13

Function Composition and Automatic Average-Case Analysis

Paul Zimmermann
INRIA, Rocquencourt

[summary by Paul Zimmermann]

This talk introduces the composition of functions defined over extended context-free languages. It is shown 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.

In the field of automatic complexity analysis, the length of the problem description is often a limitation: writing a long specification program is often a difficult error prone process. Thus one needs some powerful constructs to describe algorithms, with the necessary constraint that these constructs allow an *automatic* analysis.
One is interested here in the average case analysis of programs including some *compositions* of functions. None of the existing systems, including METRIC [5], COMPLEXA [10] or the version of Lambda-Upsilon-Omega ($\Lambda\Upsilon\Omega$) described in [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 one has written a function *diff* performing the differentiation of symbolic expressions with respect to one variable, and now one would like to analyze the two-fold differentiation by just defining

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

instead of having to write the entire body of the function *diff2*. P. Zimmermann shows that this shorthand is possible: he defines 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 us to define and to analyze large problems by short description programs.

1 A class of programs with composition

This section introduces the composition of functions in the ADL language (Algorithm Description Language), especially designed for automatic average case analysis in the Lambda-Upsilon-Omega system [1, 2, 7]. The following is an ADL program performing the differentiation of symbolic expressions.

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

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. To define the complexity measure 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;
```

If one analyzes the *diff* function in the Lambda-Upsilon-Omega system, one gets the following average cost for expressions of size n:

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

Now one allows 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 (the relation "depends on" is the reflexive and transitive closure of the relation "has in its body").*

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

The proof of this theorem involves the definition of a way of expanding the composition, the *expansion process*. This process necessarily terminates when the composition graph is acyclic.
As the average case analysis of ADL programs *without* composition is already known to be automatic [2, 7], the above theorem implies directly the following result:

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

2 Two non-trivial examples

This section presents two research problems where the implementation of the expansion process on a computer allowed P. Zimmermann to discover some results which would have been very difficult to find by hand.

2.1 Analysis of kth order differentiation

The expansion process has been encoded in the version V1.4 of the system Lambda-Upsilon-Omega. When one analyzes the function `diff2` as `diff o diff`, 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 also 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 each time one more call to the function `diff`, P. Zimmermann was able to analyze the k-fold iterated differentiation until $k = 7$, and obtained the following figures.

	average cost	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`	$\frac{105}{256}\sqrt{2\pi}n^{9/2} + O(n^4)$		

These figures led 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, that was proved to be correct afterwards.

2.2 Regular languages and the Collatz conjecture

This example shows that function composition, used jointly with analysis of functions with a finite number of return values [8], helps to compute grammars of sets derived from regular languages. The Collatz conjecture says: "starting from a positive integer, the iteration of the function

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

ultimately reaches 1". For example, one obtains 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 [6], 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:

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

Wilson and Shallit proved[4] in [4] that sets S_k are 2-automatic, that is the base-two string expressions of each S_k form a regular language (accepted by a finite automaton and writable as a regular expression). For instance, $S_0 \to 1\ 0^*$ and $S_1 \to 101\ (01)^*\ 0^*$. 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 enables one to compute a grammar for S_k *automatically*, with a description file of *linear* length with respect to k. From this grammar, one easily derives a regular expression. At the end of this section, such regular expressions for S_2 and S_3 are given. The idea is to define the function g dividing its input by two as long as possible, then applying *one time* the function $3n+1$:

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

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

$$S_k = \{n \mid g^{(k)}(n) \text{ is a power of two }\}. \tag{2}$$

Therefore to construct an ADL program recognizing integers in S_k, one has to encode the function g, and a function recognizing powers of two. For this purpose, integers are encoded 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 $3n+1$:

[4] This result could also be deduced from the theory of *sequential transductors* [3, example 8 page 123].

```
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 (2), one writes 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;
        (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;
```

During the analysis of this whole program, the $\Lambda\Upsilon^{\Omega}$ 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 this stage, it has 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 [8]. 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 [8]), this grammar is unambiguous because so was the grammar of `integer`. The raw grammar one gets 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!), P. Zimmermann got the following regular expression for S_3:

$$\begin{aligned} S_3 \quad \rightarrow \quad & ((\epsilon \mid (100101111011010000)^*1001011\ (\epsilon \mid 1 \mid 1101 \mid 110110011 \mid 11011010000 \mid 11011010000011)) \\ & (100011)^*1000 \mid (100101111011010000)^*100101\ (\epsilon \mid 11101100 \mid 1110110100000))(\epsilon \mid 1)\ (10)^*10^*. \end{aligned}$$

Similarly, he 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:

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

Conclusion. This research shows 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 by an expansion process, 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 two examples presented here prove it. In these cases, the long description is automatically generated by the computer, therefore it contains no error.

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] J. E. Pin. *Variétés de langages formels*. Études et recherches en informatique. Masson, 1984.

[4] J. Shallit and D. Wilson. The "$3x+1$" Problem and Finite Automata. *Bulletin of the EATCS*, 46:182–185, 1992.

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

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

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

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

[9] P. Zimmermann. Function composition and automatic average case analysis. In P. Leroux and C. Reutenauer, editors, *Séries formelles et combinatoire algébrique*, volume 11 of *Publications du LACIM, Université du Quebec à Montréal*, pages 477–486, 1992. Proceedings of the 4th Colloquium. To appear in *Discrete Mathematics*.

[10] 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.