Implementation of a Recurrence Solver via Discrete Operational Calculus

John Cyphert
Universtiy of Wisconsin
Email: jcyphert@wisc.edu

***Abstract*—This paper presents a method and implementation of a recurrence solver based on the operational calculus algebra created by Lothar Berg. The method in this paper is able to handle a subset of linear non-homogeneous recurrences with constant coefficients. The system that implements this method of solving recurrence is named OCRS (Operational Calculus Recurrence Solver) and is able to solve first order linear non-homogeneous recurrences with constant coefficients. Preliminary experimentation show that OCRS is able to solve a smaller class of recurrences as compared to existing tools; however, since OCRS only contains capabilities to solve recurrences it often much faster in solving recurrences compared with other computer algebra systems (CAS).**

1. Introduction

Researchers and developers desire to know what numerical properties their programs obey. For many control statements this task is not too difficult. One must be able to extend and join program transformations to generate an appropriate model. However, this task becomes more difficult when dealing with loop and recursion structure, due to a loop possibly running an indeterminate number of times.

For example consider analyzing the program to the right. We seek an over-approximation of for the value of i. At the first line we see the statement that i gets 0, so $i \in [0,0]$. We then must go through a fixed number of iterations of the loop and update the interval of i until convergence. If the analysis doesn't converge within the perdetermined number of iterations, then a widening operator must be applied. In this case after a certain number of iterations i must be widened to $[0,\infty]$ since the value of i increases on each iteration. This is not an incorrect statement. This interval is the best approximation for i since the loop iterates an unknown number of times. However, some information is lost. Namely, if we knew how many times the loop iterated we would have a much better sense on the value of i. What we would like is to create a summary for the loop based on the number of iterations.

```
int i = 0;
while (*) {
  i = i + 1;
}
```

This process of creating loop summaries is studied in [5] and [1] as well as others. Often in these works the procedure to create loop summaries has two steps. First recurrences must be extracted from a loop, and then those recurrences must be solved. This can be done compositionally.

Consider the program to the right. We could extract the recurrence for the inner loop as $cost_{j+1} = cost_j + 1$, and then solve that recurrence and the summarize the inner loop as $cost_i = cost_0 + i$. Then we take that result as the recurrence for the outer loop. We have $cost_{i+1} = cost_i + i$. Solving this yields the result $cost_n = cost_0 + \binom{n}{2}$.

```
int i, j;
for (i = 0; i<n; i++){
  for (j = 0; j<i; j++){
    cost = cost + 1;
  }
}
```

There are tools that accomplish this task such as described in [5]. However, the recurrence solvers in these tools are limited. There are powerful recurrence solvers[6] that could fulfill the capability deficiency. However, they are often a part of larger computer algebra systems, and thus are unnecessarily slow.

This is the problem addressed by this paper. We seek to build a symbolic recurrence solver that is powerful enough to solve recurrences that arise from real programs. Also since the program analysis tools must repeatedly call a recurrence solver the system needs to be lightweight and fast enough to not dominate the running time of the analysis tool.

The rest of the paper is laid out as follows. Section 2.1 gives some basic recurrence terminology as well as defining some different classes of recurrences. Sections 2.2, 2.3 and 2.4 introduce the mathematical framework of a particular method of solving recurrences. Section 3 gives an overview of the method used to solve recurrences as well as contains example recurrences solved by this method. Section 4 gives implementation details of the Operational Calculus Recurrence Solver. Section 5 details some preliminary results from the system. And finally Section 6 outlines some extensions and future work.

2. Background

2.1. Recurrence Relation Definitions

A recurrence relation can be loosely defined as a infinite sequence with a recursive definition. However, the recursive definition is not the only way to define a sequence. For example consider the following recurrence definition:

$$a_{n+1} = a_n + n,\ a_0 = 0$$

This recurrence relations can be represented by the sequence $< 0, 0, 1, 3, 6, 10, ... >$. This sequence can also be defined by $(\frac{n^2}{2} - \frac{n}{2})_{n\in\mathbb{N}}$ often written as just $a_n = \frac{n^2}{2} - \frac{n}{2}$. This second representation of the sequence is called the *closed form* of a_n. There are many applications of both the closed form and the recursive form of a sequence. Our main concern though is being able to take a recurrence in a recursive form and retrieving its closed form. This process will be referred as *solving* a recurrence.

Definition 1. A recurrence in the following form:

$$\alpha_k y_{n+k} + \alpha_{k-1} y_{n+k-1} + \alpha_{k-2} y_{n+k-2} + ... + \alpha_0 y_n = f(n) \tag{1}$$

where k is fixed in $\mathbb{N}$ is *linear with order k*. A recurrence that is not in the above form is *non-linear*.

The following are examples of linear recurrences:

- $y_{n+1} = y_n + 1 \Leftrightarrow y_n = y_0 + n$
- $y_{n+1} = 2y_n \Leftrightarrow y_n = y_0 * 2^n$
- $y_{n+2} = y_{n+1} + y_n,\ y_0 = 1,\ y_1 = 1 \Leftrightarrow y_n = \frac{\phi^{n+1} - \psi^{n+1}}{\sqrt{5}}$

Whereas the following examples are non-linear.

- $y_n = y_{\frac{n}{2}} + 1 \Leftrightarrow y_n = y_0 + \log_2 n$
- $y_n = y_{\frac{n}{2}} + n \Leftrightarrow y_n = 2n - 2 + y_0$
- $y_n = 2y_{\frac{n}{2}} + n \Leftrightarrow y_n = n \log_2 n + \frac{y_0 n}{2}$
- $y_n = y_{\sqrt{n}} + 1 \Leftrightarrow y_n = \log_2 \log n + y_0$

Definition 2. Consider a recurrence of the form in (1). If $f(n) = 0$ then the recurrence is a *homogeneous* recurrence. Otherwise the recurrence is *non-homogeneous*.

The distinction between homogeneous and non-homogeneous recurrences is an important one, because there exists multiple direct methods for solving homogeneous recurrences. Non-homogeneous recurrences on the other hand require stronger methods to solve. In some instances there exists transforms to translate non-homogeneous recurrences to equivalent homogeneous versions; however, these transforms are not applicable in general.

Recurrences can also be classified by the coefficients on the variable terms.

Definition 3. Consider a recurrence of the form in (1). If every α_i for $i = 0, ..., k$ is a constant then the recurrence has *constant* coefficients. If however there exists an α_i that is not constant then the recurrence has *variable* coefficients.

2.2. Berg's Operational Calculus Field

Operational calculus refers to a technique of solving analysis problems by creating a suitable algebra for operators. One can then determine an analogous algebra problem for the original analysis problem. If a solution is found for the algebra problem one is then found for the original analysis problem. This technique has applications in solving differential equations, where the differential operator is thought of as an element of an algebra as opposed to an operator on objects.

The "Operational Calculus of a Discrete Variable" is a specific operational calculus algebra defined by Lothar Berg in [2, Chapter 2]. This algebra is formalized as a field $\mathfrak{R}$. The elements of the field are infinite sequences in both directions with finitely many non-zero elements in negative indexes.

More formally the elements of $\mathfrak{R}$ is the following set:

$$\{z =< ..., z_{-2}, z_{-1} || z_0, z_1, z_2, ... > |\ z_j \in \mathbb{R} \text{ and } \exists N \in \mathbb{Z}, \forall i < N, z_i = 0\}$$

Addition (+) in $\mathfrak{R}$ is the point-wise sum of the two sequences.

$$\begin{aligned} a_n &=< ..., a_{-2}, a_{-1} || a_0, a_1, ... > \\ b_n &=< ..., b_{-2}, b_{-1} || b_0, b_1, ... > \\ a_n + b_n &=< ..., a_{-2} + b_{-2}, a_{-1} + b_{-1} || a_0 + b_0, a_1 + b_1, ... > \end{aligned}$$

The zero element, $\mathbf{0}$, is the sequence with zeros in all positions.

At the moment we are not equipped to fully define the multiplication operator $\cdot$. First we must observe the operational calculus part of this algebra. Consider the following two functions of field elements:

$$\begin{aligned} \text{Let } a_n &=< ..., a_{-2}, a_{-1} ||\ a_0, a_1, ... > \\ LeftShift(a_n) &=< ..., a_{-1},\ a_0 ||\ a_1, a_2, ... > \\ RightShift(a_n) &=< ..., a_{-3}, a_{-2} || a_{-1}, a_0, ... > \end{aligned}$$

We will now introduce two new elements in $\mathfrak{R}$, q and v. Multiplication by these elements will have the effect of the previous shifts. Namely these elements have the following properties:

- $q \cdot a_n = a_n \cdot q = LeftShift(a_n)$
- $v \cdot a_n = a_n \cdot v = RightShift(a_n)$
- $q^m \cdot a_n = LeftShift(...(LeftShift(a_n)))$ m times
- $v^m \cdot a_n = RightShift(...(RightShift(a_n)))$ m times

Note that since $LeftShift$ and $RightShift$ are inverses of each other, $q \cdot v = v \cdot q = \mathbf{1}$, which is equal to $< ..., 0, 0, 0 || 1, 1, 1, ... >$ in $\mathfrak{R}$.

Therefore q and v can be represented in the following way:

$$\begin{aligned} q &=< ..., 0, 0, 1 || 1, 1, 1, ... > \\ v &=< ..., 0, 0, 0 || 0, 1, 1, ... > \end{aligned}$$

We are now equipped to define multiplication in the field $\mathfrak{R}$. First we will only define multiplication for elements that have only 0's in negative indexes. Call this set of elements $\mathfrak{d}$. Multiplication in this context is a convolution difference. The n'th value of the product is the following:

$$(a \cdot b)_n = \sum_{\nu=0}^{n} a_\nu b_{n-\nu} - \sum_{\nu=0}^{n-1} a_\nu b_{n-1-\nu} \tag{2}$$

This defines multiplication for all elements in $\mathfrak{d}$, but it is not obvious how to define multiplication by an element that is in $\mathfrak{R}$ but not in $\mathfrak{d}$. This can be resolved due to the commutative

and associative properties of $\cdot$ and the introduction of q and v. Consider an element

$$c =< ..., 0, 0, ..., c_{-k}, c_{-k+1}, ..., c_{-1}||c_0, c_1, ... >$$

for $k \in \mathbb{N}$ where $c_{-k} \neq 0$. Now consider a sequence $x \in \mathfrak{d}$. Then:

$$\begin{aligned} c \cdot x &= 1 \cdot c \cdot x \\ &= (q^k \cdot v^k) \cdot c \cdot x \\ &= q^k \cdot (v^k \cdot c) \cdot x \end{aligned}$$

Now $v^k \cdot c \in \mathfrak{d}$, so $(v^k \cdot c) \cdot x$ can be evaluated by (2) and the result can be left shifted k times by multiplying by q^k. Thus multiplication is defined for $\mathfrak{K}$.

2.3. Properties of the Field

Since $\mathfrak{K}$ comes equipped with operators to shift sequences we are able to relate sequences which are off by a constant. Consider the following sequences:

$$\begin{aligned} a_n &=< ..., 0, 0||a_0, a_1, a_2, ... > \\ a_{n+1} &=< ..., 0, 0||a_1, a_2, a_3, ... > \\ \text{so } qa_n &=< ..., 0, a_0||a_1, a_2, a_3, ... > \end{aligned}$$

If we could remove the value in the -1 position of qa_n we would have exactly a_{n+1}, and thus be able to relate sequences which are off by one. It can be verified by (2) that

$$\begin{aligned} (q-1)a_0 =< ..., 0, 0, a_0||0, 0, 0, ... > \text{ where} \\ a_0 =< ..., 0, 0, 0||a_0, a_0, a_0, ... > \end{aligned}$$

Therefore we have the following relationship:

$$a_{n+1} = qa_n - (q-1)a_0$$

Or in general

$$a_{n+m} = q^m a_n - (q-1)\sum_{\mu=0}^{m-1} a_{m-\mu-1} q^\mu \tag{3}$$

This allows us to relate common sequences we are familiar with in elementary algebra to elements in the operational calculus. For example consider the functional sequence $\alpha^n =< ...0, 0, 0||\alpha^0, \alpha^1, \alpha^2, ... >$ for $n \in \mathbb{N}$ and $\alpha \in \mathbb{R}$.

$$\begin{aligned} \alpha^{n+1} &= q\alpha^n - (q-1)\alpha^0 \\ q\alpha^n - \alpha^{n+1} &= (q-1) \\ \alpha^n(q-\alpha) &= (q-1) \\ \alpha^n &= \frac{q-1}{q-\alpha} \end{aligned}$$

Berg also provides some other identities relating common sequences to operational calculus elements. Here are some elementary algebra to operational calculus relations:

- $\alpha^n = \frac{q-1}{q-\alpha}$
- $\binom{n}{k} = \frac{1}{(q-1)^k}$
- $n = \frac{1}{q-1}$
- $\binom{n}{c} k^{n-c} = \frac{q-1}{(q-k)^{c+1}}$
- $\frac{k^n-1}{k-1} = \frac{1}{q-k}$
- $\lfloor \frac{n}{k} \rfloor = \frac{1}{q^k-1}$

2.4. Connection to Generating Functions

Some of the more popular recurrence solvers such as Mathematica™are based on the properties of generating functions [6] to solve recurrences. In this manner sequences are represented as coefficients of power series. For example consider

$$a =< a_0, a_1, a_2, ... >$$

Then a can be represented by the sum

$$\sum_{n=0}^{\infty} a_n z^{-n}$$

Multiplication of two sequences when represented this way is the following:

$$\sum_{n=0}^{\infty} a_n z^{-n} * \sum_{n=0}^{\infty} b_n z^{-n} = \sum_{n=0}^{\infty} \sum_{i=0}^{n} a_i b_{n-i} z^{-n}$$

Thus the new sequence of the multiplication is the convolution of the two sequences, so the multiplication operator in the generating function world is a convolution whereas the multiplication in Berg's algebra is a convolution difference. This is because the power series representing Berg's sequences is

$$\sum_{n=0}^{\infty} a_n z^{-n}(1 - z^{-1})$$

where z corresponds to the operator q.

At this time it is unclear to the author if this means there is an isomorphism between the so called Z domain and the operational calculus algebra.

It does mean however that the operational calculus algebra often results in simpler terms to manage then the Z domain. For example in operational calculus a constant sequence c is represented by just the variable c, whereas in the Z domain the constant sequence c is represented by $\frac{c}{1-z^{-1}}$. Also n^2 is represented as $\frac{2}{(q-1)^2} + \frac{1}{q-1}$ whereas in the Z domain n^2 corresponds to $\frac{z^{-1}(1+z^{-1})}{(1-z^{-1})^3}$.

These slightly simpler terms might lead to a faster and simpler solver, since the algebra system will have to deal with less complex terms.

3. The Method

In [2, Chapter 2] shows how his operational calculus algebra can be used to solve recurrences. This is done through equation (3) in $\mathfrak{K}$ relating shifted sequences as well as using the common identities to translate operational calculus expressions to well known sequences. For example,

suppose we wished to solve $y_{n+1} = 2y_n + n$. We could perform the following steps:

$$y_{n+1} = 2y_n + n$$

$$\text{by (3) and } n = \frac{1}{q-1}$$

$$qy_n - (q-1)y_0 = 2y_n + \frac{1}{q-1}$$

$$y_n(q-2) = (q-1)y_0 + \frac{1}{q-1}$$

$$y_n = \frac{q-1}{q-2}y_0 + \frac{1}{(q-1)(q-2)}$$

$$\text{by partial fractions}$$

$$y_n = \frac{q-1}{q-2}y_0 + \frac{1}{q-2} - \frac{1}{q-1}$$

$$\text{using transformation identities}$$

$$y_n = 2^n y_0 + \frac{2^n - 1}{2-1} - n$$

$$y_n = 2^n y_0 + 2^n - n - 1$$

This method is not limited to recurrences of order 1. Suppose we would like to derive a closed form expression for the Fibonacci sequence. The Fibonacci sequence can be described by $F_{n+2} = F_{n+1} + F_n$ with $F_0 = F_1 = 1$. Thus after applying equation (3) we have

$$q^2F_n - (q-1)F_0 - (q-1)qF_1 = qF_n - (q-1)F_0 + F_n$$

$$F_n(q^2 - q - 1) = (q-1)q$$

$$F_n = \frac{q(q-1)}{q^2 - q - 1}$$

The roots of $q^2 - q - 1$ are $\phi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$.

$$\frac{q}{q^2 - q - 1} = \frac{q}{(q-\phi)(q-\psi)}$$

$$\text{by partial fraction decomposition}$$

$$= \frac{\phi}{\sqrt{5}(q-\phi)} - \frac{\psi}{\sqrt{5}(q-\psi)}$$

Therefore

$$F_n = (q-1)\left(\frac{\phi}{\sqrt{5}(q-\phi)} - \frac{\psi}{\sqrt{5}(q-\psi)}\right)$$

$$= \frac{(q-1)\phi}{\sqrt{5}(q-\phi)} - \frac{(q-1)\psi}{\sqrt{5}(q-\psi)}$$

$$= \frac{\phi}{\sqrt{5}}\phi^n - \frac{\psi}{\sqrt{5}}\psi^n \qquad \text{by } \alpha^n = \frac{q-1}{q-\alpha}$$

$$F_n = \frac{\phi^{n+1} - \psi^{n+1}}{\sqrt{5}}$$

The general procedure of solving recurrences via this algebra is as follows:

1) Apply equation (3) to all terms $\alpha_k y_{n+k}$ and apply identity transforms to all terms containing the independent variable.
2) Solve the equation for y_n
3) Perform algebraic manipulations such as partial fraction decomposition so the right side is a sum of terms with identity transforms.
4) Apply the identity transforms to get a result in elementary algebra

This method of solving recurrences is able to solve a recurrence of the form of equation (1), where $f(n)$ has an appropriate transform in the operational calculus domain.

3.1. Difference Between Multiplications

It should be noted here about the caution when performing the first step of the procedure. When operating in the elementary algebra the implicit multiplication of sequences is component-wise multiplication. Meaning

$$n * n =< 0, 1, 2, 3, \ldots > * < 0, 1, 2, 3, \ldots >$$
$$=< 0, 1, 4, 9, \ldots >$$

Whereas in the operational calculus algebra the implicit multiplication operator is the convolution difference described in equation (2). Thus

$$n * n =< 0, 1, 2, 3, \ldots > * < 0, 1, 2, 3, \ldots >$$
$$=< 0, 0, 1, 3, 6, 10\ldots >$$

It should then be noted that we subtly changed multiplication types in our first example. This is when we took the component-wise multiplication of $2y_n$ to the convolution difference multiplication yet keeping the term as $2y_n$. In this case, though, the transformation is justified. This is due to the multiplication described in (2) being equivalent to component-wise multiplication when one of the factors is a constant sequence. Here 2 is a constant sequence so we are able to change multiplications without mention.

4. Implementation Details

The section will describe some of the implementation details of an Operational Calculus Recurrence Solver (OCRS) written in OCaml based on the methods described above. Since the goal is to symbolically solve recurrences, OCRS borrows many techniques from computer algebra systems. Specifically many of the algorithms used in OCRS were created by Joel Cohen in [3] and [4]. What follows are some important functions implemented in OCRS that are required to solve recurrences via the method described in Section 3.

4.1. Automatic Simplification

Implemented in OCRS is a function to automatically simplify an algebraic expression. This function is based on the algorithm described in [4]. The main purpose of this automatic simplification function is to remove ambiguity between expressions. Thus the function attempts to transform expressions towards a canonical form. Specifically, the automatic simplification process will flatten product and sum

operators, order operands of sums and products according to an order function so as to remove commutativity ambiguity, and transform division and subtraction operators to the appropriate sums and products. Also included in the automatic simplify algorithm is logic to combine like terms and bases, as well as flatten exponents.

Below is a set of example simplifications performed by OCRS.

$$\begin{aligned}((x^{\frac{1}{2}})^{\frac{1}{2}})^8 &\to x^2\\ 2x^2 + x^2 * x &\to 3x^3\\ \frac{1}{a} * b * a &\to b\\ 2 * (x + y) &\to 2x + 2y\\ a + b - (a + b) &\to 0\\ 3 * ((2 * x) + y) &\to 6x + 3y\\ 3 * (2 * (x + y)) &\to 6x + 6y\end{aligned}$$

It should be noted at that the automatic simplify algorithm does not transform all equivalent expressions to the same form. For example variables are not always distributed through sums. Thus

$$a * (b + c) \neq ab + ac \text{ according to OCRS}$$

4.2. Binomial Transformation

Since there is no rule for transforming a polynomial in elementary algebra to an expression in the operational calculus algebra, some algebraic manipulations must be performed. Specifically in this case the only possible transform that could potentially be applied is $\binom{n}{k} = \frac{1}{(q-1)^k}$. Thus polynomials must be transformed to the appropriate binomials. This can be done via the following calculation:

$$n^{deg} = \sum_{k=0}^{deg}(\sum_{j=0}^{k}(-1)^{k-j} j^{deg}\binom{k}{j})\binom{n}{k}$$

For example when asked to perform the binomial transform on $n^4 + n^3$ OCRS returns the following:

$$n^3 + n^4 \to 24\binom{n}{4} + 42\binom{n}{3} + 20\binom{n}{2} + 2\binom{n}{1}$$

4.3. Operational Calculus to Elementary Algebra

The process for translating an input recurrence relation to an equation in $\mathfrak{R}$ is done by translating each term in the equation. Dependent variables are translated by (3). For all other terms OCRS tries to apply identity transforms. However, as noted in the previous section polynomials are not directly able to be transformed. Thus before OCRS translates an equation to the operational calculus domain a binomial transform will be applied where able.

4.4. Solving for Variables

Implemented in OCRS is a function that takes in as input an automatically simplified equation, a string representing the variable to solve for, and another string representing the iteration variable of that function. The general strategy of this function is to first get all terms with the desired variable on the left side of the equation and all other terms on the right side of the equation. Then the function checks to see if the left side of the equation is a product or a sum. If the expression is a sum we factor out the desired variable from each term. If the expression is a product we must divide both sides of the equation by the factors that don't contain the desired variable.

After each of these steps the function also calls the appropriate automatic simplify function, so as to cancel factors and terms. We also recursively call the solving function until we have an equation where the desired variable is the only expression on the left side of the equation, and the right side of the equation contains no references to that variable.

It is important to note that this function is not able to solve all possible equations that can be specified in my intermediate representation. However, it should be the case that every recurrence in the form of (1), defined above, should generate an operational calculus equation that can be solved by the function described here.

4.5. Algebraic Expand

As noted earlier automatic simplification won't expand $a(b+c)$. This is justified for many reasons, one for example is that factoring would not be possible. However sometimes it is necessary to distribute terms over a sum. That is the purpose of this function.

4.6. The Procedure

The overall procedure to solve a recurrence performs the following steps:

1) Automatically simplify the input equation
2) Translate the equation to the operational calculus domain
3) Solve the equation for the desired dependent variable
4) Algebraically expand the right side of the equation
5) Perform partial fraction decomposition on the right side of the equation
6) If all terms on the right side of the equation have an appropriate operational calculus translation, then translate the equation back to the elementary algebra domain. If not fail. The resultant expression may contain binomials and won't necessarily be simplified
7) Eliminate binomials by constructing the product of terms of the binomial. Then expand the expression and simplify.

See Appendix A for an example run of this procedure.

5. Preliminary Results

The following few recurrences were run on both OCRS as well as the computer algebra system Mathematica™. For these examples OCRS was run on an Ubuntu virtual machine while Mathematica™completed the examples natively in a Windows environment. The Mathematica™times are averaged over a thousand runs and the times only include the kernel time Mathematica™took to solve the recurrence.

	OCRS	Mathematica™
$y_{n+1} = y_n + 1$	0.000198	0.003130
$y_{n+1} = y_n + n^3 + n^4$	0.002235	0.013300
$y_{n+1} = 2y_n$	0.003040	0.004240
$y_{n+1} = 2y_n + n^2 + 3^n$	0.001831	0.407000

These preliminary results seem to show the efficacy of this method for solving recurrences. While OCRS took much less time to solve these few problems, Mathematica™is able to solve a much larger class of recurrences. However, Mathematica™contains much more functionality than just the ability to solve recurrences. Thus it seems reasonable to conclude that if OCRS was extended to be able to solve a larger class of recurrences, by using transformation techniques, such as in [6], OCRS would continue to outperform Mathematica™in terms of execution time.

6. Future Work

Even though the timing results are promising much more capability needs to be added to OCRS to be competitive with the solving capabilities of other recurrence solvers. However, the core of OCRS is fairly comparable to the core of the RSolve package in Mathematica™[6]. Thus many of the transformations used in [6] should be able to be adopted to OCRS.

One transformation of interest is finding an appropriate change of variable for the case of non-linear recurrences. For example there is no obvious way to handle the following recurrence:

$$y_n = y_{\frac{n}{2}} + 1$$

However, by substituting 2^k for n we have

$$y_{2^k} = y_{2^{k-1}} + 1$$

We can then let $z_k = y_{2^k}$, then we have to solve

$$z_{k+1} = z_k + 1$$

which has solution $z_k = k$. We have $y_{2^k} = k$ and $\log_2 n = k$, so $y_n = \log_2 n$. Similar substitutions can be made in other situations.

It was noted in Section 3 that the method of solving recurrences via the operational calculus has the capability to solve linear non-homogeneous recurrences of arbitrary order with constant coefficients. However, OCRS only has the capability of solving recurrences of order one. This restriction is due to OCRS not having a sufficiently powerful symbolic factoring algorithm. A recurrence of order k will correspond to a polynomial in $[Q][q]$ with degree k, and this polynomial must be factored into a product of monomials to generate terms that can be found in the operational calculus algebra identity list. This means that if a powerful enough factoring algorithm was implemented in OCRS, the system would have the ability to solve recurrences with arbitrary order.

Another avenue for extension is the ability to solve multivariate recurrences. There exists a matrix analog of the operational calculus algebra for higher dimensions. However, this analog is not fully understood at the moment by the author.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proceedings of the 15th International Symposium on Static Analysis*, SAS '08, pages 221–237, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] Lothar Berg. *Introduction To The Operational Calculus*. North Holland.

[3] Joel S. Cohen. *Computer Algebra and Symbolic Computation: Elementary Algorithms*. A K Peters/CRC Press.

[4] Joel S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. A K Peters/CRC Press.

[5] Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 57–64, Austin, TX, 2015. FMCAD Inc.

[6] Marko Petkovsek. *Finding Closed-form Solutions of Difference Equations by Symbolic Methods*. PhD thesis, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-33361.

Appendix A: Example Run of OCRS

Below will be an example output of OCRS. The output as presented in this paper will be typeset. However, OCRS does not format its output in the same way presented here.

Input: $y_{n+1} = 2 * y_n + n^2 + 3^n$

Simplified Expression: $y_{n+1} = 3^n + n^2 + 2y_n$

Operational Calculus:

$$-(q-1)y_0 + qy_n = \frac{2}{(q-1)^2} + \frac{1}{q-1} + \frac{q-1}{q-3} + 2y_n$$

Isolated Expression:

$$y_n = (\frac{1}{q-2})(\frac{2}{(q-1)^2} + \frac{1}{q-1} + \frac{q-1}{q-3} + 2y_n + (q-1)y_0)$$

Expanded Expression:

$$y_n = \frac{2}{(q-2)(q-1)^2} + \frac{1}{(q-1)(q-2)} + \frac{q-1}{(q-3)(q-1)} + \frac{q-1}{q-2}y_0$$

After Partial Fraction:

$$y_n = \frac{2}{q-3} + \frac{2}{q-2} - \frac{2}{(q-1)^2} - \frac{3}{q-1} + \frac{q-1}{q-2}y_0$$

Initial Result:

$$y_n = 2\frac{3^n - 1}{3-1} + 2\frac{2^n - 1}{2-1} - 2\binom{n}{2} - 3\binom{n}{1} + 2^n y_0$$

Final Result: $y_n = -3 + 2 * 2^n + 3^n + 2^n y_0 - 2n - n^2$