ZET User Manual

Manuel Kauers*

March 22, 2004

Abstract

We describe the Mathematica package ZET that provides functions for deciding zero equivalence of a very large class of sequences. After specifying the class of admissible sequences, we give a thorough introduction to the usage of the package.

Contents

* Partially supported by the Austrian Science Foundation (FWF) grant F1305 and the German Academic Exchange Service (DAAD) grant D/03/40515

1 Introduction

This document describes the Mathematica package ZET which provides an implementation of a new algorithm for proving zero equivalence of certain sequences. This algorithm and its theoretical background is described elsewhere [4, 5].

In this document, we do not repeat theoretical aspects that are dealt with in the original papers. Rather, we want to provide a user manual for our Mathematica package. Theoretical aspects (such as the definition of admissible sequences in Section 4) are only included as far as they are of importance for the user.

The package is meant as a prototype implementation of our algorithm, and although we carefully tested it, we cannot guarantee its correctness in all circumstances. Any user encountering an unexpected behavior is encouraged to send a bug report to manuel@kauers.de .

How to read this manual: After some notes on installation and system requirements (Section 2), we present a first tour showing how to handle a number of common example situations with the package (Section 3). After that, we define in Section 4 the class of sequences that can be dealt with by the package. The next section, Section 5, provides a comprehensive survey of the functionality of the package. Section 6 contains some implementation notes which will not be of interest for most users. Some additional material is enclosed as an appendix.

2 Installation and System Requirements

There is no complication concerning installation of the package. Just put the package file into a directory where Mathematica is able to find it. A choice that should

always work is the directory from which Mathematica is started, i.e., the directory ".".

After Mathematica has been started, the package has to be loaded. You may use the command

In[1]:= << Zet.m

for loading the package. (This should be the first thing you do in a session in which you want to use the package.) If Mathematica succeeds in finding the package, it will print a copyright note of the following form:

Zet Package by Manuel Kauers — © RISC Linz — V 0.1 (04-02-16)

Otherwise, it will print a message.

Get::noopen : Cannot open Zet.m

Out[1]= $Failed

The package has been developed and tested for Mathematica 5. Probably it will also run under later versions of Mathematica 4 (e.g., under 4.2), but we do not expect it to run under older versions of Mathematica.

3 A Quick start

After the package has been installed and loaded into Mathematica (see the previous Section), functions for defining sequences and proving their equality are available. The function for proving the equality of two sequences is IdenticalQ. It takes the equation to be proven as an argument. In addition, it is necessary to specify the induction variable by the option ForAll. IdenticalQ returns True, False, or the symbol Unknown. Examples are in order.

In[2]:= $\text{IdenticalQ}[\sum_{i=1}^{n} i == \frac{1}{2}n(n+1), \text{ForAll} \to n]$

Out[2]= True

In[3]:= $\text{IdenticalQ}[\sum_{i=1}^{n} i == \frac{1}{2}n(n-1), \text{ForAll} \to n]$

Out[3]= False

Note that ZET redefines Mathematica's Sum function such that no attempts are made in order to compute symbolic sums. Yet it is still possible to evaluate sums if the bounds are integers. A similar remark holds for products.

In[4]:= $\text{Sum}[i, \{i, 1, n\}] // \text{InputForm}$

Out[4]= $\text{Sum}[i, \{i, 1, n\}]$

In[5]:= $\text{Sum}[i, \{i, 1, 20\}]$

Out[5]= 210

A lot of sequences can be entered in terms of usual expressions, but it is also possible to define additional sequences using the command DefineSequence. Definitions made with this command are stored in a global database, and can be used both in

IdenticalQ and in the definition of further sequences. For example, the sequence $f(n) = \sum_{i=1}^{n} g(i)$ where $g(n)$ is defined by the recurrence

$$g(n+3) = 2n^2 g(n+2) + 2^n g(n) + \sum_{i=0}^{n} \frac{1}{i}, \quad g(1) = g(2) = \pi$$

may be defined by the commands

In[6]:= DefineSequence[$g[n+3] == 2n^2 g[n+2] + 2^n g[n] + \sum_{i=0}^{n} 1/i, g[1] == \mathrm{Pi}, g[2] == \mathrm{Pi}$]

In[7]:= DefineSequence[$f[n] == \sum_{i=1}^{n} g[i]$]

We may use undefined sequences in the definition of another sequences. It is then assumed that the undefined sequence bears no algebraic relation (recurrence) with other sequences. This feature is useful for proving identities in an undetermined number of variables, see [5] for theoretical details.

Sequences from the database can not only be used for proving identities, but they can also be evaluated at integer points. The function GetValue takes an expression in terms of defined sequences as input and returns the evaluation of that expression where the running variable takes a value that is specified in the form of an option. Example:

In[8]:= GetValue[$f[n], n \to 10$]

Out[8]= $$\frac{208093905664}{105} + 4126044512\pi + 3394452635 g[0]$$

Note that it did not matter that we specified only two initial values in the definition of g, although three are necessary for a unique definition of a sequence by a recurrence of order three. ZET inserts symbolic values at undefined points.

4 Mathematical Background

Before specifying the previously introduced commands in a more rigorous manner, we first provide some mathematical underpinnings about the objects that we are able to deal with.

4.1 Admissible Sequences

Admissible sequences are defined by certain difference ideals [2] over the field of rational numbers (or some transcendental or algebraic extension thereof).

The set of admissible sequences may be defined by structural induction as follows. If $(f_1(n))_{n=1}^{\infty}, \ldots, (f_s(n))_{n=1}^{\infty}$ are admissible sequences ($s \geq 0$ fixed; $s = 0$ establishes the induction base), if p is a polynomial in the indeterminates $x_1, \ldots, x_{(s+1)(r+1)-1}$ ($r \geq 0$ fixed), and if the sequence $(f(n))_{n=1}^{\infty}$ satisfies the recurrence

$$\begin{aligned} f(n+r) = p\big(&f_1(n), f_1(n+1), \ldots\ldots, f_1(n+r-1), f_1(n+r), \\ &f_2(n), f_2(n+1), \ldots\ldots, f_2(n+r-1), f_2(n+r), \\ &\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ &f_s(n), f_s(n+1), \ldots\ldots, f_s(n+r-1), f_s(n+r), \\ &f(n),\ f(n+1),\ \ldots\ldots, f(n+r-1)\big) \end{aligned} \tag{$*$}$$

or the recurrence $f(n+r) = 1/p$ where p takes the same arguments as above, then $(f(n))_{n=1}^{\infty}$ is also admissible.

The class of admissible sequences is obviously closed under arithmetic operations, indefinite sums and products. It is further closed under taking continuants (useful for defining continued fractions), contains doubly exponential sequences like 2^{3^n}, and so on.

In the original paper [4], admissible sequences are called *nested polynomially recurrent* reflecting the fact that the defining recurrence of admissible sequence need not be linear, and that the defining recurrences may depend on sequences that have been defined before (structural induction).

4.2 Sequences of Variables

It is also possible to handle an extended class of admissible sequences. The class of extended admissible sequences is defined by structural induction as follows. If $(u_n)_{n=1}^{\infty}$ is a sequence of indeterminates, each u_n algebraically independent of all others, then the sequence of the u_n is extended admissible. Furthermore, as before, if $(f_1(n))_{n=1}^{\infty}$, ..., $(f_s(n))_{n=1}^{\infty}$ are extended admissible sequences ($s \geq 0$ fixed), p is a polynomial in the indeterminates $x_1, \ldots, x_{(s+1)(r+1)-1}$ ($r \geq 0$ fixed), and the sequence $(f(n))_{n=1}^{\infty}$ satisfies the recurrence $(*)$ or the recurrence $f(n+r) = 1/p$ where p takes the same arguments as in $(*)$, then $(f(n))_{n=1}^{\infty}$ is also admissible.

Example: The sequence $(f(n))_{n=1}^{\infty}$ with $f(n) = \sum_{i=1}^{n} x_n$ is extended admissible.

4.3 Analytical vs. Algebraic Correctness

The use of extended admissible sequences introduces a complication: The algorithm of [4] does not terminate for these sequences in general. It does terminate in many cases in practice, and in these cases its result is fully correct. However, if it doesn't terminate, nothing can be said about whether the conjectured identity does or does not hold.

There is a way to force termination [5]. Sloppily speaking, the key idea is not to compute in a ring $\mathbb{Q}[f_1, f_2, \ldots, f_m]$ of sequences, but to compute in the ring

$$\mathbb{Q}(f_1, \ldots, f_s)[f_{s+1}, \ldots, f_m]$$

where the sequences of variables are considered as part of the ground field. It can be shown that this modification recovers the termination of the proving algorithm.

The disadvantage of using the field $\mathbb{Q}(f_1, \ldots, f_s)$ is that it allows to divide by polynomials in $f_1, \ldots, f_s$ during the algorithm, and hence the proof of an identity depending on a free sequence $x_1, x_2, \ldots$ may not be valid for all instances of $x_1, x_2, \ldots$. This is a common phenomenon in symbolic algorithms. We say that the algorithm is "algebraically correct" but not necessarily "analytically." It can be shown, however, that if the algorithm proves an identity "algebraically correct," then this identity holds for almost all instances. Here "almost all" means that the exceptions form a set of measure zero. On the other hand, if the algorithm returns False, then the identity is, of course, definitely false.

Both versions of the algorithm are implemented in the package. See the documentation of the proving commands for further information.

5 User Manual

We proceed by a complete description of the functionality provided by the package.

5.1 Definition of Sequences

The package maintains a global database of sequences. The user may add additional sequences to this database by the command DefineSequence. This command takes a number of equations defining the sequence as arguments. A sequence is defined by a recurrence or by initial values or, most commonly, by both.

The recurrence is to be given in the form

$$\text{name[variable} + \text{order]} == \text{recurrence}$$

where

- name is a symbol. The sequence will be entered to the database under this name.
- variable is a symbol. This symbol has only a meaning while defining the sequence and it will be forgotten afterwards. (Such variables are called "dummy variables" in the Mathematica book.)
- order is a nonnegative integer. For all occurrences of name[variable + i] on the right hand side, i must be less than order.
- recurrence is the right hand side of the recurrence defining the sequence. It has to be of the form described in Section 4, or at least it must be able to transform it into such a form.

Initial values are given in the form

$$\text{name[point]} == \text{value}$$

where

- name is as above.
- point is an integer for which a special value of the sequence is to be defined.
- value is the value which the sequence takes at the specified point.

The definition of a sequence may consist of zero or one recurrence and zero or more specifications of initial values. If a recurrence is given, it has to be given as the first argument.

5.1.1 Form of the Recurrence

The recurrence may be any expression that can successfully be turned into a defining difference polynomial. This is, according to Section 4, definitely possible, if the right hand side of the recurrence is a polynomial in (possibly shifted) sequences which are already in the database and of lower shifts of the sequence to be defined.

Subexpression $f[variable + i]$ where i is an integer but no sequence named f has been defined before, will cause the definition of a free sequence with name f. If a free sequence is introduced to the database, the user will be informed about this step by a message.

Example:

In[1]:= DefineSequence[$f[n+1] == f[n] + x[n]$]

Zet:defarbitrary : Definition involves undefined function symbol x which will be regarded as free.

The right hand side of the recurrence can refer to other (!) sequences also in shifts higher than order. The DefineSequence command knows how to transform such defining relations into the required form.

Example:

In[2]:= DefineSequence[$g[n+2] == f[n+3] * g[n] + x[n-1]$]

Here $f[n+3]$ is internally replaced by $f[n+2]+x[n+2]$ in order to ensure that the right hand side of the recurrence does not contain higher shifts as 2. Afterwards, the negative shift in $x[n-1]$ is removed by shifting the whole recurrence by one to the right. Finally, the relation $g[n+3] == (f[n+3]+x[n+3]) * g[n+1]+x[n]$ will be written into the database as definition of g.

In addition to previously defined sequences, the right hand side of a recurrence may involve more sophisticated expression for which DefineSequence knows how to translate them into defining relations. Section 5.2 lists these constructions. An example of such a definition is

In[3]:= DefineSequence[$h[n+1] == h[n] * \sum_{i=1}^{n} \frac{g[i]^2+5}{f[i]+i^2} - 7^{\underline{n}} * 2^{3^n}$]

5.1.2 Initial Values

The DefineSequence command accepts an unlimited number of initial conditions as arguments after the recurrence. The initial conditions are given in the form

$$\text{name}[i] == \text{value}$$

where name is the name of the sequence being defined, i is an integer, and value is the desired value of the sequence at the point i. The value can be an arbitrary Mathematica expression. It will not be processed any further by ZET.

Example: The following line defines the "tribonacci sequence."

In[1]:= DefineSequence[$f[n+3] == f[n]+f[n+1]+f[n+2], f[1] == 1, f[2] == 1, f[3] == 1$]
In[2]:= GetValue[$f[n], n \to 300$]

Out[2]= 3442037825395850124953697216417582090412534767003462109126853127 61

The sequence $f[n]$ is not defined for $n < 1$:

In[3]:= GetValue[$f[n], n \to -7$]

Out[3]= $f[-7]$

Without specifying any initial values, an evaluation of the sequence is not possible:

In[4]:= DefineSequence[$g[n+3] == g[n] + g[n+1] + g[n+2]$]
In[5]:= GetValue[$g[n], n \to 300$]

Out[5]= $g[300]$

We may, however, give only partial information about the initial values:

In[6]:= DefineSequence[$h[n+3] == h[n] + h[n+1] + h[n+2], h[1] == 1, h[3] == 1$]
In[7]:= GetValue[$h[n], n \to 300$]

Out[7]= $4475476725048788547177790 6 + 24332675219681431451788241 h[2]$

The "missing" initial values will be padded by symbolic values.

Note that the initial values, if specified at all, must come after the recurrence, otherwise the definition call remains unevaluated.

In[8]:= DefineSequence[$h[1] == 1, h[n+3] == h[n] + h[n+1] + h[n+2], h[3] == 1$]

Out[8]= DefineSequence[$h[1] == 1, h[n+3] == h[n] + h[n+1] + h[n+2], h[3] == 1$]

5.1.3 Definition of free sequences

A free sequence, or a sequence of indeterminates, is a sequence for which no recurrence is specified. An identity involving a free sequence $(x_n)_{n=1}^{\infty}$ is interpreted as being valid for any sequence in place of $(x_n)_{n=1}^{\infty}$. See Sections 4.2 and 4.3 for details.

Free sequences may still have specific values at certain specific points. This makes it possible to prove identities that hold, say, for all sequences $(x_n)_{n=1}^{\infty}$ with $x_{17} = 0$. Such sequences are defined using DefineSequence without specifying a defining recurrence.

Example:

In[1]:= DefineSequence[$a[5] == 19, a[19] == 5$]

In[2]:= GetValue[$a[n], n \to 5$]

Out[2]= 19

In[3]:= GetValue[$a[n], n \to 15$]

Out[3]= $a[15]$

In[4]:= GetValue[$a[n], n \to 19$]

Out[4]= 5

If a sequence has neither specific values nor a defining recurrence, it cannot be defined using DefineSequence. Such "completely free" sequences are created automatically when the definition of another sequence involves a name for which no sequence information is available in the database. A warning informs the user that a symbol is understood as a free sequence. Once defined, the free sequence remains in the database and its use in subsequent definitions will not cause a warning message any more.

See the definition of f in Section 5.1.1 for an example.

5.1.4 Overwriting and Extending Definitions

Sequences stay in the database until the end of the Mathematica session or until a ClearDefinitions[] is raised. Despite of this, it is possible to define a new sequence using a name that has already been used before. If the definition is successful, the binding of the name to the old sequence will be released, and the user will be informed about this by a warning.

It is important to note that overriding a definition only affects the usage of the name in future definitions, but past definitions remain as they are. This is because overridden sequences do remain in the database, just their name has been removed.

Example: Let f be the sequence of the tribonacci numbers and $F(n) = \sum_{i=1}^{n} f(n)$.

In[1]:= DefineSequence[$f[n+3] == f[n]+f[n+1]+f[n+2], f[1] == 1, f[2] == 1, f[3] == 1$]

In[2]:= DefineSequence$[F[n] == \sum_{i=1}^{n} f[i]]$

In[3]:= GetValue$[F[n], n \to 250]$

Out[3]= 754318419046334600679914454791421846795201163712517505845557838462

In[4]:= GetValue$[f[n], n \to 250]$

Out[4]= 344203782539585012495369721641758209041253476700346210912685312761

If we now redefine f as the sequence of Fibonacci numbers, this will not affect the definition of F, although the definition of F contained the symbol f.

In[5]:= DefineSequence$[f[n+2] == f[n+1] + f[n], f[0] == 0, f[1] == 1]$

Zet::overdef : Old definition of f has been replaced by the new one.

In[6]:= GetValue$[F[n], n \to 250]$

Out[6]= 754318419046334600679914454791421846795201163712517505845557838462

In[7]:= % == Out[3]

Out[7]= True

In[8]:= GetValue$[f[n], n \to 250]$

Out[8]= 7896325826131730509282738943634332893686268675876375

In[9]:= % == Out[4]

Out[9]= False

In[10]:= GetValue$[\sum_{i=1}^{n} f[i], n \to 250]$

Out[10]= 20672849399056463095319772838289364792345825123228623

In[11]:= % == Out[3]

Out[11]= False

If a definition for a sequence with name f is stated and a sequence with this name is already in the database, the old definition will usually be overridden. It might be, however, that the new definition is consistent with the old definition. This is the case when the two sequences evaluate to the same values on the intersection of their domains of definition. In this case, for efficiency reasons, both sequences will share one entry of the database, and the larger domain of definition will be used for both of the sequences. This is called the *extension* of a definition. For the matter of extending a definition, it is irrelevant whether the extended sequence has the same name as the extending sequence or not.

Example: The Fibonacci sequence f from above is as yet undefined for negative integers:

In[12]:= GetValue$[f[n], n \to -2]$

Out[12]= $f[-2]$

We can define the sequence g of Fibonacci numbers starting at $n = -1$. If we specify $g[-2] = -1$ and $g[-1] = 1$, then $f[n]$ and $g[n]$ will have the same values for $n \geq 0$. Hence the definition of f will be extended and g will become a synonym for f.

In[13]:= DefineSequence[$g[n+2] == g[n+1] + g[n], g[-2] == -1, g[-1] == 1$]

In[14]:= GetValue[$g[n], n \to -2$]

Out[14]= -1

In[15]:= GetValue[$f[n], n \to -2$]

Out[15]= -1

Extension of definitions works also in the other direction: If we let $h[n]$ be the sequence of Fibonacci numbers starting at $n = 10$, then the definition will be automatically extended to the range $n \geq -2$ due to the already existing definitions of f and g.

In[16]:= DefineSequence[$h[n+2] == h[n+1] + h[n], h[10] == 55, h[11] == 89$]

In[17]:= GetValue[$h[n], n \to -2$]

Out[17]= -1

Note that it is not sufficient that the recurrences agree for extending a definition: also the evaluation has to be the same:

In[18]:= DefineSequence[$F[n+2] == F[n+1] + F[n], F[1] == 7, F[2] == 8$]

In[19]:= GetValue[$F[n], n \to -2$]

Out[19]= $F[-2]$

In[20]:= GetValue[$F[n], n \to 10$] == GetValue[$f[n], n \to 10$]

Out[20]= False

5.1.5 Failing Definitions

The definition of a sequence may fail. In case of a failure, a warning is raised and DefineSequence returns the value $Failed. (In case of success, it returns the value Null.)

An attempt to define a sequence will fail if the recurrence is not of the required form. It is of the required form if it can be transformed into the shape specified in Section 4.1, i.e., into a difference polynomial possibly involving other (extended) admissible sequences.

Example: Sin[$1/n$] is not an appropriate right hand side.

In[1]:= DefineSequence[$f[n] ==$ Sin[$1/n$]]

Zet::unable : Unable to convert definition to algebraic recurrence.

Out[1]= $Failed

If a definition for f failed, possibly existing other definitions for f remain untouched. If f was not defined before, it will also be undefined afterwards.

Note, by the way, that the following definition goes through:

In[2]:= DefineSequence[$f[n] == \mathrm{Sin}[n]$]

Zet::defarbitrary : Definition involves undefined function symbol Sin which will be regarded as free.

The reason is that ZET does not know about the special meaning of the symbol Sin in Mathematica. See Section 5.2 for a list of expressions which are known to ZET.

Using sophisticated expressions as described in Section 5.2, it is important that ZET has to have a chance to transform the given expression into an algebraic recurrence. For instance, when using the summation symbol, the sum must not be definite, for ZET doesn't know how to deal with definite sums.

In[3]:= DefineSequence[$f[n] == \mathrm{Sum}[1/(i+n), \{i, 1, n\}]$]

Zet::definite : Definite Sum encountered. No algebraic relation can be found.

Out[3]= $Failed

It is also not allowed that the sequence which is currently being defined appears in the summand:

In[4]:= DefineSequence[$f[n] == \mathrm{Sum}[i * f[i]^2, \{i, 1, n-1\}]$]

Zet::contains : Recurrence not admissible: f must not appear within Sum

Out[4]= $Failed

5.1.6 Removing Definitions

It is not possible to remove single definitions from the database in an other way than overwriting definitions by new definitions. (This does actually not remove a definition but only changes the binding of a name to a different definition.)

It is possible to remove all definitions in one go by means of the ClearDefinitions[] command. This command takes no arguments and resets the database to the initial state which it had immediately after loading the package.

5.2 Input by High-Level Expressions

Though in theory the right hand side of a recurrence must be of the form described in Section 4.1, it is also possible to specify sequences using certain high-level constructions, if ZET knows how to translate them into the required from.

Example: If the sequence $f(n)$ is already defined, the definition

In[1]:= DefineSequence[$F[n] == \sum_{i=1}^{n} f[i]$]

will internally cause the definition of a sequence with a temporary name, say t, by $t(n+1) = t(n) + f(n+1)$, $t(1) = f(1)$. After that, the recurrence in the original definition is replaced by $F[n] == t[n]$, which is of the appropriate form and has the desired meaning.

We shall investigate in this section for which high-level constructions this is possible (resp. implemented). The same set of high-level constructions applies not only to the recurrence argument of DefineSequence but also to most other functions of ZET, e.g., the proving functions to be discussed later.

5.2.1 Built-in Atomic Expression

ZET knows how to translate the following expressions into recurrences. We suppose that the running variable of the sequence under consideration is n.

- the term n itself as well as any expression α independent of n
- α^{an+b} where α is independent of n and $a, b \in \mathbb{Z}$; also Exp$[an+b]$
- $(an+b)!$ (factorial) if $a \in \mathbb{N}, b \in \mathbb{Z}$
- RaisingFactorial$[x, an+b]$ if $a \in \mathbb{N}, b \in \mathbb{Z}$ and x independent of n. The raising factorial $x^{\overline{an+b}}$ is defined as

$$x^{\overline{an+b}} = \prod_{i=1}^{an+b} (x+i-1) \qquad (n \in \mathbb{N})$$

- FallingFactorial$[x, an+b]$ if $a \in \mathbb{N}, b \in \mathbb{Z}$ and x independent of n. The falling factorial $x^{\underline{an+b}}$ is defined as

$$x^{\underline{an+b}} = \prod_{i=1}^{an+b} (x-i+1) \qquad (n \in \mathbb{N})$$

- Binomial$[an+b, cn+d]$ (the binomial coefficient) for $a, c \in \mathbb{N}$, $a > c$, $b, d \in \mathbb{Z}$
- Harmonic$[an+b]$, (harmonic number $H(n) := \sum_{i=1}^{n} 1/i$) if $a \in \mathbb{N}, b \in \mathbb{Z}$; also Harmonic$[r, an+b]$ (harmonic number of the rth kind, $H^{(r)}(n) := \sum_{i=1}^{n} 1/i^r$ if $r \in \mathbb{Z}$) where $r \in \mathbb{Z}$.
- Fib$[n]$ the sequence of Fibonacci numbers; also $a^{\mathrm{Fib}[n]}$ if a is a constant
- $a^{bc^{dn+e}+f}$ if a, b, c, d, e, f are independent of n and c^d is an integer
- Gamma$[an+b]$ (the Gamma function) if $a \in \mathbb{Q}$ and b is independent of n
- Gamma$[an+b,\ x]$ (the incomplete Gamma function) if $a \in \mathbb{Q}$ and b, x are independent of n

5.2.2 Built-in Closure Properties

As opposed to the previously listed expressions, it is also possible to construct new sequences from sequences already defined and/or definable by reference to the list above.

In the list below, we assume that f and g are sequences which are known to the database, and that n is the running variable of the sequence. Then the following expressions can be turned into recurrences by ZET.

- $f[n] + g[n]$, $f[n] * g[n]$, $f[n]/g[n]$, $f[n]^a$ if $a \in \mathbb{Z}$,
- Sum$[f[i], \{i, a, bn+c\}]$ where $b \in \mathbb{N}$, $a, c \in \mathbb{Z}$, and i is a symbol.
- Product$[f[i], \{i, a, bn+c\}]$ where $b \in \mathbb{N}$, $a, c \in \mathbb{Z}$, and i is a symbol.
- Cfrac$[f[i], \{i, a, n+c\}]$ where $a, c \in \mathbb{Z}$ and i is a symbol. This defines the continued fraction

$$\mathrm{Cfrac}[f[i], \{i, a, n+c\}] = \operatorname*{K}_{i=a}^{n+c} f[i] = f[a] + \cfrac{1}{f[a+1] + \cfrac{1}{\ddots \; f[n+c-1] + \cfrac{1}{f[n+c]}}}$$

- $f[ab^{cn+d}+e]$ if a,b,c,d,e are independent of n, b^c is an integer, and f is defined by a C-finite recurrence. This is the case if the defining relation for f has the form

$$f(n+r)=\alpha_{r-1}f(n+r-1)+\cdots+\alpha_1 f(n+1)+\alpha_0 f(n)$$

for some fixed $r\in\mathbb{N}$ and constants $\alpha_0,\ldots,\alpha_{r-1}$ independent of n.

Note that the above closure properties may well be combined with each other, mutually nested, or whatever. A complicated and meaningless example is

In[1]:= DefineSequence[$f[n]$ == $\frac{\text{Sum[Sum[Binomial}[2i,i]*2^{2^i},\{i,1,k\}]*k!,\{k,1,n\}]}{(2^n)^{15}+\text{Sum[1/Harmonic}[i],\{i,1,n\}]}$]

In[2]:= GetValue[$f[n],n\to 8$]

Out[2]=
$$\frac{3585754412416766563017697851125789650986743388507126 00}{24712280543980429935406358465771502020 03}$$

The functions Sum, Product and Cfrac also support nested iterators in analogy to the Table function of Mathematica. This allows to type, for example,

$$\text{Sum}[f[l],\{i,1,n\},\{j,1,i\},\{k,1,j\},\{l,1,k\}],$$

instead of the more tedious expression

$$\text{Sum}[\text{Sum}[\text{Sum}[\text{Sum}[f[l],\{l,1,k\}],\{k,1,j\}],\{j,1,i\}],\{i,1,n\}].$$

Likewise for Product and Cfrac. Recall that according to Mathematica's convention, the outermost iterator has to be stated first.

5.2.3 Pretty printing

It is worth noting that the ZET package redefines function symbols such as Sum and Product in order to prevent them from being evaluated. This makes it possible to prove identities like $\sum_{i=1}^{n} i=\frac{1}{2}n(n+1)$ without that Mathematica itself evaluates the left hand side to the right hand side and nothing remains to be proven for ZET.

In order to remind the user that, say, Sum is no longer a function but only a symbol, their appearance has also been changed. Summation and product signs have been replaced by capital sigma and capital pi, respectively.

In[1]:= Sum[$f[i],\{i,1,n\}$]

Out[1]=
$$\sum_{i=1}^{n} f[i]$$

In[2]:= << Zet.m

```
Zet  Package by Manuel Kauers — © RISC Linz — V 0.1  (04-02-16)
```

In[3]:= Sum[$f[i],\{i,1,n\}$]

Out[3]=
$$\overset{n}{\underset{i=1}{\Sigma}} f[i]$$

Further pretty printing is defined for

- Product[$f[i],\{i,1,n\}$] — appears as $\Pi_{i=1}^{n} f[i]$
- Cfrac[$f[i],\{i,1,n\}$] — appears as $\mathrm{K}_{i=1}^{n} f[i]$

- Fib$[n]$ — appears as $\mathbf{F}[n]$
- Harmonic$[n]$ — appears as $\mathbf{H}[n]$
- Harmonic$[r, n]$ — appears as $\mathbf{H}^{(r)}[n]$
- RaisingFactorial$[a, n]$ — appears as $a^{\overline{n}}$
- FallingFactorial$[a, n]$ — appears as $a^{\underline{n}}$

5.3 Evaluation of Sequences

The function GetValue admits the evaluation of a sequence at a given integer point. This function expects as a first argument an expression which is an admissible right hand side for a recurrence in DefineSequence, i.e., some expression which is built up of sequences in the database and known atomic expressions (Section 5.2.1) by means of known closure properties (Section 5.2.2). The running variable and the integer which should be substituted for it are specified by an option "variable $\rightarrow$ point."

Example:

In[1]:= $\text{DefineSequence}[f[n] == \sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j}\sum_{l=1}^{k} 1/l]$

In[2]:= $\text{GetValue}[f[n], n \rightarrow 150]$ // Timing

Out[2]= $\{0.04\text{Second}, \frac{2831310667393933652581438949350085399328696611613766191618657727927}{1280597361239530116161982520711799208053839773728194575324000}\}$

Observe that the evaluator at the core of GetValue is very efficient, plain Mathematica needs much more time for doing the same job:

In[3]:= $\sum_{i=1}^{150}\sum_{j=1}^{i}\sum_{k=1}^{j}\sum_{l=1}^{k} 1/l$ // Timing

Out[3]= $\{86.07\text{Second}, \frac{2831310667393933652581438949350085399328696611613766191618657727927}{1280597361239530116161982520711799208053839773728194575324000}\}$

5.4 Deciding Zero Equivalence

We now turn to the functions that implement the proving algorithm.

5.4.1 The Zero Equivalence Test and Proving Identities

Let e_1, e_2 be expressions with a free variable n. We want to decide if $e_1 = e_2$ for all $n \geq 0$. If e_1, e_2 are such that we can say

$$\text{DefineSequence}[e1[n] == e_1]; \text{DefineSequence}[e2[n] == e_2];$$

where $e1, e2$ are new symbols, then we can decide whether e_1, e_2 are identical by means of the function IdenticalQ. This function takes as an argument an equation $e_1 == e_2$ where e_1, e_2 are as above, and it expects the option ForAll to point to the variable in which these sequences are given.

Examples:

In[1]:= $\text{IdenticalQ}[\text{Sum}[i, \{i, 1, n\}] == n * (n + 1)/2, \text{ForAll} \rightarrow n]$

Out[1]= True

In[2]:= $\mathrm{IdenticalQ}[\mathrm{Sum}[i,\{i,1,k\}] == k*(k+1)/2, \mathrm{ForAll} \to k]$

Out[2]= True

In[3]:= $\mathrm{IdenticalQ}[\mathrm{Sum}[i,\{i,1,n\}] == n*(n-1)/2, \mathrm{ForAll} \to n]$

Out[3]= False

The IdenticalQ function internally calls ZeroQ on the difference of the two sides of the equation. ZeroQ takes one expression as argument and the same options as IdenticalQ. It returns True or False, depending on whether the given expression vanishes identically.

It is worth noting that the procedure is guaranteed to terminate no matter if the result is True or False.

Example: Consider the sequence $(f(n))_{n=1}^{\infty}$ defined by

$$f(n) = (n-3)(n-7)\sum_{k=1}^{n}(-1)^k(1+\prod_{k=1}^{n}(-1)^k).$$

We have $f(n) = 0$ for $n = 1, 2, 3, 4, 5, 6, \ldots$ and we may assume $f(n) = 0$ for all n. Is this true?

In[4]:= $\mathrm{ZeroQ}[(n-3)(n-7)\sum_{k=1}^{n}(-1)^k(1+\prod_{k=1}^{n}(-1)^k), \mathrm{ForAll} \to n, \mathrm{MaxOrder} \to \mathrm{Infinity}]$

Out[4]= False

Indeed, it turns out that $f(11) = -64 \neq 0$. (See pages 16 and 24 for an explanation of the MaxOrder option.)

A synonym for both ZeroQ and IdenticalQ is Prove. This function takes the same options as ZeroQ and IdenticalQ do. Its first argument may or may not be an equation, if it's not, then zero equivalence of the argument is tested.

No termination is guaranteed if the query involves free sequences. A warning is raised in this event.

Example:

In[5]:= $\mathrm{DefineSequence}[p[k+2] == a[k+2]*p[k+1]+p[k], p[-2] == 0, p[-1] == 1]$

Zet::defarbitrary : Definition involves undefined function symbol a which will be regarded as free.

In[6]:= $\mathrm{DefineSequence}[q[k+2] == a[k+2]*q[k+1]+q[k], q[-2] == 1, p[-1] == 0]$

In[7]:= $\mathrm{IdenticalQ}[p[i+1]*q[i]-p[i]*q[i+1] == (-1)^i, \mathrm{ForAll} \to i]$

Zet::arbitrary : Conjecture involves arbitrary sequences. Algorithm may not terminate.

Out[7]= True

In[8]:= $\mathrm{IdenticalQ}[p[i+1]*q[i-1]-p[i-1]*q[i+1] == -(-1)^i*a[i+1], \mathrm{ForAll} \to i]$

Zet::arbitrary : Conjecture involves arbitrary sequences. Algorithm may not terminate.

Zet::maxorder : Maximum order 6 reached without having found an answer

Out[8]= Unknown

The identity queried in line 8 is as correct as the one in line 7, but the algorithm fails to terminate here owing to the appearance of the free sequence a. It could have failed to terminate in line 7 as well, because it suffices that the a appears in the definition of p and q (hence the warning also here), but by coincidence it succeeds here.

In line 8, the proving engine gave up after reaching the threshold order 6, and it returned the value Unknown. We will see next how this threshold and other parameters can be modified via options, and how it can be requested that free sequences are regarded as field elements, ensuring termination of the algorithm (Section 4.3).

5.4.2 Options

The proving commands IdenticalQ, ZeroQ, and Prove accept the following options:

- ForAll → the running variable for all values of which the identity is to be proven. This option is obligatory.
- From → smallest point for which the identity is claimed. The default value for this option is 1. It is assumed that the option points to an integer.
- MaxOrder → the number of iterations after which to give up. If the algorithm does not find an answer (proof or counterexample) after that many iterations, it prints a warning and returns the value Unknown. This option defaults to 5.
- OrderMessage → True to indicate that a progress report should be printed at the beginning of each iteration of the prover's main loop. This option defaults to False.
- Field → boolean value specifying whether free sequences should be put into the ground field or left in the polynomial ring. This option defaults to False. If a query involves free sequences and this option is set to False, then the proving algorithm may fail to terminate, and an according message will be raised.
- CheckInitialValues → boolean value indicating if the base of the induction should be checked. If set to True, the prover returns True if the induction step has been found, and Unknown otherwise. The default is False.
- StartOrder → nonnegative integer i requesting the prover to omit the first i iterations of the proving algorithm and start directly with the ith iteration. This option defaults to 0.
- GroebnerBasis → name of the function to be used for performing Gröbner basis computations. By default, Mathematica's built-in GroebnerBasis command is used, but it might be useful to change to a faster implementation. Any function that fulfills the specification of Mathematica's GroebnerBasis function may replace it.

5.5 Additional Functions

Some additional tools are provided by the ZET package. These are described next.

5.5.1 Automated Proof Generation (Experimental)

There is an experimental function which allows not only to prove or disprove an identity, like ZeroQ and IdenticalQ do, but which justifies the result also by providing the user with a rigorous mathematical proof.

The function Proof takes the same arguments as IdenticalQ and ZeroQ do, in addition it requires the specification of a function for computing cofactors, its name to be declared by the option Cofactors. This function should take three arguments p, $\{p_1, p_2, \dots\}$, $\{x_1, x_2, \dots\}$, where $p, p_1, p_2, \dots$ are polynomials in $x_1, x_2, \dots$ and and p belongs to the ideal generated by $p_1, p_2, \dots$. The function should return a list of polynomials $\{c_1, c_2, \dots\}$ such that $\sum_i c_i p_i = p$. The c_i are called the cofactors of p wrt. $p_1, p_2, \dots$. It is assumed that the function for computing the cofactors accepts the same options as the GroebnerBasis command.

Unfortunately, Mathematica does not provide an appropriate function for computing cofactors by itself. But the author's Gb package for Mathematica, providing an interface to Jean-Charles Faugere's Gb library for fast Gröbner basis computation, does contain such a function.

Suppose now that a function for cofactor computation is available. Then it is possible to construct proof objects by means of the Proof function. For example, we can create a proof object for our favorite example identity.

In[1]:= $\text{myIdentity} = \sum_{i=1}^{n} i == \frac{1}{2}n(n+1)$

Out[1]= $\sum_{i=1}^{n} == \frac{1}{2}n(n+1)$

In[2]:= $p = \text{Proof}[\text{myIdentity}, \text{ForAll} \to n, \text{Cofactors} \to (\dots)];$

It is a good idea not to look at a proof object directly, as it is an ugly expression containing the information of the proof in a Mathematica-readable format.

In[3]:= $\text{Head}[p]$

Out[3]= Proof

For transforming a proof object into a human readable form, there exists as yet a function TeXify which takes a proof object and a string as arguments, and which writes LaTeX code into the file whose name is specified by the string argument. Two examples for such automatically generated proofs are provided in the appendix (see page 25).

The author considers the Proof function as an experimental toy for playing around, but not as principal part of the package. It is not decided if the Proof function will be kept in future releases of the package.

A possible application of the idea of proof objects is to implement brothers and sisters of TeXify which are able to transform the proof into the form of an automatic proof checker, but we are not willing to waste too much time into implementations of this kind.

5.5.2 Some Sum Manipulation Tools

ZET is unable to handle definite sums directly. A definite sum is a sum in which the bounding index appears in the summand. The difficulty is that such sums $F(n) = \sum_{k=1}^{n} f(n,k)$ does not necessarily obey the simple relation $F(n+1) = F(n) + f(n, n+1)$. More delicate techniques are necessary to compute a recurrence that may serve as defining relation for a definite sum, but such techniques are not implemented in ZET.

However, in some simple instances it is possible to make definite sums indefinite by just applying basic arithmetic laws to the sum. This is particularly useful for

treating polynomial identities. A standard example is

$$\sum_{k=1}^{n}(n+k)^2 = n^2\sum_{k=1}^{n}1 + 2n\sum_{k=1}^{n}k + \sum_{k=1}^{n}k^2.$$

Rewritings of the above shape can be automatically carried out by the package's function MakeDefiniteSumsIndefinite which takes any expression as first argument and attempts to rewrite it in terms of indefinite sums by applying basic rewrite rules. Note that this method may run for a long time already for input of moderate size, and that there is no guarantee that the output is indeed indefinite.

The option Simplify (True or False, False by default) indicates if simplification rules should be applied during the computation. This may help to avoid expression swell, but sometimes increases the runtime.

6 Internals and Technicalities

This section describes some internal details which will most probably not be of any interest for most users. They are mainly included for the sake of completeness.

6.1 The sequence database

Definitions of sequences are maintained in an internal database. Every entry of this database consists of the following data:

- A unique identification integer for the sequence, called the "id". Ids are not necessarily assigned consecutively.
- A defining relation in terms of a difference polynomial. A sequence is free iff its defining relation is the zero polynomial.
- A startpoint. This specifies the smallest integer for which the sequence is defined, according to the initial values given in its definition. If no initial values are specified, the startpoint takes the value $-$Infinity.
- An evaluator. This is a function that efficiently computes the value of a sequence at an integer point by repeatedly applying the defining recurrence until the initial values are reached.
- An order. This is the maximum shift occurring in the defining relation of the sequence.
- An effective order. This is the difference between minimum and maximum shift occurring in the defining relation of the sequence.
- A list of names under which the sequence is known to the user.

The defining relations of the sequences in the database are elements of an m-fold polynomial difference ring $F\{y^{(1)}, y^{(2)}, \ldots, y^{(m)}\}$ [2]. They are represented internally by means of the package private variable t: $t[i,j]$ represents the ith shift of $y^{(j)}$, where j refers to the id of the sequence.

Similarly, the evaluators are implemented by means of the package private variable T. The expression $T[i,j]$ is defined such as to evaluate to the value of the sequence with id j at the point i. It remains unevaluated if the queried function

value is not available. It is GetValue's task to replace subexpressions $T[i, j]$ by user readable function symbols after an evaluation process.

The maintainance of the sequence database proceeds via a couple of Get and Set functions. All this is package private and invisible for the user.

Get functions

Each of these functions has the option DefineIfUndefined which is by default True for GetId and False for the other functions. In this case, reference to an undefined sequence defines this sequence to be free. If set to False, the data base is left unchanged and $Failed is returned if the element did not exist.

- GetId[*name*] — The unique id of the sequence with the given name.
- GetRelation[*id*] — Gives the defining relation of the specified sequence.
- GetStartpoint[*id*] — Gives the startpoint of a sequence, possibly −Infinity.
- GetOrder[*id*] — Gives the order of the specified sequence, i.e. the maximum number m such that $t[m, id]$ occurs in the defining relation. The order of 0 is −Infinity.
- GetEffectiveOrder[*id*] — Gives the effective order of the specified sequence, i.e. the difference $m - n$ where m is the maximum number such that $t[m, id]$ occurs in the defining relation and n is the minimum number such that $t[n, id]$ occurs in the defining relation. The effective order of free sequences is 0.
- GetNames[*id*] — Gives the names associated to the specified sequence.

Set functions

Each of these functions has the option OverwriteMessage which is by default set to False. If set to True, a message is raised if existing information is overridden.

- AddDefinition[*id*, *listOfNames*, *definingRelation*, *startpoint*] — Adds a definition to the database.
- SetRelation[*id*, *defrel*] — (re)defines the defining relation for the specified sequence.
- SetStartpoint[*id*, *defrel*] — (re)defines the startpoint of the specified sequence.
- SetNames[*id*, *listOfNames*] — (re)defines the names associated to the specified sequence.
- AddNames[*id*, *listOfNames*] — associates additional names to a particular sequence, removing bindings of the names to other sequences.

Further functions

- RemoveNames[*listOfNames*] — removes the associations of the given names to their sequences.
- RemoveDefinition[*id*] — Removes a definition from the database. Caution: Calls of this function may lead to dangling references. Only the raw entry is removed, all other entries will remain unchanged.

- RemoveDefinitionIfUseless[id] — removes the entry corresponding to the sequence with the given id unless there are other sequences whose defining relation contains a term $t[\_, id]$.
- GetFreshId[] — a free id that can be used for the next definition.
- MaximumId — maximum id in use

6.2 Bug Parade

The package is in a very early stage, and though it is continuously tested by the author, we are sure that it contains a lot of additional bugs.

At the time of writing, no bugs are known.

If you happen to encounter an unexpected behavior while using the package, in particular if you succeed in proving something which is wrong or in disproving something which is true, please send a detailed bug report to manuel@kauers.de.

When doing so, don't forget to include the package version you are using. This data is printed when you load the package.

Appendix

A Example Gallery

A.1 Exercise 6.61

Exercise 6.61 in [3] asks for a proof of the identity

$$\sum_{k=0}^{n} \frac{1}{\mathrm{Fib}(2^k)} = 3 - \frac{\mathrm{Fib}(2^n - 1)}{\mathrm{Fib}(2^n)}$$

where $\mathrm{Fib}(n)$ denotes the nth Fibonacci number. As the Fibonacci sequence is C-finite, we can directly enter the requested identity into ZET:

In[1]:= $\mathrm{IdenticalQ}[\sum_{k=0}^{n} \frac{1}{\mathrm{Fib}[2^k]} == 3 - \frac{\mathrm{Fib}[2^n - 1]}{\mathrm{Fib}[2^n]}, \mathrm{ForAll} \to n]$

Out[1]= True

This is a full solution of the exercise.

A.2 Exercise 5.93

Exercise 5.93 in [3] asks for a closed form of the indefinite sum $\sum_{k=1}^{n} \frac{\prod_{1=1}^{k}(x_i+\alpha)}{\prod_{i=1}^{k} x_i}$ for arbitrary $x_1, x_2, \ldots$ and arbitrary α. ZET is not able to find the requested closed form, but once the answer

$$\sum_{k=1}^{n} \frac{\prod_{1=1}^{k} (x_i + \alpha)}{\prod_{i=1}^{k} x_i} = \frac{1}{\alpha}\Big(\prod_{k=1}^{n} \frac{x_k + \alpha}{x_k} - 1\Big)$$

is conjectured, ZET is able to prove it effortlessly.

In[1]:= $\mathrm{IdenticalQ}[\sum_{k=1}^{n} \frac{\prod_{1=1}^{k} (x[i] + \alpha)}{\prod_{i=1}^{k} x[i]} = \frac{1}{\alpha}\Big(\prod_{k=1}^{n} \frac{x[k] + \alpha}{x[k]} - 1\Big), \mathrm{ForAll} \to n]$

Zet::defarbitrary : Definition involves undefined function symbol x which will be regarded as free.

Zet::arbitrary : Conjecture involves arbitrary sequences. Algorithm may not terminate.

Out[1]= True

The first warning says that there is no defining relation available for the function symbol x. This message is raised when x is inserted into the database. As a consequence, the proving algorithm may not terminate, as indicated by the second warning.

By coincidence, it did terminate in this particular example. In case it didn't, we could force termination by using the Field option.

In[2]:= $\mathrm{IdenticalQ}[\sum_{k=1}^{n} \frac{\prod_{1=1}^{k} (x[i] + \alpha)}{\prod_{i=1}^{k} x[i]} = \frac{1}{\alpha}\Big(\prod_{k=1}^{n} \frac{x[k] + \alpha}{x[k]} - 1\Big), \mathrm{ForAll} \to n, \mathrm{Field} \to \mathrm{True}]$

Out[2]= True

The first warning from before doesn't appear because x is already in the database, and the second warning doesn't appear because there is no danger of nontermination due to the usage of the Field option.

A.3 The Christoffel-Darboux identity for orthogonal polynomials

Given two sequences $(c_i)_{i=1}^{\infty}, (\lambda_i)_{i=1}^{\infty}$, the sequence of univariate polynomials in x defined by the recurrence

$$P_{n+2}(x) = (x - c_{n+2})P_{n+1}(x) - \lambda_{n+2}P_n(x), \qquad p_{-1}(x) = 0, p_0(x) = 1$$

constitutes a family of orthogonal polynomials [1].

For any family of orthogonal polynomials, we have the Christoffel-Darboux identity

$$\sum_{k=0}^{n} \frac{p_k(u)p_k(x)}{\prod_{i=1}^{k+1} \lambda_i} = \frac{p_{n+1}(x)p_n(u) - p_n(x)p_{n+1}(u)}{(x-u)\prod_{i=1}^{n+1} \lambda_i}.$$

ZET is able to prove this identity in full generality. We take $\mathbb{Q}(x, u)$ as field of constants where x and u are transcendental elements, and we define two sequences $(P_n(u))_{n=-1}^{\infty}$, $(P_n(x))_{n=-1}^{\infty}$ by the recurrence above. $(c_i)_{i=1}^{\infty}, (\lambda_i)_{i=1}^{\infty}$ will be treated as free sequences.

In[1]:= DefineSequence[$px[n+2] == (x - c[n+2]) * px[n+1] - \lambda[n+2] * px[n]$, $px[-1] == 0, px[0] == 1$]

Zet::arbitrary : Definition involves undefined function symbol c which will be regarded as free.

Zet::arbitrary : Definition involves undefined function symbol λ which will be regarded as free.

In[2]:= DefineSequence[$pu[n+2] == (u - c[n+2]) * pu[n+1] - \lambda[n+2] * pu[n]$, $pu[-1] == 0, pu[0] == 1$]

In[3]:= $cd = \sum_{k=0}^{n} \frac{pu[k] * px[k]}{\prod_{i=1}^{k+1} \lambda[i]} = \frac{px[n+1] * pu[n] - px[n] * pu[n+1]}{(x-u)\prod_{i=1}^{n+1} \lambda[i]}$;

In[4]:= IdenticalQ[cd, ForAll $\to n$, Field $\to$ True]

Out[4]= True

In[5]:= IdenticalQ[cd, ForAll $\to n$]

Zet::arbitrary : Conjecture involves arbitrary sequences. Algorithm may not terminate.

Out[5]= True

There is also the following confluent version of the Christoffel-Darboux identity:

$$\sum_{k=0}^{n} \frac{p_k(x)^2}{\prod_{i=1}^{k+1} \lambda_i} = \frac{p_n(x)p'_{n+1}(x) - p_{n+1}(x)p'_n(x)}{\prod_{i=1}^{n+1} \lambda_i}$$

This version can also be verified by ZET. A recurrence for the derivatives $p'_n(x)$ can be found by differentiating the recurrence for $p_n(x)$.

In[6]:= DefineSequence[
$Dpx[n+2] == px[n+1] + (x - c[n+2]) * Dpx[n+1] - \lambda[n+2] * Dpx[n]$,
$Dpx[-1] == 0, Dpx[0] == 0$]

In[7]:= $\text{IdenticalQ}[\sum_{k=0}^{n} \frac{px[k]^2}{\prod_{i=1}^{k+1} \lambda[i]} == \frac{px[n]*Dpx[n+1]-px[n+1]*Dpx[n]}{\prod_{i=1}^{n+1} \lambda[i]}, \text{ForAll} \to n]$

Zet::arbitrary : Conjecture involves arbitrary sequences. Algorithm may not terminate.

Out[7]= True

A.4 Exercise 5.12

The next example is taken form [1], Exercise 5.12. It asks to prove the identity

$$\frac{\varkappa_1(n)}{\varkappa_2(n)} = x_0 + \sum_{k=1}^{n} \frac{(-1)^{k+1} \prod_{i=1}^{k} y_i}{\varkappa_2(k-1)\varkappa_2(k)}$$

where $\varkappa_1$ and $\varkappa_2$ denote the continuants defined by the recurrences

$$\varkappa_1(n+2) = x_{n+2}\varkappa_1(n+1) + y_{n+2}\varkappa_1(n) \qquad \varkappa_1(0) = x_0, \varkappa_1(1) = x_0x_1 + y_1$$
$$\varkappa_2(n+2) = x_{n+2}\varkappa_2(n+1) + y_{n+2}\varkappa_2(n) \qquad \varkappa_2(0) = 1, \varkappa_2(1) = x_1$$

where $x_0, x_1, \ldots$ and $y_1, y_2, \ldots$ are arbitrary sequences.

We have ZET do the exercise.

In[1]:= DefineSequence[
$k1[n+2] == x[n+2]*k1[n+1] + y[n+2]*k1[n]$,
$k1[0] == x[0], k1[1] == x[0]*x[1] + y[1]$];

Zet::defarbitrary : Definition involves undefined function symbol y which will be regarded as free.

Zet::defarbitrary : Definition involves undefined function symbol x which will be regarded as free.

In[2]:= DefineSequence[
$k2[n+2] == x[n+2]*k2[n+1] + y[n+2]*k2[n]$,
$k2[0] == 1, k2[1] == x[1]$];

In[3]:= $\text{IdenticalQ}[\frac{k1[n]}{k2[n]} = x[0] + \sum_{k=1}^{n} \frac{(-1)^{k+1} \prod_{i=1}^{k} y_i}{k2[k-1]k2[k]}, \text{ForAll} \to n]$

Zet::arbitrary : Conjecture involves arbitrary sequences. Algorithm may not terminate.

Out[3]= True

A.5 Disproving, Order messages, and Startpoints

ZET is also able to discover that an identity does not hold if this is the case. Unless in obvious examples, this requires much more resources in general.

Example: Consider the "identity"

$$(n-3)(n-7)\sum_{k=1}^{n}(-1)^k\left(1+\prod_{k=1}^{n}(-1)^k\right) = 0$$

which holds for $n = 1, 2, 3, 4, 5, 6, \ldots$ Does it hold for all n?

In[4]:= $\text{id} = (n-3)(n-7)\sum_{k=1}^{n}(-1)^k\left(1+\prod_{k=1}^{n}(-1)^k\right) == 0;$

In[5]:= $\text{IdenticalQ}[\text{id}, \text{ForAll} \to n]$

```
Zet::maxordex :  Maximum order 6 reached without having found an answer
```

Out[5]= Unknown

ZET's proving algorithm consists of a main loop which iterates until a proof has been found. It is known that this loop will eventually terminate, but it is now known how many iterations are necessary. The higher this number is, the more time will be spent for each single iteration, and it becomes more and more unlikely that an answer is found in the user's lifetime. Therefore, ZET has a threshold number and it gives up once the number of iterations exceeds this bound. This has happened in the example above.

The option MaxOrder redefines the threshold number. It may be set to an arbitrary integer, or to the value Infinity.

In[6]:= IdenticalQ[id, ForAll → n, MaxOrder → Infinity]

Out[6]= False

The OrderMessage option enables to print progress report messages.

In[7]:= IdenticalQ[id, ForAll → n, MaxOrder → Infinity, OrderMessage → True]

```
Considering order 0...
Considering order 1...
Considering order 2...
Considering order 3...
Considering order 4...
Considering order 5...
Considering order 6...
Considering order 7...
Considering order 8...
Considering order 9...
```

Out[7]= False

If it is known for some reason that the number of iterations will be greater than, say, 5, then time can be saved by skipping the first iterations of the loop. The option StartOrder specifies the first iteration to be tried.

In[8]:= IdenticalQ[id, ForAll → n, MaxOrder → Infinity,
OrderMessage → True, StartOrder → 5]

```
Considering order 5...
Considering order 6...
Considering order 7...
Considering order 8...
Considering order 9...
```

Out[8]= False

Another strategy for saving time is to ask for proving the identity from a starting point different from 1 on. In our particular example, this runs much faster than the previous calls because the starting point is closer to a counterexample.

In[9]:= IdenticalQ[id, ForAll → n, From → 10]

Out[9]= False

The From option is also useful for proving identities which are valid only from some point on. For example, proving

$$\prod_{k=1}^{n}(k-3)\sum_{i=1}^{k}\mathrm{Fib}(i)=0$$

will not work, because this identity does not hold for all $n \in \mathbb{N}$. It does, however, hold for $n \geq 3$.

In[10]:= $t = \prod_{k=1}^{n}(k-3) * \sum_{i=1}^{k}\mathrm{Fib}[i];$

In[11]:= $\mathrm{ZeroQ}[t, \mathrm{ForAll} \to n]$

Out[11]= False

In[12]:= $\mathrm{ZeroQ}[t, \mathrm{ForAll} \to n, \mathrm{From} \to 3]$

Out[12]= True

B Automatically Generated Proofs

The following proofs were automatically generated by the calls of the form

In[13]:= $\mathrm{TeXify}[\mathrm{Proof}[\sum_{i=1}^{k} i = \frac{1}{2}k(k+1), \mathrm{ForAll} \to k, \mathrm{Cofactors} \to \mathit{function}], \mathit{file}]$

B.1 Gauß' sum

Theorem The identity

$$\sum_{i=1}^{k} i = \frac{k\ (1+k)}{2}$$

holds for all $k \geq 1$. More precisely, we consider the sequence $(f_1(k))_{k=1}^{\infty}$ defined by the recurrence relation

$$\frac{-k-k^2-2\,f_1(k)+2\,f_2(k)}{2}=0, \tag{1}$$

where

$$1+k+f_2(k)-f_2(1+k)=0, \quad f_2(1)=1, \tag{2}$$

and we claim that $f_1(k) = 0$ for all $k \geq 1$.

Proof We proceed by induction on k. As base of the induction, observe the following evaluations

k	1
$f_1(k)$	0
$f_2(k)$	1

.

For the induction step, suppose there exists a $k \geq 1$ such that $f_1(k) = 0$. We will show that this implies $f_1(1+k) = 0$.
Assume otherwise. Then we can define $X = 1/f_1(1+k)$. Equations (1)–(2) hold for all $k \geq 1$, so they also hold for $k+1$, $k+2$, ... This immediately implies

$$\frac{-2 - 3\,k - k^2 - 2\,f_1(1+k) + 2\,f_2(1+k)}{2} = 0. \tag{3}$$

Multiplying Eq. (1) by X gives

$$\frac{-\left(X\,\left(k + k^2 + 2\,f_1(k) - 2\,f_2(k)\right)\right)}{2} = 0. \tag{4}$$

Multiplying Eq. (2) by $-X$ gives

$$-\left(X\,\left(1 + k + f_2(k) - f_2(1+k)\right)\right) = 0. \tag{5}$$

Multiplying Eq. (3) by $-X$ gives

$$\frac{X\,\left(2 + 3\,k + k^2 + 2\,f_1(1+k) - 2\,f_2(1+k)\right)}{2} = 0. \tag{6}$$

Furthermore: Multiplying $f_1(k) = 0$ from the induction hypothesis by X gives

$$X\,f_1(k) = 0 \tag{7}$$

Finally, multiplying $X = 1/f_1(1+k)$ by $-f_1(1+k)$ and subtracting the left hand side gives

$$1 - X\,f_1(1+k) = 0. \tag{8}$$

Adding up equations (4)–(8) gives, after some simplification,

$$1 = 0,$$

contradicting the assumption $f_1(1+k) \neq 0$. Hence, $f_1(1+k) = 0$, and this completes the proof.

B.2 The multinomial theorem for exponent 2

Theorem The identity

$$\left(1 + \sum_{i=1}^{n} x(i)\right)^2 = 1 + 2\sum_{i=1}^{n}\left(x(i)\sum_{j=1}^{-1+i} x(j)\right) + 2\sum_{i=1}^{n} x(i) + \sum_{i=1}^{n}\left(x(i)^2\right)$$

holds for all $n \geq 1$. More precisely, we consider the sequence $(f_1(n))_{n=1}^{\infty}$ defined by the recurrence relation

$$-f_1(n) - f_2(n) - 2\,f_3(n) + f_4(n)^2 = 0, \tag{1}$$

where

$$x(1+n)^2 + f_2(n) - f_2(1+n) = 0, \quad f_2(1) = x(1)^2, \tag{2}$$
$$f_3(n) - f_3(1+n) + x(1+n)\,f_4(n) = 0, \quad f_3(1) = 0, \tag{3}$$
$$x(1+n) + f_4(n) - f_4(1+n) = 0, \quad f_4(1) = x(1), \tag{4}$$

and we claim that $f_1(n) = 0$ for all $n \geq 1$.

Proof We proceed by induction on n. As base of the induction, observe the following evaluations

n	1
$f_1(n)$	0
$f_2(n)$	$x(1)^2$
$f_3(n)$	0
$f_4(n)$	$x(1)$

.

For the induction step, suppose there exists a $n \geq 1$ such that $f_1(n) = 0$. We will show that this implies $f_1(1+n) = 0$.

Assume otherwise. Then we can define $X = 1/f_1(1+n)$. Equations (1)–(4) hold for all $n \geq 1$, so they also hold for $n+1$, $n+2$, ... This immediately implies

$$-f_1(1+n) - f_2(1+n) - 2\,f_3(1+n) + f_4(1+n)^2 = 0. \tag{5}$$

Multiplying Eq. (1) by X gives

$$-\left(X\,\left(f_1(n) + f_2(n) + 2\,f_3(n) - f_4(n)^2\right)\right) = 0. \tag{6}$$

Multiplying Eq. (2) by X gives

$$X\,\left(x(1+n)^2 + f_2(n) - f_2(1+n)\right) = 0. \tag{7}$$

Multiplying Eq. (3) by $2\,X$ gives

$$2\,X\,\left(f_3(n) - f_3(1+n) + x(1+n)\,f_4(n)\right) = 0. \tag{8}$$

Multiplying Eq. (4) by $-\left(X\,\left(x(1+n) + f_4(n) + f_4(1+n)\right)\right)$ gives

$$-\left(X\,\left(x(1+n) + f_4(n) - f_4(1+n)\right)\,\left(x(1+n) + f_4(n) + f_4(1+n)\right)\right) = 0. \tag{9}$$

Multiplying Eq. (5) by $-X$ gives

$$X\,\left(f_1(1+n) + f_2(1+n) + 2\,f_3(1+n) - f_4(1+n)^2\right) = 0. \tag{10}$$

Furthermore: Multiplying $f_1(n) = 0$ from the induction hypothesis by X gives

$$X\,f_1(n) = 0 \tag{11}$$

Finally, multiplying $X = 1/f_1(1+n)$ by $-f_1(1+n)$ and subtracting the left hand side gives

$$1 - X\,f_1(1+n) = 0. \tag{12}$$

Adding up equations (6)–(12) gives, after some simplification,

$$1 = 0,$$

contradicting the assumption $f_1(1+n) \neq 0$. Hence, $f_1(1+n) = 0$, and this completes the proof.

B.3 An automated disproof

Theorem The identity

$$(-5+n)\,(-4+n)\,(-3+n)\,(-2+n)\,(-1+n) = 0$$

does not hold for all $n \geq 1$.

Proof We have the counterexample

$$\mathrm{lhs}(6) = 120 \neq 0 = \mathrm{rhs}(6).$$

C Summary of Functions

We summarize the most important parts of the specification of the functions defined by the package. For details, consult the main part of this documentation.

- ClearDefinitions[ ]

  Removes all definitions from the database.

- DefineSequence[$rec, init_1, init_2, \ldots$]

  Adds a new definition to the database. The sequence may be defined by a recurrence and/or initial values.

  If a recurrence is given, it has to be the first argument.

  If the recurrence is missing or no initial value is given, the sequence will evaluate to symbolic values.

  Sequences without defining recurrence and without initial values are implicitly defined when they appear for the first time on the right hand side of a recurrence. A warning is printed in order to inform the user about the introduction of a free sequence.

  If the definition fails because the recurrence cannot be translated into an appropriate internal format, a corresponding warning message is raised, and the function call returns the value $Failed.

- GetValue[$expr, var \to value$]

  Evaluates *expr*, considered as a sequence in *var* at the point $var = value$.

  The expression *expr* has to be such that a call

  $$\text{DefineSequence}[newname[var] = expr]$$

  successful.

  If *expr* involves sequence which can not be evaluated because they are free or not enough initial values are given, then the evaluation will be done in a (possibly partially) symbolic fashion.

- IdenticalQ[$expr_1 == expr_2, \text{ForAll} \to var$]

  Tests if $expr_1$ and $expr_2$, considered as sequences in *var*, are identical on the natural numbers.

  The ForAll option is obligatory, further options are listed on page 5.4.2.

  This function internally calls ZeroQ on $expr_1 - expr_2$ with the supplemented set of options.

- MakeDefiniteSumsIndefinite[*expr*]

  Attempts to apply rewrite rules to the given expression *expr* in order to write definite sums in terms of indefinite sums.

  This is very elementary, it applies basic algebraic laws like the distributive law for pulling factors independent of the summation variable out in front of the sum, etc. In general, this method fails, but in case of simple polynomial expressions it proved useful.

  The option Simplify, when set to True, causes the application of rewrite rules for simplifying the result during the computation. This can reduce the expression swell considerably.

- Proof[$expr_1 == expr_2$, ForAll $\to$ var, Cofactors $\to$ $function$]

  Constructs a proof object proving the identity $expr_1 = expr_2$ for all var.

  The proof object may be used by functions like TeXify for further processing.

  Proof objects are available also if the identity to be proven does not hold. In this case, the proof consists of a counterexample.

  The options ForAll and Cofactors are obligatory. The meaning of the latter is described on page 16, while the former specifies the running variable, e.g., n. In addition, all options accepted by IdenticalQ and ZeroQ are admissible also for the Proof function.

- Prove[$expr$, ForAll $\to$ var]

  Synonym for

  $$\text{IdenticalQ}[expr, \text{ForAll} \to var]$$

  if $expr$ is an equation ($\_ == \_$), and for

  $$\text{ZeroQ}[expr, \text{ForAll} \to var]$$

  otherwise.

- TeXify[$proof$, $filename$]

  Creates a human readable form of a *proof* object composed by the Proof function in form of a LaTeX document whose code is written into the file with the specified *filename*.

- ZeroQ[$expr$, ForAll $\to$ var]

  Synonym for IdenticalQ[$expr == 0$, ForAll $\to$ var]

References

[1] George E. Andrews, Richard Askey, and Ranjan Roy. *Special Functions*, volume 71 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1999.

[2] Richard Moses Cohn. *Difference Algebra*. Interscience Publishers, John Wiley & Sons, 1965.

[3] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[4] Manuel Kauers. An algorithm for deciding zero equivalence of nested polynomially recurrent sequences. Technical Report 2003-48, SFB F013, Johannes Kepler Universität, 2003. (submitted).

[5] Manuel Kauers. Computer proofs for polynomial identities in arbitrary many variables. In *Proceedings of ISSAC '04*, 2004.

Index