordan, and many of the analyses in Knuth's *magnum opus* are witnesses of this approach.

From research conducted by Rota, Foata, Schützenberger and their schools, there has emerged a general principle:

> *A rich collection of combinatorial constructions have direct translation into generating functions.*

More precisely, let $\mathcal{A}$ be a class of combinatorial objects, with $\mathcal{A}_n$ the subclass consisting of objects of size n, and $A_n = \mathrm{card}(\mathcal{A}_n)$. We define the *ordinary generating function* (OGF) and *exponential generating function* (EGF) of $\mathcal{A}$ by

$$A(z) = \sum_{n\geq 0} A_n z^n \qquad \text{and} \qquad \hat{A}(z) = \sum_{n\geq 0} A_n \frac{z^n}{n!}. \tag{3.1}$$

A combinatorial construction, say $\mathcal{C} = \Phi[\mathcal{A}, \mathcal{B}]$ is said to be *admissible* if the counting sequence $\{C_n\}_{n\geq 0}$ of the result depends only on the counting sequences $\{A_n\}$ and $\{B_n\}$ of the arguments. An admissible construction then defines an *operator* (or a *functional*) on corresponding generating functions:

$$C(z) = \Psi[A(z), B(z)] \qquad \text{and} \qquad \hat{C}(z) = \hat{\Psi}[\hat{A}(z), \hat{B}(z)].$$

For instance the cartesian product construction is admissible since

$$\mathcal{C} = \mathcal{A} \times \mathcal{B} \qquad \Longrightarrow \qquad C_n = \sum_{k=0}^{n} A_k B_{n-k} \quad \text{and} \quad C(z) = A(z) \cdot B(z).$$

Combinatorial enumerations are developed systematically within a comparable framework in the book by Goulden and Jackson [1983]. The tables in Figure 2 summarize a collection of admissible constructions borrowed from [Flajolet 1985, 1988].

In this context, the primary object for combinatorial enumerations is no longer *integer sequences* but rather *generating functions*. Furthermore, that approach fits nicely with asymptotic analysis, the main tool for asymptotic analysis being analytic function theory rather than explicit integer sequences. It is on the conjunction of these two principles that our system is built.

The task of enumerating a class of combinatorial structures is then reduced to *specifying* it (up to isomorphism) by means of admissible constructions. Once this is done, the task of computing a set of generating function equations reduces to performing simply a purely formal translation. In the context of the analysis of algorithms, data structure declarations are thus converted to generating functions (GF's), each data type having its own GF also called its type *descriptor*. An interesting approach similar to ours and based on an extension of context–free grammars is presented in Greene's thesis [1983].

OGF:

Disj. Union	$\mathcal{C} = \mathcal{A} \uplus \mathcal{B}$	$c(z) = a(z) + b(z)$
Cart. Product	$\mathcal{C} = \mathcal{A} \times \mathcal{B}$	$c(z) = a(z) \cdot b(z)$
Diagonal	$\mathcal{C} = \Delta(\mathcal{A} \times \mathcal{A})$	$c(z) = a(z^2)$
Sequence	$\mathcal{C} = \mathcal{A}^*$	$c(z) = (1 - a(z))^{-1}$
Marking	$\mathcal{C} = \mu\mathcal{A}$	$c(z) = z\frac{d}{dz}a(z)$
Substitution	$\mathcal{C} = \mathcal{A}[\mathcal{B}]$	$c(z) = a(b(z))$
PowerSet	$\mathcal{C} = 2^{\mathcal{A}}$	$c(z) = \exp\left(a(z) - \frac{1}{2}a(z^2) + \frac{1}{3}a(z^3) - \cdots\right)$
MultiSet	$\mathcal{C} = \mathrm{M}\{\mathcal{A}\}$	$c(z) = \exp\left(a(z) + \frac{1}{2}a(z^2) + \frac{1}{3}a(z^3) + \cdots\right)$

EGF:

Disj. Union	$\mathcal{C} = \mathcal{A} \uplus \mathcal{B}$	$\hat{c}(z) = \hat{a}(z) + \hat{b}(z)$
Label. Product	$\mathcal{C} = \mathcal{A} * \mathcal{B}$	$\hat{c}(z) = \hat{a}(z) \cdot \hat{b}(z)$
Label. Sequence	$\mathcal{C} = \mathcal{A}^{\langle * \rangle}$	$\hat{c}(z) = (1 - \hat{a}(z))^{-1}$
Marking	$\mathcal{C} = \mu\mathcal{A}$	$\hat{c}(z) = z\frac{d}{dz}\hat{a}(z)$
Label. Subst.	$\mathcal{C} = \mathcal{A}[\mathcal{B}]$	$\hat{c}(z) = \hat{a}(\hat{b}(z))$
Label. Set	$\mathcal{C} = \mathcal{A}^{[*]}$	$\hat{c}(z) = \exp\left(\hat{a}(z)\right)$

Figure 2. A catalog of admissible constructions and their translation to ordinary or exponential generating functions. The OGF constructions are relative to unlabelled structures, the EGF constructions are relative to labelled structures.

3.2. Analysis of Algorithms

Let Γ be an algorithm that takes its inputs from a data type $\mathcal{I}$ and produces some output of type $\mathcal{O}$. We consider exclusively *additive complexity measures*, thereby restricting ourselves to *time complexity* analyses. Let $\tau\Gamma[e]$ denote the complexity of an execution of Γ on input e. By the additive character:

$$\Gamma = (\Gamma^{(1)}; \Gamma^{(2)}) \qquad \Longrightarrow \qquad \tau\Gamma = \tau\Gamma^{(1)} + \tau\Gamma^{(2)}.$$

The purpose of average case analysis is to determine the expectation of $\tau\Gamma[e]$ when e is a random element of $\mathcal{I}$ with size n. Thus, assuming $\mathcal{I}_n$ is a finite set, that quantity is a quotient

$$\frac{\tau\Gamma_n}{I_n} \qquad \text{where} \qquad \tau\Gamma_n = \sum_{e \in \mathcal{I}_n} \tau\Gamma[e],$$

$\tau\Gamma_n$ being thus a cumulated value of $\tau\Gamma$ over $\mathcal{I}_n$.

The ordinary *complexity descriptor* (OCD) of algorithm Γ is defined as the generating function

$$\tau\Gamma(z) = \sum_{n \geq 0} \tau\Gamma_n \, z^n.$$

(There is an obvious analogue for exponential descriptors.)

A program construction $\Gamma = \Phi[\Gamma^{(1)}, \Gamma^{(2)}]$ is said to be *admissible* if the cost sequence $\{\tau\Gamma_n\}$ of Γ depends only on the cost sequences of the arguments Γ_1, Γ_2 and the counting sequences of intervening data structures. An admissible construction again defines an operator over corresponding generating functions.

Assume for instance that $P(z : \mathcal{C})$ is a procedure that operates on inputs $z = (a, b)$ of type $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ and is defined by

```
P(z :  C) := Q(a);
```

where Q is of type $Q(x : \mathcal{A})$. Then, is is easy to see that

$$\tau P(z) = \tau Q(z) \cdot b(z).$$

If in addition, we make use of the additivity of τ, the scheme

```
P(x :  C) := Q(a); R(b)
```

translates into

$$\tau P(z) = \tau Q(z) \cdot b(z) + a(z)\tau R(z).$$

It turns out that there is a collection of program schemes naturally associated to constructions described above that are admissible. Corresponding to $\mathcal{C} = \mathcal{A} \uplus \mathcal{B}$, $\mathcal{C} = \mathcal{A} \times \mathcal{B}$, $\mathcal{C} = \mathcal{A}^*$, $\mathcal{C} = 2^{\mathcal{A}}$, $\mathcal{C} = \mathbf{M}\{\mathcal{A}\}$, we find

$P(c) =$ **if** $c \in \mathcal{A}$ **then** $Q(c)$ **else** $R(c)$	$\tau P(z) = \tau Q(z) + \tau R(z)$
$P((a, b)) = Q(a)$	$\tau P(z) = \tau Q(z) b(z)$
$P((a_1, \ldots, a_k)) = Q(a_1); \ldots; Q(a_k)$	$\tau P(z) = \tau Q(z)/(1 - a(z))^2$
$P(\{a_1, \ldots, a_k\}) = Q(a_1); \ldots; Q(a_k)$	$\tau P(z) = c(z)(\tau Q(z) - \tau Q(z^2) + \tau Q(z^3) - \cdots)$
$P(\{a_1, \ldots, a_k\}) = Q(a_1); \ldots; Q(a_k)$	$\tau P(z) = c(z)(\tau Q(z) + \tau Q(z^2) + \tau Q(z^3) + \cdots)$

For instance, a recursive type definition for trees

$T \;=\; \{a\} \times T^*;$

together with a recursive procedure specification sheme

$Q(x : T) \;:=\; R(x);$ **for** y **root_subtree_of** x **do** $Q(y);$

will result in the system of equations

$$\begin{cases} T(z) = \dfrac{z}{1 - T(z)} \\ \tau Q(z) = \tau R(z) + z \dfrac{\tau Q(z)}{(1 - T(z))^2}. \end{cases}$$

Observe that $T(z)$ is an algebraic function of degree 2, and owing to the structure of the algorithm, $\tau Q(z)$ is expressed *linearly* in terms of itself. This is roughly the situation that we encounter when analyzing symbolic differentiation as well as many similar algorithms [Steyaert 1984].

The algebraic analyzer of $\Lambda\Upsilon\Omega_0$ implements a calculus based on previously exposed principles, but restricted to trees. Nonetheless (cf Fig. 1), it can produce automatic analyses of versions of matching, simplification, or various types of evaluations etc.

4. The Analytic Analyzer System

At this stage, our task is to take a generating function, defined either explicitly (for non recursive data types) or implicitly via a functional equation (for most recursive data types). The current version of $\Lambda\Upsilon\Omega$ treats only functions that lead to explicit expressions after a possible usage of the 'solve' routine of Maple. We shall therefore limit ourselves to this case.

4.1. Analytic Principles

Let $f(z)$ be a function analytic at the origin. We assume further that $f(z)$ is explicitly given by an *expression*, a blend of sums, products, powers, exponential and logarithms. Most explicit generating functions constructed by the combinatorial tools of Section 3 are of this type.

The starting point is Cauchy's coefficient formula

$$[z^n]f(z) = \frac{1}{2i\pi}\int_\Gamma f(z)\,\frac{dz}{z^{n+1}}, \tag{4.1}$$

where $[z^n]f(z)$ is the usual notation for the coefficient of z^n in the Taylor expansion of $f(z)$. Two major classes of methods are applicable to determine the Taylor coefficients of those functions:

- For functions with singularities at a finite distance, the local behaviour of the function near its dominant singularities (the ones of smallest modulus) determines the growth order of the Taylor coefficients of the function. Asymptotic information is obtained by taking Γ to be a contour that comes close to the dominant singularities.
- For entire functions, saddle point contours Γ are usually applicable.

Several observations are useful here. First, functions defined by expressions are analytically continuable – except for possible isolated singularities – to the whole of the complex plane (though they may be multivalued). Second, by Pringsheim theorem, functions with positive coefficients (such is the case for our generating functions) always have a positive dominant singularity, a fact that eases considerably the search for singularities.

Though a complete algorithm covering all elementary functions is not (yet) available since the classification of singularities, even for such functions, is not fully complete, a good deal of functions arising in practice can be treated by the following algorithm.

Procedure equivalent(f : expression) : expression;

{determines an asymptotic equivalent of $[z^n]f(z)$}

1. Determine whether $f(z)$ is entire or $f(z)$ has singularities at a finite distance.

2. If $f(z)$ has finite singularities, let ρ be the modulus of a dominant singularity. We know at least that

$$f_n = [z^n]f(z) \approx \rho^{-n}.$$

Compute a local expansion of $f(z)$ around its dominant singularity (-ies). This is called a singular expansion.

2a. If a singular expansion is of an 'algebraico-logarithmic' type, namely

$$f(z) \sim (1 - z/\rho)^\alpha \log^\beta(1 - z/\rho)^{-1} \qquad \text{as} \qquad x \to \rho \tag{4.2}$$

then apply methods of the Darboux-Pólya type to transfer singular expansions to coefficients

$$f_n = [z^n]f(z) \sim \rho^{-n}\frac{n^{-\alpha-1}}{\Gamma(-\alpha)}\log^\beta n \tag{4.3}.$$

This applies generally to functions that are "not too large" near a singularity.

2b If the function is large near its singularity, for instance

$$f(z) \approx \exp\Big(\frac{1}{(1 - z/\rho)^\alpha}\Big),$$

then apply saddle point methods like (3) below.

3. If $f(z)$ is entire, then use a saddle point integral. If this succeeds, we get

$$f_n = [z^n] f(z) \sim \frac{e^{h_n(R_n)}}{\sqrt{2\pi h_n(R_n)}}$$

where $h_n(z) = \log f(z) - (n+1)\log z$, and R_n is such that

$$\left[\frac{dh_n(z)}{dz}\right]_{z=R_n} = 0.$$

This is the outline of the algorithm that we have implemented in Maple, with the minor exception of step (2b) (saddle point at a finite distance) and with the current limitation that singularities and saddle points should be within reach of Maple's 'solve' routine.

It is important to note that a few *theorems*, whose conditions can be automatically tested, are used to support this algorithm.

Singularity Analysis. The classical form of the Darboux-Pólya method requires differentiability conditions on error terms. However, from [Flajolet, Odlyzko 1987], we now know that analytic continuation is enough to ensure the transition from (4.2) to (4.3), and by our earlier discussion, these conditions are always fulfilled for functions defined by expressions. Thus, the use of (2a) is guaranteed to be sound. Furthermore, that approach makes it possible to cope with singularities involving iterated logarithms as well (not yet implemented).

Saddle Point Integrals. There has been considerable interest for those methods, due to their recognized importance in mathematical physics and combinatorial enumerations. We thus know, from works by Hayman, Harris and Schoenfeld, or Odlyzko and Richmond, classes of functions *defined by closure properties* for which saddle point estimates are valid. Such conditions, that are extremely adequate for combinatorial generating functions, can be checked inductively on the expression.

4.2. Some Applications

Let us take here the occasion of a few examples to discuss some further features of ANANAS. The next three examples are all taken from combinatorial enumerations: (E1) Trees of cycles of cycles of beads; (E2) Involutive permutations; (E3) Children's Rounds of [Stanley 1978]; (E4) Bell numbers counting partitions of n.

```
[E1]> equivalent(1/2*(1-sqrt(1-4*log(1/(1-log(1/(1-z)))))));
                      1/2
-1/2   exp(exp(-1/4))    exp(-3/8)
           3/2                                    1/2
      1 / (n    (- exp(exp(-1/4)) exp(-1) + 1)    )
                                             n   1/2
         / ((- exp(exp(-1/4)) exp(-1) + 1)  Pi    ) + etc ...
                                    (1/2 - n)
        (- exp(exp(-1/4) - 1) + 1)
   + O(-----------------------------------)
                         7/2
                        n
```

```
[E2]> equivalent(exp(z+z^2/2));
                          1/2                                  1/2 2
exp(- 1/2 - 1/2 (1 + 4 n)    + 1/2 (- 1/2 - 1/2 (1 + 4 n)   ) )

                            1/2 n     1/2   1/2                        1/2 1/2
   / ((- 1/2 - 1/2 (1 + 4 n)   )  (-2)    Pi    (- 1/2 - 1/2 (1 + 4 n)   )   )

             1/4
   / (1 + 4 n)


[E3]> equivalent((1-z)^(-z),5);
          ln(n)     gamma       1     ln(n)           1     gamma       ln(n)
1 - 1/n - ------- - ------- + ---- - ------- + 5/2 ---- - ------- + O(--------)
             3         3        3        4           4        4           5
            n         n        n        n           n        n           n

[E4]> equivalent(exp(exp(z)-1));
                                 1/2
1/2   exp(exp(W(n + 1)) - 1) 2    + etc... [W(z)exp(W(z))=z]
```

Example 1 demonstrates the processing of functions with singularities at a finite distance. The singularity has an explicit form and the function behaves locally like a square root, whence the final result of the form:

$$Cn^{-3/2}\left(1-e^{e^{-1/4}-1}\right)^{-n}.$$

Example 2 shows the asymptotic analysis of the Involution numbers (Knuth [1973, p.65-67] does it by the Laplace method). It is treated here by "Hayman admissibility" (a classical notion bearing no direct relation to our previous usage of this word). Hayman's Theorem provides a class of admissible functions for which a saddle point argument (Step 3 of procedure 'equivalent') can be applied: *If f and g are admissible, h is an entire function and P is a real polynomial with a positive leading coefficient, then*

$$\exp(f),\ f+P,\ P(f),\ \textit{and under suitable conditions, } f+h$$

are admissible. These conditions can be checked syntactically here.

Example 3 is a further illustration of singularity analysis in a non trivial case. Example 4, the classical asymptotics of Bell numbers [De Bruijn 1981] resembles Example 2. It is treated here by Harris–Schoenfeld admissibility (which also provides complete expansions), and the corresponding step in the algorithm implements a theorem of Odlyzko and Richmond relating Hayman admissibility to Harris–Schoenfeld admissibility.

5. Conclusions

We have presented here some preliminary design considerations for a system that would assist *research* in the analysis of algorithms. There are two benefits to be expected from such a research.

The first and most obvious benefit is to help an analyst explore some statistical phenomena that seem "tractable" in principle, but too intricate to be done by hand.

The second and most important one in our view is that the design of such a system creates *needs* of a new nature in algorithmic analysis methodology. (Thus, our approach departs radically from "Artificial Intelligence").

a. There is a need for extracting *general* program schemes that can be analyzed by these methods. In this way, we wish to attack the analysis of elementary but structurally complex programs, and we can hope to find general theorems relating complexity and structure of algorithms (cf [Steyaert 1984]).

b. When making algorithmic some parts of complex asymptotics, we naturally discover "gaps" that have never been revealed before. For instance, nobody seems to have considered such a simple asymptotic problem as

$$[z^n] \exp\left(\log^2\left(\frac{1}{1-z}\right)\right)$$

In summary, all we hope is that the development of $\Lambda\Upsilon^{\Omega}$ will bring even more questions than answers!

6. Bibliography

N. G. de Bruijn [1981]. *Asymptotic Methods in Analysis.* Dover, New York, 1981.

L. Comtet [1974]. *Advanced Combinatorics.* Reidel, Dordrecht, 1974.

P. Flajolet [1985]. "Elements of a general theory of combinatorial structures", in *Proc. FCT Conf., Lecture Notes in Comp. Sc*, Springer Verlag, 1985, 112-127.

Ph. Flajolet [1988]. "Mathematical Methods in the Analysis of Algorithms and Data Structures," in *Trends in Theoretical Computer Science*, E Börger Editor, Computer Science Press, 1988.

P. Flajolet and A. M. Odlyzko [1987]. "Singularity Analysis of Generating Functions", preprint, 1987.

P. Flajolet and J-M. Steyaert [1987]. "A Complexity Calculus for Recursive Tree Algorihms", *J. of Computer and System Sciences* **19**, 1987, 301-331.

I. Goulden and D. Jackson [1983]. *Combinatorial Enumerations.* Wiley, New York, 1983.

D. H. Greene [1983]. "Labelled Formal Languages and Their Uses," Stanford University, Technical Report STAN-CS-83-982, 1983.

B. Harris and L. Schoenfeld [1968]. "Asymptotic Expansions for the Coefficients of Analytic Functions", *Illinois J. Math.* **12**, 1968, 264-277.

W. K. Hayman [1956]. "A Generalization of Stirling's Formula", *J. Reine und Angewandte Mathematik* **196**, 1956, 67-95.

P. Henrici [1977]. *Applied and Computational Complex Analysis.* Three Volumes. Wiley, New York, 1977.

T. Hickey and J. Cohen [1988]. "Automatic Program Analysis", *J.A.C.M.* **35**, 1988, 185-220

D. E. Knuth [1973a]. *The Art of Computer Programming.* Volume 1: *Fundamental Algorithms.* Addison-Wesley, Reading, MA, second edition 1973.

D. E. Knuth [1973b]. *The Art of Computer Programming.* Volume 3: *Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

A. Meir and J. W. Moon [1978]. "On the Altitude of Nodes in Random Trees," *Canadian Journal of Mathematics* **30**, 1978, 997–1015.

G. Pólya [1937]. "Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen", *Acta Mathematica* **68**, 1937, 145–254. Translated in: G. Pólya and R. C. Read, *Combinatorial Enumeration of Groups, Graphs and Chemical Compounds*, Springer, New-York, 1987.

V. N. Sachkov [1978]. *Verojatnostnie Metody v Kombinatornom Analize,* Nauka, Moscow, 1978.

R. SEDGEWICK [1983]. *Algorithms*. Addison-Wesley, Reading, 1983.

R. P. STANLEY [1978]. "Generating Functions," in *Studies in Combinatorics*, edited by G-C. Rota, M. A. A. Monographs, 1978.

R. P. STANLEY [1986]. *Enumerative Combinatorics*, Wadsworth and Brooks/Cole, Monterey, 1986.

J-M. STEYAERT [1984]. "Complexité et Structure des Algorithmes", These de Doctorat ès-Sciences, Université Paris 7, 1984.