An Automatic Induction Proof for Program Termination Analysis

Kathlyn Doss, Stefan Andrei, Kami Makki

No. 3, August 2010

Computer Science Department Technical Reports - Lamar University

Computer Science Department, Lamar University,
211 Red Bird Lane, Box 10056, Beaumont, TX 77710, U.S.A.
URL: http://cs.lamar.edu/tech_reports
Email: tech_reports@cs.lamar.edu
An Automatic Induction Proof for Program Termination Analysis

Kathlyn Doss
Lamar University
Beaumont, TX, USA
kdoss@my.lamar.edu

Stefan Andrei
Lamar University
Beaumont, TX, USA
Stefan.Andrei@lamar.edu

Kami Makki
Lamar University
Beaumont, TX, USA
Kami.Makki@lamar.edu

Abstract

The termination problem is in general undecidable, however, termination can be proved for specific classes of programs. This work describes an automatic method that tests for termination of modulo-case functions by conducting a mathematical induction proof. The system takes a modulo-case function and builds an execution trace tree from its inverse. Based on the execution trace tree, a linear polynomial is formed in order to capture the level of the executed trace tree for which the termination problem holds. Using the cases of the original function, an inductive proof is performed and compared to the coefficient of the polynomial. If the completed proof is found to be within the polynomial’s coefficient then termination has been proved, else a larger polynomial needs to be found and tested. If a function successfully terminates, its runtime is generated based on the same polynomial.

Keywords: termination analysis, induction proof, modulo-case function

1 Introduction

One key topic in computer science is termination. When a program has finished computation of a given input it has terminated. Termination for any arbitrary program can be difficult to determine. In fact, in 1936, Alan Turing showed that the termination problem, also known as the halting problem, was undecidable [7]. The truth of any non-trivial statements about arbitrary functions is undecidable, as stated in Rice's theorem [5]. Although there does not exist a general solution applicable for all programs, termination can be solved on an individual basis.

Termination is an important part in determining the correctness of a program. Total correctness can be defined as partial correctness plus termination. According to [6], termination analysis is a challenging research topic in both theory (mathematical logic, proof theory) and practice (software development, formal methods).

The motivating example's termination cannot be automatically solved using the size-change termination technique of [4] because the recursive calls may not retain the same parameters throughout. Meanwhile, the affine-based SCT of Anderson and Khoo [1] does not directly show that this function terminates. [2] Also, systems like the Omega Calculator [3] were not able to assess the termination of all of the example functions that this system was able to confirm.

This paper details the automated implementation of Andrei's method for determining termination [2] and details the induction proof used to prove termination of modular based functions. The system takes the numerical normal form of a modular function, f(), as input and builds an execution trace tree based on the function's inverse. Using initial sets, created from the arguments of f(), and the levels they were generated by, based on the execution trace tree, a linear polynomial is created. An induction proof is conducted on the function, f(), which is then compared to the polynomial found. If the steps of the induction proof are within the coefficient of the linear polynomial then the polynomial holds and termination has been proved. If the steps exceed the coefficient then a polynomial of the next degree needs to be created and tested. The current system deals only with modular based functions and linear polynomials.

The technique presented in [2] is systematic, in that it shows termination of modulo-case functions with the designer’s assistance. The tool of [2] does not proved an automatic proof of function termination. This current work extends the systematic algorithm of [2] by transforming the previous method into a completely automatic algorithm that is able to provide the proof by mathematical induction for the modulo-case functions that terminate.
Structure of the paper: Section 2 provides notations, definitions, examples, algorithms, data structures, correctness, and complexity. Section 3 includes the analysis, design, implementation details, and testing. Section 4 makes comparisons with existing related techniques and provides screen shots of the working tool. Section 5 includes the conclusion and future work.

2 Preliminaries and Method

Concentrating on the non-trivial class of modulo functions, the numerical normal form of these functions is used to represent recursive calls as a sequence of function compositions of \( f() \) [2]. The numerical normal form for functions having one parameter is \( F : \mathbb{N} \rightarrow \mathbb{N} \), given by:

\[
F(x) = \begin{cases} 
  n'_0 & x = n_0, \\
  \ldots & , \\
  n'_{m-1} & x = n_{m-1}, \\
  f_0(x) & \beta_1(x), \\
  \ldots & , \\
  f_{p-1}(x) & \beta_{p-1}(x).
\end{cases}
\]

We can see the form of \( F(x) \) in [2], which states that since \( n'_0, \ldots, n'_{m-1} \) are constants, the guards \( x = n_0, \ldots, x = n_{m-1} \) may be also called terminating conditions. In addition, the functions \( f_0(x), \ldots, f_{p-1}(x) \) may contain \( x \), then \( \beta_1(x), \ldots, \beta_{p-1}(x) \) are called non-terminating conditions. Note that \( \beta_1(x), \ldots, \beta_{p-1}(x) \) are boolean expressions that may contain binary operators including modulo. The functions \( f_0(x), \ldots, f_{p-1}(x) \) are required to be invertible functions for the termination method. The function \( f : A \rightarrow B \) is an invertible function if there exists \( g : B \rightarrow A \) such that \( f \circ g = 1_B \) and \( g \circ f = 1_A \), where \( 1_A \) and \( 1_B \) represent the identity functions over \( A \) and \( B \), and \( \circ \) represents the function’s composition.

From this point on, the functions will be defined in numerical normal form, for example:

\[ f_i(n) = \begin{cases} 
  0 & n = 0 \text{ or } n = 1 \text{ or } n = 2 \\
  n/3 & n = 0 \text{ (mod 3) and } n > 2 \\
  2n-2 & n = 1 \text{ (mod 3) and } n > 2 \\
  n+1 & n = 2 \text{ (mod 3) and } n > 2
\end{cases} \]

In \( f_i(n) \), \( n = 0, n = 1, \) and \( n = 2 \) are all terminating cases. Non-terminating cases, or conditions, contain modulo operators: \( n \equiv 0 \text{ (mod 3)} \) and \( n > 2, n \equiv 1 \text{ (mod 3)} \) and \( n > 2, \) and \( n \equiv 2 \text{ (mod 3)} \) and \( n > 2 \). Affine expressions on variable \( n \) include \( n/3, 2n-2, \) and \( n+1 \). Note that \( n \) and \( m \) are integers, and \( n \equiv m \text{ (mod p)} \) is shorthand for there exists an integer \( r \) such that \( n - m = p \cdot r \).

Formally, the execution trace tree of function \( F() \) is denoted \( \text{ETT}(F) \). Such a tree is a pair \((V, E)\), where \( V \) is the set of nodes or vertices, and \( E \) is the set of arcs known as edges. Note that the root remains unlabeled. The direct descendents of the root are the terminating cases of the function, and all other nodes will be labeled with naturals. The arcs represent a solution from \( F(x) = y \) as going from node \( y \) to node \( x \).

In [2], we learn to associate any \( y \) and \( k \), which are strictly natural, a finite \( p \)-ary tree with root \( y \), having \( k \) levels, denoted \( \text{ETT}_k(y) = (V_k, E_k) \). Each node \( v \) belonging to \( V_k \), except the root, is labeled with a natural number, denoted by \( \text{label}(v) \). The node \( v \) may have \( 1, 2, \ldots, p \) descendents depending on its label \( x \). That is for all \( i \) belonging to \( \{0, \ldots, p-1\} \) whenever \( \beta_i(f_i^{-1}(x)) \) holds, then \( v \) has the descendant labeled by \( f_i^{-1}(x) \). If all the paths from the root are finite, then the termination problem can be easily solved. Namely, for the inputs that are labels of \( \text{ETT}(F) \), the termination problem holds, whereas for the rest of them, the program does not terminate [2].
For example $f_2()$ is terminating for inputs 0, 1, and 2, and is non-terminating for all other inputs. When a tree's nodes are generated up to a specific level, then termination is solved for the generated labels. Therefore, the difficulty of termination is when the tree has at least one infinite path. The modulo-case functions used with the automatic system discussed here have the pattern:

$$f_2(n) = \begin{cases} 
0 & n=0 \text{ or } n=1 \text{ or } n=2, \\
2n+1 & 2>n \text{ and } n<10, \\
n-1 & 9<n
\end{cases}$$

![Figure 1. Function $f_2(n)$ and ETT($f_2$)](image)

The ETT(F) is built using the inverse of the affine functions from the F(x) function. Based on the modulo-case function, any arbitrary node $v$ may have up to $p$ descendants each. “That is, for all $i$ belonging to $\{0, \ldots, p-1\}$ whenever $(x-b_i)/a_i \equiv i \pmod{p}$, then $v$ has a descendant labeled by $(x-b_1)/a_1$ “ \[2\]. For example $f_{i}^{-1} = \{(m, 3m) \mid m \geq 1\} \cup \{(m, (m+2)/2) \mid m \geq 3 \text{ and } m \equiv 0 \pmod{6}\} \cup \{(m, m-1) \mid m \geq 4 \text{ and } m \equiv 0 \pmod{3}\}$. The ETT($f_i$) is initially constructed by labeling the root's immediate descendants with the termination cases 0, 1, and 2. The following levels' nodes are created based on which conditions their immediate ancestors met. Figure 2 shows the first five levels of ETT($f_i$).

![Fig. 2. The first four levels of ETT($f_1$)](image)

The method provided in [2] is utilized for preparation of the induction proof. Step 1 is to determine $s$ and $\varphi(k)$:

- $s = \max_{i \in \{0, \ldots, p-1\}} \{[1/a_i], 1\}$
- $\varphi(k) = s^k$ if $s>1$ or $k$ if $s=1$, where $[x]$ represents the integer ceiling of $x$

The number $s$ is used to define $\varphi(k)$, the finite integer set, $\{0, 1, \ldots, \varphi(k)\}$, that appears in the proof. Step 2 is to inverse the function, $f()$, and build the ETT($f$). The ETT($f$) will be generated up to the smallest level where every label from the the first set $\{0, 1, \ldots, \varphi(k)\}$ and then for the following set $\{0, 1, \ldots, \varphi(k+1)\}$. A polynomial, P, is then formed such that $P(k)$ is the smallest level needed for a given set $\{0, 1, \ldots, \varphi(k)\}$. Our implementation deals only with linear polynomials at this time, but the below algorithm works for general polynomials. Algorithm A details the steps from the systematic method:
Algorithm A:

1. Find \( f^1, \text{ETT}(f), m, \) and \( \phi() \)
2. \( d = 1; \text{polynomialFound} = \text{false} \)
3. \( \text{while (true) \{} \)
   4. \( \text{while (polynomialFound == false) \{} \)
      5. \( \text{Find the polynomial } P \text{ of degree } d \text{ such that } P(k) = s_k \text{, for all } k \text{ belonging to } \{1, ..., d+1\}, \text{ where } s_k \text{ represents the smallest level such that } \{0, ..., \phi(k)\} \text{ labels ETT}(f) \)
      6. \( \text{if } (s_{d+2} \leq P(d+2)) \text{ polynomialFound = true; } \)
      7. \( \text{else } d++: \)
      8. \( \text{if } (d > d_{\text{max}}) \text{ return } ('\text{don't know'; 0); } \)
   9. \( \text{result = CheckInductionStep(); } \)
10. \( \text{if (result == true) } \)
    11. \( \text{Let } S \text{ be the set of naturals for which the induction step holds} \)
    12. \( \text{if } (S == N) \text{ return } ('\text{yes'}; \phi^{-1}(P(k))); \)
    13. \( \text{else return } ('\text{yes'} \text{ for } n \text{ belonging to } S; \phi^{-1}(P(k)) \text{ and } ('\text{no'} \text{ for } n \text{ not belonging to } S; \text{infinity});) \)
14. \( \text{else polynomialFound = false; } d++; \)
\( \} \)

**Theorem 1.** Let \( f \) be a modulo-case function, and \( d_{\text{max}} \) a positive integer as the maximum degree for the polynomial. Algorithm A will provide:

- If \( f \) is terminating, then return ‘yes’ and a domain constraint as well as an estimation of running time;
- If \( f \) is not terminating, then return ‘no’ and a domain constraint;
- If the algorithm cannot reach a conclusion by finding a proper polynomial of degree up to \( d_{\text{max}}, \) then return ‘don’t know’.

The CheckInductionStep() is the most challenging step from Algorithm A. In fact, there is no algorithm for solving the termination problem in general [5]. Dealing with symbolic proofs represents an important area in the symbolic computation community. To the best of our knowledge, there is no fully automatic algorithm able to generate proofs by mathematical induction for general problems. Algorithm B details the main steps of applyDefinition_R(), the induction proof.
3 Implementation

Not all of the steps from Algorithm A were implemented in our system. Algorithm A includes steps to determine the runtime of modular functions along with the termination analysis. Our system deals with the termination portion, and runtime for only linear polynomials. The CheckInductionStep() is implemented through a class of its own, the Term class. The structures used to store the ETT include two array lists, one for the numbers generated and one for the corresponding level of each generated number. The polynomial also has a class of its own.

The Term class makes use of a canonical expression for the proof. The expression is in the form \(an^{(k+dk)} + br + c\). This expression is closed under addition of a constant, multiplication of a scalar, and substitution. This means that by performing any of the mentioned operations to a general term (from our Term class), we obtain a term of the same form. This is important because when the proof is being performed the expression will keep the same form no matter what degree the expression grows to be. This allows the applyDefinition_R() to be flexible without explicitly needing to have change the way the expression is stored.

Function f1()’s ETT() has been built, see Figure 2. The number s was determined to be 3, and \(\varphi(k) = s^k\), since \(s>1\). When k is 1 the set to be found is \(\{0, \ldots, 3\}\), and when k is 2 the set is \(\{0, \ldots, 9\}\). These sets are generated by ETT() levels 1 and 4, respectively. From this information we can find a linear polynomial that fits \(P(1) = 1\) and \(P(2) = 4\), which is \(P(k) = 3k-2\). We initially test the polynomial with \(P(3)\), which is the set \(\{0, \ldots, 27\}\), and see that by level 7, \(P(3) = 3*3^2 - 2 = 7\), all of the labels from the new set have been generated. The coefficient of our polynomial is used to determine if termination can be proved through induction. The induction proof should not exceed the coefficient in the number of times that the function must be reapplied to any case of...
the function. Through our system, the termination problem holds. Details about termination of \( f_1() \) will be given in Section 4 (Experimental Results).

Let us consider a modulo-case function with only two cases. Let \( f_1 : \mathbb{N} \rightarrow \mathbb{N} \) be such that:

\[
\begin{align*}
f_1(n) = \begin{cases} 
0 & n = 0 \text{ or } n = 1, \\
\frac{n}{2} & n \equiv 0 \pmod{2} \text{ and } n > 1, \\
n + 1 & n \equiv 1 \pmod{2} \text{ and } n > 1.
\end{cases}
\end{align*}
\]

The inverse of \( f_1() \) is \( f_1^{-1}() = \{ (m, 2m) \mid \text{for all } m \geq 1 \} \cup \{ (m, m-1) \mid m \text{ is even} \} \) which creates the ETTR() in Figure 3. In this example \( s = 2 \), and \( \phi(k) = s^k \), since \( s > 1 \). Therefore, the first integer set to find is \( \{ 0, \ldots, 2 \} \), which has been generated by level 1 of ETTR(f1), or simply P(1) = 1. The second set, \( \{ 0, \ldots, 4 \} \), is generated by level 3 of ETTR(f1), because while label 4 was generated on level 2, label 3 was not generated until level 3, or P(2) = 3. The polynomial constructed to fit the sets and levels was P(k) = 2k - 1. We then test the polynomial on P(k+1), P(3) = 5, which is correct by the ETTR(f1). With our polynomial discovered, we now know that the induction proof should not have more than two applications of the function f1() to any of the modular cases. When run through our system, the proof needed a maximum of two applications for the second case only. Therefore, the termination problem holds for all natural numbers.

We consider another modulo-case example with two cases. Let \( f_4 : \mathbb{N} \rightarrow \mathbb{N} \) be such that:

\[
\begin{align*}
f_4(n) = \begin{cases} 
0 & n = 0 \text{ or } n = 1, \\
\frac{n}{2} & n \equiv 0 \pmod{2} \text{ and } 1 < n, \\
2n-2 & n \equiv 1 \pmod{2} \text{ and } 1 < n.
\end{cases}
\end{align*}
\]

\( \phi(k) = s^k \), since \( s > 1 \). Set \( \{ 0, \ldots, 2 \} \) is generated by ETTR(f4) level 1, and set \( \{ 0, \ldots, 4 \} \) by level 3, when \( k = 2 \) respectively. This gives us the linear polynomial P(k) = 3k - 3. The number of times that the induction proof can apply f4() to any of the cases is equal to the coefficient 3. Termination does hold for this modular function for all natural inputs.

Let us consider another successful example for which our tool was able to provide a complete mathematical proof for termination. Let \( f_5 : \mathbb{N} \rightarrow \mathbb{N} \), a modulo-case function with three cases:

\[
\begin{align*}
f_5(n) = \begin{cases} 
0 & n = 0 \text{ or } n = 1, \\
n+7 & n \equiv 0 \pmod{3} \text{ and } n > 2, \\
n-2 & n \equiv 1 \pmod{3} \text{ and } n > 2, \\
n-4 & n \equiv 2 \pmod{3} \text{ and } n > 2.
\end{cases}
\end{align*}
\]

Obviously, \( f_5^{-1}() = \{ (m, m-7) \mid m \equiv 1 \pmod{3} \text{ and } m > 9 \} \cup \{ (m, (m+2)/2) \mid m \equiv 0 \pmod{4} \text{ and } m > 1 \} \cup \{ (m, m+4) \mid m \equiv 1 \pmod{3} \text{ and } m > 0 \}. \) For this example \( s = 1 \), and \( \phi(k) = k \), since \( s = 1 \). The set for \( k = 1 \) is \( \{ 0, 1 \} \), and when \( k = 2 \) the set is \( \{ 0, 1, 2 \} \). However, after building the ETTR(f5), we see that the two sets are generated by the termination cases. So, the system automatically looks for the next set, \( \{ 0, 1, 2, 3 \} \), which isn't fully generated until level four. Using the P(2) = 0 and P(3) = 4, because they are adjacent sets, the polynomial created is P(k) = 4k - 8. The induction proof states that termination holds true for f5 because the number of times the function needed to be reapplied did not exceed four, the polynomial's coefficient.

Not all modulo-case functions have proofs generated by our implementation. Here is an unsuccessful example. Let \( f_6 : \mathbb{N} \rightarrow \mathbb{N} \) be such that:

After presenting four successful examples, with \( s \) varying from 1 to 3, here is an unsuccessful example. \( F_6^{-1}() = \{ (m, 3m) \mid m \geq 1 \} \cup \{ (m, (m+1)/2) \mid m \equiv 1 \pmod{6} \text{ and } m \geq 4 \} \cup \{ (m, m-1) \mid m \equiv 0 \pmod{3} \text{ and } m \geq 4 \}. \) S = 3, and \( \phi(k) = s^k \), since \( s > 1 \). Set \( \{ 0, \ldots, 9 \} \) is generated by ETTR(f6) level 2 when \( k = 1 \). When \( k = 2 \), the set we are looking for is \( \{ 0, \ldots, 9 \} \). However, the problems is that labels 4 and 7 from the set never get generated by the ETTR(f6). Because those numbers are not created termination does not hold for all natural inputs into f6().
4 Experimental Results

Related systems included the Omega Calculator [3]. We looked at the Omega Calculator in comparison with the CheckInductionStep() from the method proposed in [2]. The CheckInductionStep() was not defined, and we felt that if the Omega Calculator was able to correctly identify termination in the examples checked with induction hypothesis we would attempt to incorporate it into our system. However, the Omega Calculator was inconclusive with example f3, as seen in Figure 4. The input of f3 into the calculator took the form of:

\[
U := \{(n,k) \rightarrow [n',k'] : (n \geq 2 \text{ and } \exists (e: n=2e \text{ and } 0 \leq e \text{ and } 2n' \leq n \leq 2n' + 1 \text{ and } k' + 1 = k) \text{ or } \\
\exists (d: n=1+2d \text{ and } 1 \leq d \text{ and } n'=n+1 \text{ and } k' + 1 = k)\}\}; U+.
\]

With our system we were able to completely analyze the termination of all six examples presented. A results table is presented in Figure 5. Four of the six were running examples, for all of which the termination problem held, see Figure 6 for the induction proof on f1. Another one of the examples was a running example, though it only terminated for the cases 0, 1, and 2, and was non-terminating for all other natural number inputs, Figure 1. The last example was inconclusive because it terminated for all but two inputs from the set \{0, \ldots, 9\}, Figure 7.

<table>
<thead>
<tr>
<th>(f_i)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1)</td>
<td>“Yes” ([n \geq 0])</td>
</tr>
<tr>
<td>(f_2)</td>
<td>“Yes” ([n \geq 0 \text{ and } n \leq 2])</td>
</tr>
<tr>
<td></td>
<td>“No” ([n \geq 3])</td>
</tr>
<tr>
<td>(f_3)</td>
<td>“Yes” ([n \geq 0])</td>
</tr>
<tr>
<td>(f_4)</td>
<td>“Yes” ([n \geq 0])</td>
</tr>
<tr>
<td>(f_5)</td>
<td>“Yes” ([n \geq 0])</td>
</tr>
<tr>
<td>(f_6)</td>
<td>“Yes” ([n \geq 0 \text{ and } n \leq 4])</td>
</tr>
<tr>
<td></td>
<td>“Don’t know” [otherwise]</td>
</tr>
</tbody>
</table>

Figure 5. Results Table
We list below the proof of termination provided by the tool for function $f_1()$.

**PROOF:**

**INDUCTIVE BASE:** We check the initial values $n = 0$, $n = 1$, $n = 2$.
- The set of integers $\{0, \ldots, 3.0\}$ are generated by level $P(1) = 1$
- The set of integers $\{0, \ldots, 9.0\}$ are generated by level $P(2) = 4$

**INDUCTIVE STEP:** We assume that the set of integers $\{0, 1, \ldots, 3^k\}$ is generated by level $P(k) = 3.0k^{-1} + 2.0$.
- We need to show that the set of integers $\{0, 1, \ldots, 3^{(k+1)}\}$ is generated by level $P(k+1) = 3.0k^{-1} + 1.0$.

We consider an arbitrary number from the set of integers $\{3^k + 1, \ldots, 3^{(k+1)}\}$, that is, $3^k + r$, where $r \in \{1, \ldots, 3^k\}$.

We distinguish 3 cases:

**Case 1:** $r \equiv 0 \pmod{3}$.
- We will apply case 0.33$n - 0.0$, creating the current term $0.33*n^{-}(k+0) + 1.0r + 0.0$.
- We test and conclude that $f(2^k + r)$ is smaller than $0.33*n^{-}(k+0) + 0.0r + 0.0$.
- According to inductive hypothesis, $3^k + r$ is obtained before level $3.0k^{-1} + 2.0$

**Case 2:** $r \equiv 1 \pmod{3}$.
- We will apply case 2.00$n - 2.0$, creating the current term $2.00*n^{-}(k+0) + 6.0r + 0.0$.
- We will need to reapply the case 0.33$n + 0.0$
- We will apply case 0.33$n + 0.0$, creating the current term $0.67*n^{-}(k+0) + 2.0r + 0.0$.
- We test and conclude that $f(2^k + r)$ is smaller than $0.67*n^{-}(k+0) + 0.0r + 0.0$.
- According to inductive hypothesis, $3^k + r$ is obtained before level $3.0k^{-1} + 2.0$

**Case 3:** $r \equiv 2 \pmod{3}$.
- We will apply case 1.00$n + 1.0$, creating the current term $1.00*n^{-}(k+0) + 3.0r + 3.0$.
- We will need to reapply the case 0.33$n + 0.0$
- We will apply case 0.33$n + 0.0$, creating the current term $0.33*n^{-}(k+0) + 1.0r + 1.0$.
- We test and conclude that $f(2^k + r)$ is smaller than $0.33*n^{-}(k+0) + 0.0r + 1.0$.
- According to inductive hypothesis, $3^k + r$ is obtained before level $3.0k^{-1} + 2.0$

**Figure 6. Induction Proof of $f_1()$**
We list below the attempt for proof of termination provided by the tool for function \( f_6() \).

**PROOF :**

**INDUCTIVE BASE :** We check the initial values \( n = 0, n = 1, n = 2 \).

The set of integers \( \{0, \ldots, 3.0\} \) are generated by level \( P(1) = 1 \).

**INDUCTIVE STEP :** We assume that the set of integers \( \{0, 1, \ldots, 3^{-k}\} \) is generated by level \( P(k) = -2.0k^{-1} + 3.0 \).

We need to show that the set of integers \( \{0, 1, \ldots, 3^{-k+1}\} \) is generated by level \( P(k+1) = -2.0k^{-1} + 1.0 \).

We consider an arbitrary number from the set of integers \( \{3^{-k+1}, \ldots, 3^{-k+1}\} \), that is, \( 3^{-k} + r \), where \( r \in \{1, \ldots, 3^{-k}\} \).

We distinguish 3 cases:

**Case 1 :** \( r \equiv 0 \pmod{3} \).

We will apply case \( 0.33n + 0.0 \), creating the current term \( 0.33n^{-k+1} + 1.0r + 0.0 \). We test and conclude that \( f(2^{-k} + r) \) is smaller than \( 0.33n^{-k+1} + 0.0r + 0.0 \).

According to inductive hypothesis, \( 3^{-k} + r \) is obtained before level \( -2.0k^{-1} + 3.0 \).

**Case 2 :** \( r \equiv 1 \pmod{3} \).

We will apply case \( 2.00n + -1.0 \), creating the current term \( 2.00n^{-k+1} + 6.0r + 1.0 \). We will need to reapply the case \( 2.00n + -1.0 \).

**Polynomial has to be changed.**

**Case 3 :** \( r \equiv 2 \pmod{3} \).

We will apply case \( 0.33n + 0.0 \), creating the current term \( 0.33n^{-k+1} + 3.0r + 3.0 \). We will need to reapply the case \( 0.33n + 0.0 \).

**Polynomial has to be changed.**

**Figure 7. Induction Proof on \( f_6() \)**

5 Conclusion and Future Work

The automatic system presented here proves termination of a non-trivial class of programs, namely the modulo-case functions. The algorithm used to perform the induction proof uses symbolic expressions to evaluate the function and a counter to track the steps of the induction. The steps are compared to the polynomial found from the execution trace tree, determining termination. The results from the system include a written proof indicating termination of the function, or a partial proof stating the polynomial needs to be considered at a higher degree.

Future work with this project could include extending the class of functions that this method works for, and extending the polynomials that can be tested by the induction proof.

6 References


