Function Composition and Automatic Average case Analysis

Paul Zimmermann*
zimmermann@inria.inria.fr

Abstract

We define the composition of functions defined over extended context-free languages. We show that this composition is automatically computable. It enables the automatic analysis of complex problems with *small* input descriptions, for example repeated differentiation or iterated automata on regular languages.

Keywords: analysis of algorithms, average-case complexity, automatic analysis, program transformation.

1 Introduction

In the field of automatic complexity analysis, the length of the problem description is often a limitation: writing a long program is not only boring, but may introduce some errors. Thus we need some powerful constructs to describe algorithms, with the necessary constraint that these constructs allow an *automatic* analysis.

We are particularly interested here in the average case analysis of programs including some *compositions* of functions. To our knowledge, none of the existing systems, including METRIC [3], COMPLEXA [7] or Lambda-Upsilon-Omega [2], is able to analyze the composition of functions. The main reason may be the following: in these systems, the analysis of statements like $f(x)$ relies on the fact that all required data types are defined, either implicitly like in METRIC and COMPLEXA where all data structures are lists, or explicitly like in Lambda-Upsilon-Omega. But in the statement $f(g(y))$, the difficulty is to get a formal description of the object $g(y)$, which is not known *a priori.*

As an example, suppose we have written a function *diff* performing the differentiation of symbolic expressions with respect to one variable, and now we would like to analyze the two-fold differentiation by just defining

$$\mathit{diff2}(e) \overset{\text{def}}{=} \mathit{diff}(\mathit{diff}(e))$$

*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).

instead of having to write the entire body of the function *diff2*. We show in this paper that this shorthand is possible. More precisely, we define a class of programs including function compositions, such that every program can be ***automatically*** expanded into another one without any composition, and equivalent to the original one in what concerns complexity analysis. This result allows to define and to analyze large problems by short description programs.

The paper contains two main sections: in Section 2, we define a class of programs including compositions in terms of the Lambda-Upsilon-Omega system language, and we state the main result (Theorem 1): these programs are automatically reducible to other equivalent programs, the latter lying in a class where automatic analysis methods are already known. In Section 3, we explain how the implementation of Theorem 1 helped us to guess a conjecture concerning iterated differentiation, and to obtain an interesting result about the Collatz conjecture; it is also shown that using composition may drastically reduce the size of the description program.

2 A class of programs with composition

In this section, we will introduce the composition of functions in a language called ADL (Algorithm Description Language), especially designed for automatic average case analysis in the Lambda-Upsilon-Omega system [1, 2]. Instead of giving a formal description of this language, we prefer to show as example an ADL program performing the differentiation of symbolic expressions.

An ADL program has three parts: the data type specification, the definition of one or more procedures, and the declaration of complexity measures. The data type specification looks like a formal language grammar; symbolic expressions constructed with $0, 1$, the variable x and the binary operators $+$ and $\times$ are defined for example as follows:

```
type expression = zero | one | x
                | plus(expression,expression)
                | times(expression,expression);
     plus,times,zero,one,x = atom(1);
```

where the last line defines $+, \times, 0, 1, x$ as atoms (or terminals) of *size* 1. The size is additive, thus the size of an expression as defined above is the number of atoms it contains. For example,

$$E = \texttt{plus(times(x,x),one)}$$

is of type `expression` and of size 5. The function computing the derivative of such expressions with respect to x is written like this:

```
function diff(e : expression) : expression;
begin
   case e of
      plus(e1,e2)      : plus(diff(e1),diff(e2));
      times(e1,e2)     : plus(times(diff(e1),copy(e2)),
                                times(copy(e1),diff(e2)));
      zero             : zero;
```

```
        one                  : zero;
        x                    : one
    end;
end;
```

where the function copy, which simply makes a carbon copy of one expression, is also defined in the same manner. If we apply this function *diff* on the above expression E, we obtain

$$\mathit{diff}(E) = \mathtt{plus(plus(times(one,x),times(x,one)),zero)}$$

(the *diff* function performs no simplification). Now if we want to define the cost of the function as the number of atoms in the output of *diff*, it suffices to define the cost of each atom as 1:

```
measure plus,times,zero,one,x : 1;
```

With this declaration, the cost of *diff*(E) is 9. If we analyze the *diff* function in the Lambda-Upsilon-Omega system, we will get the following average cost for expressions of size n:

$$\tau\overline{\mathit{diff}}_n = \frac{1}{4}\sqrt{2\pi}n^{3/2} + O(n).$$

More details about Lambda-Upsilon-Omega or ADL will be found in [2] or [5].

Now we allow the use of the composition in ADL programs, that is statements of the form $f(g(y))$ where f and g are two functions defined in the program, and y is a local variable. For example, the second order differentiation is defined as follows

```
function diff2(e : expression) : expression;
begin
    diff(diff(e))
end;
```

Definition 1 *The* composition graph *associated to an* ADL *program is the graph whose vertices are the function names, and for each composition $f(g(\ldots))$ in the body of a function h, there is an arrow from h to all functions on which f and g depend.*[1]

Theorem 1 *If the composition graph of an* ADL *program is acyclic, then the program translates into an equivalent program without composition.*

Proof: [Sketch] If the composition graph is acyclic, it is possible to totally order the functions, say $h_1, \ldots, h_k$, such that all arrows starting from a function go to functions of smaller index. Then we *expand* the body of the functions in increasing index order, that is we compute an equivalent body without any composition. Expanding h_1 is trivial because there is no composition in the body of h_1. Suppose we have already expanded $h_1, \ldots, h_{j-1}$. For each composition $f(g(\ldots))$ appearing in the body of h_j, the functions f and g are necessary of smaller index, therefore they have already been expanded.

Thus the only difficulty is to expand a call $f(g(x))$ where f and g have already been expanded. To do this, we replace this call by $k(x)$, where k is a new function name. We put as body for k the body of g where every returned expression y is replaced by $f(y)$. We

[1]The relation "depends on" is the reflexive and transitive closure of the relation "has in its body".

then simplify the expression $f(y)$ according to the body of f while it is possible. During this simplification, some compositions $f' \circ g'$ may appear. In that case either f' (same for g') has been created during the expansion process (thus is already expanded), or there is necessary an arrow from h_j to f', therefore f' has already been expanded. As the number of such new compositions that may appear is bounded (by the number of functions f' on which f depends by the number of functions g' on which g depends), the expansion of the body of h_j will eventually terminate. ∎

As the average case analysis of ADL programs *without* composition is already known to be automatic [2, 5], the above theorem implies directly the following result:

Corollary 1 *The average case analysis of* ADL *programs with an acyclic composition graph is possible automatically.*

An example of ADL program whose composition graph has a cycle is the following:

```
type integer = one | one integer;
     one = atom(1);

function f (i : integer) : integer;
begin
   case i of
      one : one;
      (one,j) : f(f(j))
   end;
end;
```

The expansion process described above will not terminate, though the function f simply returns one for all inputs.

3 Automatic analysis of programs with composition

In this section, we present two research problems where the implementation of the expansion process on a computer allowed us to discover some results which would have been very difficult to find at hand. We also prove an interesting result: the expansion process may produce exponentially large programs, with respect to the initial one.

3.1 Analysis of kth order differentiation

The expansion process described in the proof of Theorem 1 has been encoded in an experimental version (V1.4) of the system Lambda-Upsilon-Omega. When we analyze the function `diff2` as defined in Section 2, the system displays with "printlevel" 3 the expanded form of the function body:

```
function diff_of_diff (e : expression) : expression;
begin
   case e of
```

```
    (plus,(e1,e2)) : plus(diff_of_diff(e1),diff_of_diff(e2));
    (times,(e1,e2)) : plus(plus(times(diff_of_diff(e1),copy_of_copy(e2)),
                                times(copy_of_diff(e1),diff_of_copy(e2))),
                           plus(times(diff_of_copy(e1),copy_of_diff(e2)),
                                times(copy_of_copy(e1),diff_of_diff(e2))));
    zero : zero;
    one : zero;
    x : zero;
  end;
end;
```

Three other new functions have been introduced, namely diff_of_copy, copy_of_diff and copy_of_copy (the function copy is not initially known by the system). The system then proceeds in the usual way (Algebraic Analysis, Solver, Analytic Analysis) described in [2] and gives the final result:

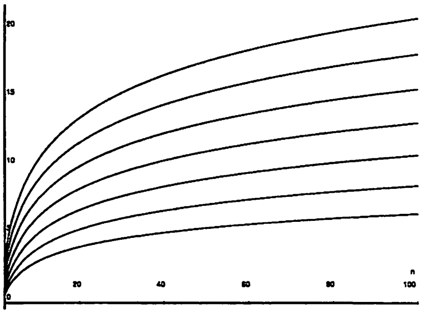
```
Average cost for diff2 on random inputs of size n is:

                                          2          3/2
                                   (1/2 n ) + (O(n     ))

for n mod 2 = 1, and 0 otherwise.
```

In this way, by just adding one more call to the function diff, we were able to analyze the k-fold iterated differentiation until $k = 7$. To figure out the difficulty of the task, just try to write the body of the function diff3 without any error! We obtained the following figures.

	average cost
diff	$\frac{1}{4}\sqrt{2\pi}n^{3/2} + O(n)$
diff2	$\frac{1}{2}n^2 + O(n^{3/2})$
diff3	$\frac{3}{16}\sqrt{2\pi}n^{5/2} + O(n^2)$
diff4	$\frac{1}{2}n^3 + O(n^{5/2})$
diff5	$\frac{15}{64}\sqrt{2\pi}n^{7/2} + O(n^3)$
diff6	$\frac{3}{4}n^4 + O(n^{7/2})$
diff7	$\frac{105}{256}\sqrt{2\pi}n^{9/2} + O(n^4)$



(The graph on the right hand side shows the logarithm of the complexity of the kth order derivative, for $1 \le k \le 7$). These figures gave us the idea to conjecture an average cost of

$$\frac{\Gamma(k/2+1)}{2^{k/2}}n^{k/2+1} + O(n^{(k+1)/2}) \tag{1}$$

for the kth order differentiation. The equation (1) is indeed the correct expansion. To prove this, we translate the body of diff_k into an equation for its cost generating function according to the rules given in [2]²:

$$\begin{aligned}\tau\mathit{diff}_k(z) &= zE(z)^2 + 2zE(z)\tau\mathit{diff}_k(z)\\ &+ (2^{k+1}-1)zE(z)^2 + zE(z)\sum_{i=0}^{k}\binom{k}{i}(\tau\mathit{diff}_i(z)+\tau\mathit{diff}_{k-i}(z))\\ &+ 3z.\end{aligned}$$

The right hand side in the first line reflects the kth order derivative of a sum `plus(e1,e2)`: the term $zE(z)^2$ corresponds to the atom `plus`, and the other term to the recursive calls `diff_k(e1)` and `diff_k(e2)`. The kth order derivative of a product `times(e1,e2)` looks like a complete binary tree of height $k+1$, whose nodes of depth 0 to $k-1$ are `plus` atoms, whereas the level of depth k contains only `times` atoms, and the 2^{k+1} leaves are all kth compositions of either `diff` or `copy` on either `e1` or `e2`. The first term in the second line is the cost of writing all $2^{k+1}-1$ internal nodes; the second term is the cost of the recursive calls, with the convention that $\tau\mathit{diff}_0 = \tau\mathit{copy}$. From the above equation, we derive the following recurrence relation for $\tau\mathit{diff}_k$:

$$\tau\mathit{diff}_k(z) = \frac{3z + 2^{k+1}zE(z)^2 + 2zE(z)\sum_{i=0}^{k-1}\binom{k}{i}\tau\mathit{diff}_i(z)}{1-4zE(z)}. \tag{2}$$

To get an asymptotic expansion of the coefficient of z^n in $\tau\mathit{diff}_k(z)$, we first have to compute a local expansion around its main singularities, here the roots of $24z^2 = 1$. For $k \geq 1$, the dominant contribution comes from the term corresponding to $i = k-1$ in the sum of (2), thus we get a simple recurrence leading to

$$\tau\mathit{diff}_k(z) \sim \sqrt{6}\frac{k!}{2^{k+1}}\frac{1}{(1-24z^2)^{\frac{k+1}{2}}}.$$

We transfer this expansion into coefficients, using the relation³ $[z^n](1-z)^{-\alpha} \sim n^{\alpha-1}/\Gamma(\alpha)$, and we divide by the asymptotic expansion of the number of expressions, to get the following average cost

$$\tau\overline{\mathit{diff}}_k(z) \sim \sqrt{\pi}\frac{k!}{2^{3k/2}\Gamma(\frac{k+1}{2})}n^{k/2+1}.$$

This expansion is in fact the same as (1) because $2^k\Gamma(k/2+1)\Gamma(k/2+1/2) = k!\sqrt{\pi}$. This first example shows how powerful the expansion process is: starting from a program with composition of length $O(k)$ (the kth order derivation), it produces a program without composition of length $\Omega(2^k)$!

²If $\mathcal{A}$ 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 taking inputs in $\mathcal{A}$, the (cost) generating function 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 $z=0$.

³As usually, $[z^n]f(z)$ denotes the coefficient of z^n in the Taylor expansion of f around $z=0$.

3.2 Regular languages and the Collatz conjecture

In this section, we show that function composition, used jointly with analysis of functions with a finite number of return values [6], helps to compute grammars of sets derived from regular languages. To illustrate this, we take as example the Collatz conjecture: "starting from a positive integer, the iteration of the function

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even,} \\ 3n+1 & \text{if } n \text{ is odd,} \end{cases} \tag{3}$$

ultimately reaches 1". For example, we obtain the following chain for the number 13:

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

In [4], David Wilson introduced the sets S_k, where the index k denotes the number of times the function $3n+1$ is applied before 1 is reached. In the above example, the function $3n+1$ is applied two times (from 13 to 40 and from 5 to 16), thus 13 belongs to S_2. The first sets begin like this:

$$\begin{aligned} S_0 &= \{1,2,4,8,16,32,64,128,256,512,\ldots\} \\ S_1 &= \{5,10,20,21,40,42,80,84,85,160,\ldots\} \\ S_2 &= \{3,6,12,13,24,26,48,52,53,96,\ldots\} \\ S_3 &= \{17,34,35,68,69,70,75,136,138,140,\ldots\}. \end{aligned}$$

D. Wilson has shown that the base-two string expressions of any S_k form a regular language (accepted by a finite automaton and writable as a regular expression). For instance,

$$\begin{aligned} S_0 &\to 1\ 0^* \\ S_1 &\to 101\ (01)^*\ 0^*. \end{aligned}$$

This result implies that the number of n-bit integers in S_k is easily computable: it is the coefficient of z^n in a rational function derived from the regular expression of S_k, for example $z^3/(1-z^2)/(1-z)$ for S_1.

Function composition will enable us to compute a grammar for S_k *automatically*, with a description file of *linear* length with respect to k. From this grammar, we easily derive a regular expression. At the end of this section, we obtain a regular expression for S_2 and S_3.

Let us introduce the function g dividing its input by two as long as possible, then applying *one time* the function $3n+1$:

$$g(n) = \begin{cases} g(n/2) & \text{if } n \text{ is even,} \\ 0 & \text{if } n = 1, \\ 3n+1 & \text{otherwise.} \end{cases} \tag{4}$$

We have for instance $g(13) = 40$, $g(40) = 16$ and $g(16) = 0$, and the function g gives a characterization of S_k:

$$S_k = \{n \mid g^{(k)}(n) \text{ is a power of two }\}. \tag{5}$$

Therefore to construct an ADL program recognizing integers in S_k, we have to encode the function g, and a function recognizing powers of two. For this purpose, we represent integers in base two:

```
type integer = nil | bit integer;
     bit = zero | one;
     zero, one = atom(1);
     nil = atom(0);
```

The function g is written using a function called `three_x_plus_1`, whose input is the base-two representation of an integer n, and which outputs the base-two representation of $3n + 1$:

```
function three_x_plus_1 (i : integer) : integer;
begin
   case i of
      nil : product(one,nil);
      (zero,j) : product(one,three_x(j));
      (one,j) : product(zero,three_x_plus_2(j));
   end;
end;
```

The other functions `three_x` and `three_x_plus_2` are defined similarly. With the functions g and `is_a_power_of_two`, according to equation (5), we write the function `is_in_S3` to recognize integers in S_3:

```
function g (i : integer) : integer;
begin
   case i of
      nil : nil;
      (zero,j) : g(j);
      (one,nil) : nil;
      otherwise : three_x_plus_1(i);
   end
end;

function is_a_power_of_two (i : integer) : boolean;
begin
   case i of
      nil : false;                              % 0 is not a power of 2 %
      (zero,j) : is_a_power_of_two(j);
      (one,j) : is_zero(j)
   end;
end;
```

```
function is_in_S3 (i : integer) : boolean;
begin
   is_a_power_of_two(g(g(g(i))))
end;
```

With these functions, we could compute automatically the probability for an integer of n bits to be in S_3, with a procedure like this:

```
procedure main (i : integer);
begin
   if is_in_S3(i) then count
end;
measure count : 1;
```

But our goal is to get a regular expression for S_k like those obtained by Wilson for S_0 and S_1. During the analysis of the procedure `main`, the system prints some messages, for example (among other lines):

```
Computing composition of is_a_power_of_two and g : f2
Computing composition of f2 and g : f8
Computing composition of f8 and g : f26
Introducing the new type T84 for which function f26 returns true
Introducing the new type T142 for which function f26 returns false
```

At the first line, the system computes the body of the composition of `is_a_power_of_two` and g, which is the new function `f2`. The message `Computing composition of f and g` means that the expansion process of theorem 1 is currently being applied. Thus the body of `f2` contains no composition, and looks like the body of `diff_of_diff` in section 3.1. Then it computes the composition of `f2` and g, and calls it `f8` (second line). It also computes the composition of `f8` with g, namely `f26`, which is therefore equivalent to `is_in_S3`.

At this stage, we have constructed a set of ADL functions without any composition, containing the function `f26` equivalent to `is_in_S3`. For such a set, it is possible to derive automatically a grammar of the data structures for which each function with a finite number of possible outputs (in particular a boolean function like `f26`) returns a given value [6]. For example, as explained by the last lines in the above messages, the system introduced two new data types `T84` and `T142`, which stand for the integers in S_3 and not in S_3 respectively. Like for the expansion process, a complete grammar for `T84` and `T142` was in fact generated, starting from the grammar of the type `integer`.

Due to the form of the rules used (cf [6]), this grammar is unambiguous because so was the grammar of `integer`. The raw grammar we get has 58 non-terminals, among them 27 do not derive any finite string. After some simplifications by hand (they took longer than the automatic construction of the grammar!), we got the following regular expression for S_3:

$$S_3 \rightarrow ((\epsilon \mid (1001011110110100000)^* 1001011\ (\epsilon \mid 1 \mid 1101 \mid 110110011 \mid 11011010000 \mid 1101101000011))\ (100011)^* 1000 \mid (1001011110110100000)^* 100101\ (\epsilon \mid 11101100 \mid 1110110100000))(\epsilon \mid 1)\ (10)^* 10^*.$$

Similarly, we computed with the help of the Lambda-Upsilon-Omega system the following regular expression of the set S_2, starting from a grammar with 22 non-terminals:

$$S2 \rightarrow (1 \mid 11100\ (011100)^*\ (0 \mid 01))\ (10)^*\ 1\ 0^*$$

4 Conclusion

We have shown that some kinds of function compositions are well suited for an automatic average case analysis. The main idea is the following: a program including compositions first translates into an similar program without composition (*expansion process* of Theorem 1), then this last program is analyzed by already known techniques [2].

Composition of functions is not only useful in the description of algorithms, but in some cases it is *necessary* to use it, otherwise the description would be too long, as the examples of Section 3 prove it. In these cases, the long description is generated by the computer, therefore it contains no error (we hope it!).

References

[1] 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.

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

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

[4] D. W. Wilson. Transaction ⟨1778@cvbnetPrime.COM⟩ of usenet.sci.math, 1991.

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

[6] P. Zimmermann. Analysis of functions with a finite number of return values. Research Report 1625, Institut National de Recherche en Informatique et en Automatique, 1992.

[7] 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.