The Complexity of Euler's Integer Partition Theorem

Cristian S. Calude,[1] **Elena Calude**[2]

[1]University of Auckland, NZ
[2]Massey University at Auckland, NZ

CDMTCS-409
November 2011

THE COMPLEXITY OF EULER'S INTEGER PARTITION THEOREM

CRISTIAN S. CALUDE AND ELENA CALUDE

Abstract. Euler's integer partition theorem stating that *the number of partitions of an integer into odd integers is equal to the number of partitions into distinct integers* ranks 16 in Wells' list of the most beautiful theorems [17]. In this paper we use the algorithmic method to evaluate the complexity of mathematical statements developed in [3, 4, 5] to show that Euler's theorem is in class $\mathfrak{C}_{U,7}$, the most complex mathematical statement studied to date.

1. Euler's integer partition theorem

The number of ways of writing the integer as a sum of n positive integers, where the order of addends is ignored, is denoted by $P(n)$. By $P(n|\text{odd parts})$ and $P(n|\text{distinct parts})$ we denote the number of ways of writing the integer as a sum of n odd/distinct positive integers. By convention, partitions are usually ordered from largest to smallest. Some examples are presented in the following table:

n	odd parts	$P(n\|\text{odd parts})$	distinct parts	$P(n\|\text{distinct parts})$
1	1	1	1	1
2	$1+1$	1	2	1
3	$1+1+1$		3	
	3	2	$2+1$	2
4	$1+1+1+1+1$		4	
	$3+1$	2	$3+1$	2
5	$1+1+1+1+1$		5	
	$3+1+1$		$4+1$	
	5	3	$3+2$	3
6	$1+1+1+1+1+1$		6	
	$3+1+1+1$		$5+1$	
	$3+3$		$4+2$	
	$5+1$	4	$3+2+1$	4

Table 1. Integer partitions into odd integers vs.
integer partitions into distinct integers

Leonhard Euler is credited (cf. [1] p.2) to have proved in 1748 the theorem bearing his name: no matter how long we extend the Table 1, there will always be as many items in the left column as in the right one. In other terms, *the number of partitions of an integer into odd integers is equal to the number of partitions into distinct integers.*

Euler's integer partition theorem is a Π_1–statement, i.e. a statement of the form $\forall n\, P(n)$, where $P(n)$ is a unary computable predicate, hence its complexity can be evaluated with the method developed in [3, 4, 5].

2. The complexity measure

The complexity measure for Π_1-statements is defined by means of register machine programs which implement a universal self-delimiting Turing machine U. The machine U (which is fully described in [5]) has to be *minimal* in the sense that none of its instructions can be simulated by a program for U written with the remaining instructions.

To every Π_1–problem $\pi = \forall m P(m)$ we associate the algorithm $\Pi_P = \inf\{n : P(n) = \text{ false}\}$ which systematically searches for a counter-example for π. There are many programs (for U) which implement Π_P; without loss of generality, any such program will be denoted also by Π_P. Note that π is true iff $U(\Pi_P)$ never halts.

Motivated by Occam's Razor principle of parsimony we define the complexity (with respect to U) of a Π_1–problem π to be the length of the smallest-length program (for U) Π_P—defined as above—where minimisation is calculated for all possible representations of π as $\pi = \forall n P(n)$:[1]

$$C_U(\pi) = \min\{|\Pi_P| \,:\, \pi = \forall n P(n)\}.$$

Because the complexity C_U is incomputable, we work with upper bounds for C_U. As the exact value of C_U is not important, following [5] we classify Π_1–problems into the following classes:

$$\mathfrak{C}_{U,n} = \{\pi \,:\, \pi \text{ is a } \Pi_1\text{–problem}, C_U(\pi) \leq n \text{ kbit}\}.$$

3. A universal prefix-free binary Turing machine

We briefly describe the syntax and the semantics of a register machine language which implements a (natural) minimal universal prefix-free binary Turing machine U; it is a refinement, constructed in [5], of the languages in [3].

Any register program (machine) uses a finite number of registers, each of which may contain an arbitrarily large non-negative integer.

By default, all registers, named with a string of lower or upper case letters, are initialised to 0. Instructions are labeled by default with 0,1,2,...

The register machine instructions are listed below. Note that in all cases R2 and R3 denote either a register or a non-negative integer, while R1 must be a register. When referring to R we use, depending upon the context, either the name of register R or the non-negative integer stored in R.

=R1,R2,R3

If the contents of R1 and R2 are equal, then the execution continues at the R3-th instruction of the program. If the contents of R1 and R2 are not equal, then execution continues with the next instruction in sequence. If the content of R3 is outside the scope of the program, then we have an illegal branch error.

&R1,R2

[1]For C_U it is irrelevant whether π is known to be true or false. In particular, the program containing the single instruction halt is not a Π_P program, for any P.

The contents of register R1 is replaced by R2.

+R1,R2

The contents of register R1 is replaced by the sum of the contents of R1 and R2.

!R1

One bit is read into the register R1, so the contents of R1 becomes either 0 or 1. Any attempt to read past the last data-bit results in a run-time error.

%

This is the last instruction for each register machine program before the input data. It halts the execution in two possible states: either successfully halts or it halts with an under-read error.

A *register machine program* consists of a finite list of labeled instructions from the above list, with the restriction that the halt instruction appears only once, as the last instruction of the list. The input data (a binary string) follows immediately after the halt instruction. A program not reading the whole data or attempting to read past the last data-bit results in a run-time error. Some programs (as the ones presented in this paper) have no input data; these programs cannot halt with an under-read error.

The instruction `=R,R,n` is used for the unconditional jump to the n-th instruction of the program. For Boolean data types we use integers $0 =$ `false` and $1 =$ `true`.

For longer programs it is convenient to distinguish between the main program and some sets of instructions called "routines" which perform specific tasks for another routine or the main program. The call and call-back of a routine are executed with unconditional jumps.

4. Binary coding of programs

In this section we develop a systematic efficient method to uniquely code in binary the register machine programs. To this aim we use a prefix-free coding as follows.

The binary coding of special characters (instructions and comma) is the following (ε is the empty string):

special characters	code	special characters	code
,	ε	+	111
&	01	!	110
=	00	%	100

Table 2. Special characters

For registers we use the prefix-free code $\text{code}_1 = \{0^{|x|}1x \mid x \in \{0,1\}^*\}$. The register names are chosen to optimise the length of the program, i.e. the most frequent registers have the smallest code_1 length. For non-negative integers we use the prefix-free code $\text{code}_2 = \{1^{|x|}0x \mid x \in \{0,1\}^*\}$. The instructions are coded by self-delimiting binary strings as follows:

(1) `& R1,R2` is coded in two different ways depending on R2:[2]

$$01\mathrm{code}_1(\mathrm{R1})\mathrm{code}_i(\mathrm{R2}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

(2) `+ R1,R2` is coded in two different ways depending on R2:

$$111\mathrm{code}_1(\mathrm{R1})\mathrm{code}_i(\mathrm{R2}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

(3) `= R1,R2,R3` is coded in four different ways depending on the data types of R2 and R3:

$$00\mathrm{code}_1(\mathrm{R1})\mathrm{code}_i(\mathrm{R2})\mathrm{code}_j(\mathrm{R3}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer, $j = 1$ if R3 is a register and $j = 2$ if R3 is an integer.

(4) `!R1` is coded by

$$110\mathrm{code}_1(\mathrm{R1}).$$

(5) `%` is coded by

$$100.$$

All codings for instruction names, special symbol comma, registers and non-negative integers are self-delimiting; the prefix-free codes used for registers and non-negative integers are disjoint. The code of any instruction is the concatenation of the codes of the instruction name and the codes (in order) of its components, hence the set of codes of instructions is prefix-free. The code of a program is the concatenation of the codes of its instructions, so the set of codes of all programs is prefix-free too.

5. The counting algorithm

We use the algorithm 7 in [15], p. 13, which generates all integer partitions: for each of them we test whether the partition uses odd or distinct integers, and count accordingly. Other possible algorithms for generating all integer partitions are discussed in [13, 14, 16].

6. Routines

There are two types of routines: a) 1-routines, that is routines that do not use any other routines, b) 2-routines, that is routines that call other routines. All unary routines use the register `a` for input and all binary routines use `a` and `b` for input, they all use `c` for keeping track of returning to the calling environment and `d` for storing the result. As the values in the registers `a`, `b`, and `c` have to be unchanged on exit from any routine, a copying-restoring-initial-values process takes place inside some routines.

As the registers used in the following programs are shared between the main program, 1-routines and 2-routines care must be taken so the content of a register is not changed inadvertently. There are various ways to deal with this problem. One is to reserve the letters from `a` to `h` to 1-routines and to use `aa`, `ab`, `ac`, ... in 2-routines or the main program (see [12]). The approach used in the following examples is that 1-routines use single letters name, 2-routines use double letter

[2]As $x\varepsilon = \varepsilon x = x$, for every string $x \in \{0,1\}^*$, in what follows we omit ε.

names, where the second letter comes from the first letter of the routine, and the main program uses capital letters for registers. As all 2-routines have different first-letter-names, there is no danger of using the same register in a routine that calls another routine using the same register name. The upside of this approach is the guarantee that the values in registers are the correct ones; the downside is that the number of necessary registers is bigger (this fact can be mitigated at the end by safely reusing a few registers).

The multiplication routine MUL takes as input two non-negative integers, stored in registers `a` and `b`, and produces its result `a*b` in register `d`:

label	instruction
MUL:	`&d, 0`
	`&e, 0`
LM1:	`=e, b, c`
	`+d, a`
	`+e, 1`
	`=a, a, LM1`

Table 3. Multiplication

The compare routine CMP takes as input two non-negative integers, stored in registers `a` and `b`, and produces its result in register `d` according to the formula: $d = \mathrm{CMP}(a, b) = 1$ if $a < b$, 0 if $a = b$, and 2 if $a > b$:

label	instruction
CMP:	`&d, 0`
	`=a ,b LCP3`
	`&d, 1`
	`&e, 0`
LCP1:	`=a, e, LCP3`
	`=b, e, LCP2`
	`+e, 1`
	`=a, a, LCP1`
LCP2:	`+d, 1`
LCP3:	`=a, a, c`

Table 4. Comparison

The subtraction routine SUBT takes two non-negative integers, stored in registers `a` and `b` with `a` $\geq$ `b`, as input and produces its result `a-b` in register `d`:

label	instruction
SUBT:	`&e, b`
	`&d, 0`
LS1:	`=e, a, c`
	`+e, 1`
	`+d, 1`
	`=a, a, LS1`

Table 5. Subtraction

The function $T(a) = \max\{t \mid t(t+1) \leq 2a\}, 0 \leq t \leq a$, is implemented by the routine TFC, which uses the 1-routines multiplication, subtraction and compare:

label	instruction	label	instruction	label	instruction
TFC	`&at, a`		`&b, ft`	LT4	`&c, LT5`
	`&bt, b`		`+b, 1`		`&a, ft`
	`&ct, c`		`=a, a, MUL`		`&b, 1`
	`&ft, 0`	LT2	`&a, d`		`=a, a, SUBT`
	`&d, 0`		`&c, LT3`	LT5	`&a, at`
	`=a, 0, c`		`&b, et`		`&b, bt`
	`&et, a`		`=a, a, CMP`		`&c, ct`
	`+et, a`	LT3	`=d, 2, LT4`		`=a, a, c`
LT1	`&c, LT2`		`+ft, 1`		
	`&a, ft`		`=a, a, LT1`		

Table 6. Function T

The routine KFC encodes the function $K(a) = a - T(a)T(a+1)/2, a \geq 0$:

label	instruction	label	instruction	label	instruction
KFC	`&bk, b`		`&b, fk`		`=fk, 2, LK5`
	`&ck, c`		`=a, a, MUL`		`=a, a, LK4`
	`&ak, a`	LK3	`&ek, d`	LK5	`&fk, 0`
	`&c, LK1`		`&b, 0`		`+b, 1`
	`=a, a, TFC`		`=ek, 0, LK6`		`=a, a, LK4`
LK1	`&ek, d`		`=ek, 1, LK6`	LK6	`&a, ak`
	`&c, LK2`		`&b, 1`		`&c, LK7`
	`+a, 1`		`&fk, 0`		`=a, a, SUBT`
	`=a, a, TFC`		`&gk, 2`	LK7	`&b, bk`
LK2	`&fk, d`	LK4	`=ek, gk, LK6`		`&c, ck`
	`&c, LK3`		`+gk, 1`		`=a, a, c`
	`&a, ek`		`+ fk 1`		

Table 7. Function K

The routine LFC encodes the function $L(a) = T(a) - K(a), a \geq 0$:

label	instruction	label	instruction	label	instruction
LFC	`&al, a`	LL1	`&b, d`		`=a, a, SUBT`
	`&bl, b`		`&c, LL2`	LL3	`&a, al`
	`&cl, c`		`= a, a, TFC`		`&b, bl`
	`&c, LL1`	LL2	`&a, d`		`&c, cl`
	`=a, a, KFC`		`&c, LL3`		`=a, a, c`

Table 8. Function L

7. Register machine language implementation of arrays

In this section we present a method of implementation in the register machine language for arrays using Cantor's bijection which maps (codes) a pair of non-negative integers a, b into a single non-negative integer $\langle a, b\rangle = (a+b)(a+b+1)/2 + a$.This function can be iterated to a bijection $\langle a_1, a_2, \ldots, a_k\rangle$ between $\mathbf{N}^k$ and $\mathbf{N}$, for every $k > 1$; we shall adopt, by convention, the left-associative iteration.

For example, to work with arrays in register machine programs we need to code (finite) sequences of non-negative integers into a single non-negative integer. In what follows we use Cantor's bijection for such codings; for a different coding see [12].

For example the 4-element sequence $[2, 1, 1, 0]$ is encoded by 1484 as $\langle\langle\langle 2, 1\rangle, 1\rangle, 0\rangle = 1484$. The reverse process allows to convert, for each given $k \geq 1$, any non-negative integer into a unique k-element sequence of non-negative integers. For example, the number 5564 can be converted to the 4-element sequence $[3, 1, 0, 0]$ and the 2-element sequence $[104, 0]$.

We can compute Cantor's bijection using the formula $\langle a, b\rangle = 1 + 2 + \cdots + (a + b) + a$:

label	instruction
CFC:	`&e, a`
	`+e, b`
	`&d, 0`
	`=e, 0, LCF2`
	`&f, 1`
LCF1:	`+d, f`
	`=e, f, LCF2`
	`+f, 1`
	`=a, a, LCF1`
LCF2	`+d, a`
	`=a, a, c`

Table 9: Cantor's bijection

We are now using the Cantor's bijection for the routine ELM encoding the function $ELM(A, b, N) =$ the bth element in the N-element array representation of A:

label	instruction	label	instruction	label	instruction
ELM	`&ae, a`		`&a, A`		`&a, d`
	`be, b`		`&ge, 1`	LE3	`&c, LE4`
	`&ce, c`		`&c, LE10`		`=a, a, LFC`
	`&a, A`		`=a, a, KFC`	LE4	`&a, ae`
	`=N, 2, LE1`	LE10	`=he, 1, LE8`		`&c, ce`
	`=b, 1, LE3`	LE13	`+ge, 1`		`=a, a, c`
	`&c, LE6`		`&a, d`	LE5	`=a, a, LE3`
	`&a, b`		`&c, LE12`	LE1	`=N, b, LE2`
	`&b, 1`		`=a, a, KFC`		`=a, a, LE3`
	`=a, a, SUBT`	LE12	`=he, ge, LE8`	LE2	`&c, LE4`
LE6	`&he, d`		`=a, a, LE13`		`=a, a, KFC`
	`&b, be`	LE8	`=N, b, LE4`		

Table 10. Function ELM

The routine RPL uses the iteration of Cantor's bijection to encode the function which replaces the ath element of the array A by b, $RPL(A, a, b, N) = \langle x_N, x_{N-1}, \ldots, x_{a+1}, b, x_{a-1}, \ldots, x_1\rangle$, where $A = \langle x_N, x_{N-1}, \ldots, x_a, \ldots, x_1\rangle$:

label	instruction	label	instruction	label	instruction
RPL	`&ar, a`		`=a, a, SUBT`	LR6	`&dr, d`
	`&br, b`	LR2	`&er, d`		`=a, a, LR9`
	`&cr, c`		`=er, 0, LR8`	LR3	`&fr, b`
	`&er, N`		`=er, a, LR3`		`=a, a, LR5`
	`=a, N, LR7`		`&c, LR4`	LR7	`&dr, b`
	`&b, N`		`&b, er`		`=a, a, LR9`
	`&c, LR1`		`=a, a, ELM`	LR8	`&d, dr`
	`= a, a, ELM`	LR4	`&fr, d`		`&a, ar`
LR1	`&dr, d`	LR5	`&c, LR6`		`&b, br`
LR9	`&c, LR2`		`&a, dr`		`&c, cr`
	`&a, er`		`&b, fr`		`=a, a, c`
	`&b, 1`		`=a, a, CFC`		

Table 11. Function *RPL*

8. The program $\Pi_{\text{IntegerPartition}}$

We are now ready to present the main program $\Pi_{\text{IntegerPartition}}$.

label	instruction	label	instruction	label	instruction
MAIN	`&N, 2`		`+I, 1`		`&c, L44`
L1	`&P, 0`		`&J, N`		`=a, a, SUBT`
	`&R, 0`		`+J, 1`	L44	`&a, I`
	`&I, N`		`=I, J, L2`		`&b, d`
	`&A, N`		`=a, a, L25`		`&c, L45`
L11	`&a, I`	L28	`&a, H`		`=a, a, RPL`
	`&b, 1`		`&b, 1`	L45	`&A, d`
	`&c, L9`		`&c, L29`		`&a, I`
	`=a, a, SUBT`		`=a, a, SUBT`		`&b, 1`
L9	`&I, d`	L29	`&a, I`		`&c L46`
	`=I, 0, L2`		`&b, d`		`=a, a, SUBT`
	`&a, A`		`&c, L30`	L46	`&a, d`
	`&b, 0`		`=a, a, RPL`		`&b, 1`
	`&c, L10`	L30	`&A, d`		`&c, L47`
	`=a, a, CFC`		`&a, I`		`=a, a, RPL`
L10	`&A, d`		`&b, 1`	L47	`&A, d`
	`=a, a, L11`		`&c, L31`		`=a, a, L2`
L2	`&I, 1`		`=a, a, SUBT`	L48	`+I, 1`
L17	`&J, I`	L31	`&a, d`		`& J N`
	`+J, 1`		`&J, V`		`+J, 1`
L16	`&b, I`		`+J, 1`		`=I, J, L2`
	`&c, L12`		`&b, J`		`=a, a, L42`
	`=a, a, ELM`		`&c, L32`	L4	`&I, 1`
L12	`&H, d`		`=a, a, RPL`	L49	`&b, I`
	`&b, J`	L32	`&A, d`		`&c, L53`

label	instruction	label	instruction	label	instruction
	`&c, L13`		`&J, 0`		`=a, a, ELM`
	`=a, a, ELM`	L34	`+J, 1`	L53	`& H d`
L13	`&G, d`		`=J, I, L2`		`=H, 0, L50`
	`&a, H`		`&a, J`		`=H, 1, L50`
	`&b, G`		`&b, 0`		`&F, 0`
	`&c, L14`		`&c, L33`	L51	`&E, 0`
	`=a, a, CMP`		`=a, a, RPL`		`=H, F, L52`
L14	`=d, 2, L3`	L33	`&A, d`		`+F, 1`
	`+J, 1`		`=a, a, L34`		`+E, 1`
	`&V, N`	L23	`&I, 1`		`=E, 2, L51`
	`+V, 1`	L35	`&c, L36`		`=a, a, L52`
	`=J, V, L15`		`&b, I`	L54	`= E, 0, L6`
	`=a, a, L16`		`=a, a, ELM`	L50	`+I, 1`
L15	`+I, 1`	L36	`&H, d`		`&J, N`
	`=I, N, L4`		`=H, 1, L37`		`+J, 1`
	`=a, a, L17`		`+I, 1`		`=I, J, L7`
L3	`&V, 0`		`=a, a, L35`		`=a, a, L49`
	`&I, 1`	L37	`&a, I`	L6	`&I, 1`
L21	`&b, I`		`&b, 2`	L55	`&b, I`
	`&c, L18`		`&c, L38`		`&c, L56`
	`=a, a, ELM`		`=a, a, RPL`		`=a, a, ELM`
L18	`&H, d`	L38	`&A, d`	L56	`&H, d`
	`=H, 0, L19`		`+I, 1`		`=H, 0, L58`
	`=H, 1, L20`		`&b, I`		`&J, I`
	`=a, a, L8`		`&c, L39`		`+J, 1`
L20	`+V, 1`		`=a, a, ELM`		`&b, J`
L19	`+I, 1`	L39	`&a, d`		`&c, L57`
	`&J, N`		`&b, 1`		`=a, a, ELM`
	`+J, 1`		`&c, L40`	L57	`&G, d`
	`=I, J, L8`		`=a, a, SUBT`		`=H, G, L3`
	`=a, a, L21`	L40	`&b, d`		`+I, 1`
L8	`=V, 0, L22`		`&a, I`		`=I, N, L5`
	`=V, 1, L23`		`&c, L41`		`=a, a, L55`
	`=V, N, L24`		`=a, a, RPL`	L58	`+I, 1`
	`&I, 1`	L41	`&A, d`		`=a, a, L55`
L25	`&b, I`		`=a, a, L2`	L7	`+P, 1`
	`&c, L26`	L22	`&I, 1`		`=a, a, L6`
	`=a, a, ELM`	L42	`&b, I`	L5	`+R, 1`
L26	`&H, d`		`&c, L43`		`=a, a, L3`
	`&c, L27`		`=a, a, ELM`	L24	`=P, R, L59`
	`&a, H`	L43	`&H, d`		`=a, a, L60`
	`&b, 1`		`=H, 0, L48`	L59	`+N, 1`
	`=a, a, CMP`		`&a, H`		`=a, a, L1`
L27	`=d, 2, L28`		`&b, 1`	L60	`STOP`

Table 12. The program $\Pi_{\text{IntegerPartition}}$

The register machine program for Euler's integer partition theorem consists of 389 instructions having a total length of 6,417 bits, hence it is in $\mathfrak{C}_{U,7}$. In this way this theorem is the more complex problem studied to date: the Riemann hypothesis is in $\mathfrak{C}_{U,3}$ [10] and the four colour theorem is in $\mathfrak{C}_{U,4}$ [6]. The most likely reason for this high complexity reason is the complexity of processing arrays in the register machine language (see [12] for a different approach).

9. Final comments

Occam's Razor principle motivates the complexity of Π_1–statements used in this paper. The same principle leads, under some general assumptions, to a learning algorithm which produces hypotheses that with high probability will be predictive of future observations [2]. Is there any relation between the complexity of Π_1–statement and its learnability? Can the use of different algorithms (see [13, 14, 16]) and codifications for arrays (see [12]) reduce the estimated complexity of the Euler's integer partition theorem?

References

[1] G. E. Andrews, K. Eriksson. *Integer Partitions*, Cambridge University Press, 2004.

[2] A. Blumer, A. Ehrenfeucht, D. Haussler, M. K. Warmuth. Occam's Razor, *Information Processing Letters* 24, 6 (1987), 377–380.

[3] C. S. Calude, E. Calude, M. J. Dinneen. A new measure of the difficulty of problems, *Journal for Multiple-Valued Logic and Soft Computing* 12 (2006), 285–307.

[4] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 1 *Complex Systems*, 18-3 (2009), 267–285.

[5] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 2 *Complex Systems*, 18-4, (2010), 387–401.

[6] C. S. Calude, E. Calude. The complexity of the Four Colour Theorem, *LMS J. Comput. Math.* 13 (2010), 414–425.

[7] C. S. Calude, E. Calude and K. Svozil. The complexity of proving chaoticity and the Church-Turing Thesis, *Chaos* 20 037103 (2010), 1–5.

[8] C. S. Calude, M. J. Dinneen. Exact approximations of omega numbers, *Int. Journal of Bifurcation & Chaos* 17, 6 (2007), 1937–1954.

[9] C. S. Calude, M. J. Dinneen and C.-K. Shu. Computing a glimpse of randomness, *Experimental Mathematics* 11, 2 (2002), 369–378.

[10] E. Calude. The complexity of Riemann's Hypothesis, *Journal for Multiple-Valued Logic and Soft Computing*, (2012). (to appear)

[11] E. Calude. Fermat's Last Theorem and chaoticity, *Natural Computing*, (2011), DOI: 10.1007/s11047-011-9282-9.

[12] M. J. Dinneen. A program-size complexity measure for mathematical problems and conjectures, in M. J. Dinneen, B. Khoussainov, A. Nies (eds.). *Computation, Physics and Beyond*, Springer, Heidelberg, 2012. (to appear)

[13] J. Kelleher. *Encoding Partitions as Ascending Compositions*, PhD Thesis, University College Cork, 2006.

[14] D. Knuth. *The Art of Computer Programming, Pre-Fascicle 3b: Generating all partitions*, `http://www-cs-faculty.stanford.edu/%7Eknuth/fasc3b.ps.gz`,(version of 10 December 2004).

[15] D. Stanton, D. White. *Constructive Combinatorics*, Springer-Verlag, New York, 1986.

[16] A. Zoghbi, I. Stojmenovic. Fast algorithms for generating integer partitions, *International Journal of Computer Mathematics* 70 (1998), 319–332.

[17] D. Wells. Are these the most beautiful? *The Mathematical Intelligencer* 12, 3 (1990), 37–41.

Department of Computer Science, The University of Auckland, New Zealand, www.cs.auckland.ac.nz/~cristian.

Institute of Information and Mathematical Sciences, Massey University at Auckland, New Zealand, http://www.massey.ac.nz/~ecalude.