A theory of function-induced-orders to study recursion termination

Abhinav Aggarwal, Padam Kumar
Indian Institute of Technology Roorkee

abhinav6891@gmail.com, padamfec@iitr.ernet.in

Abstract
Termination property of functions is an important issue in computability theory. In this paper, we show that repeated iterations of a function can induce an order amongst the elements of its domain set. Hasse diagram of the poset, thus obtained, is shown to look like a forest of trees, with a possible base set and a generator set (defined in the paper). ‘Isomorphic forests’ may arise for different functions and equivalences classes are, thus, formed. Based on this analysis, a study of the class of deterministically terminating functions is presented, in which the existence of a Self-Ranking Program, which can prove its own termination, and a Universal Terminating Function, from which every other terminating function can be derived, is conjectured.

Keywords
Function, recursion, termination, order, induced topology, poset, chain, isomorphism

I. Introduction

A large number of computer programs contain constructs like recursive function calls and loops, which pose difficulties in determining if the concerned program would eventually halt for a given set of inputs. It is usually a daunting task to speak with surety whether a given loop in our program would always modify the program variables in a way so that they stop satisfying the loop condition after a finite number of iterations, or if a given recursive call to a function would end up returning a predetermined value, thus ceasing any more iterations. Both scenarios are special instances of the generalized Halting Problem as given in [13], and are, therefore, unsolvable in general. A proof of this unsolvability has been well documented in [7]. However, well-founded partially ordered sets (poset), as defined in [15], have often been used to come up with possible solutions for the restricted instances of the same. If a strictly decreasing function, which maps the program variables into a well-founded poset, can be devised for a given program, then we can be sure that the value of this function will never decrease indefinitely upon changes to these variables as the program continues its execution. Alan Turing referred to such functions as Ranking functions [9, 13]. At the point when no further decrement is possible, we
say that the program has terminated. This is because the program will no longer be able to modify its state variables while avoiding any decrement in the output of the corresponding ranking function.

Though this technique sounds simple, it is often difficult to find appropriate ranking functions for complex programs. As an example, consider the function \( f : \{1, 2, 3, \ldots\} \rightarrow \{1, 2, 3, \ldots\} \), defined as:

\[
\begin{align*}
  f(x) &= \begin{cases} 
  0 & x = 1 \\
  f\left(\frac{x}{2}\right) & x \text{ is even, } x > 1 \\
  f\left(\frac{3x+1}{2}\right) & x \text{ is odd, } x > 1
  \end{cases}
\end{align*}
\]

The argument of \( f(x) \), for its next iteration, decreases if \( x \) is even and increases otherwise. Thus, even when we know that the set of natural numbers is well-founded under the natural ordering of \(<\) (less than), we cannot be sure if \( f(x) \) would eventually terminate since the function governing the change of argument for \( f(x) \) is not strictly decreasing. In fact, the recurrence in \( f \) is motivated by the famous Collatz conjecture, a good discussion of which has been done in [14]. Solving the termination problem in this case would validate the statement of this conjecture. Unfortunately, no solution for the same is known so far. An even worse situation is when we realize that there is no way of telling if a ranking function exists for a given problem, for if there was a way to find this out, it would be similar to finding a program which determines if a given program halts or not, thus solving the Halting problem itself.

The approach taken in this paper can provide some insight for analyzing functions of the kind described above. We talk not of what a function computes but on what values is the computation performed. We then study some structural properties of functions in relation to the set of inputs they may receive. These properties are specific to the nature of iterated calls of functions, mainly for the purposes of studying recursion.

### II. Function Orders and the Induced Function Topology

Consider the set \( \mathbb{N} \) of natural numbers \( \{0, 1, 2, 3, \ldots\} \). A natural ordering, denoted by \(<\) (less than), forms a total order, or a chain, in \( \mathbb{N} \). Moreover, many such orders can be defined using various relations on pairs of natural numbers [2]. For example, we can define a relation \(<\) between \( x \) and \( y \) (both belonging to \( \mathbb{N} \setminus \{0, 1\} \)) by:
$x < y$ iff $x$ divides $y$.

It is easy to see how this relation induces a partial order on $\mathbb{N} \setminus \{0, 1\}$. The set of minimal elements for this poset is precisely the set of all prime numbers, $\{2, 3, 5, 7, \ldots\}$. If the domain is changed to $\mathbb{N} \setminus \{0\}$, this set changes to $\{1\}$ because 1 divides every other number in $\mathbb{N} \setminus \{0, 1\}$, including itself. In both cases, the elements in the domain of the concerned relation seem to have been *rearranged* from their natural order to the new order; a fact, which can be verified by comparing the two Hasse diagrams, thus obtained (a good discussion of Hasse diagrams is done in [2]). The purpose of this section is to study this change in *arrangement* of natural numbers as *induced* by different functions defined on them. The properties of various structures thus obtained are of central importance to the underlying study.

Consider a set $A \subseteq \mathbb{N}$, finite or infinite, and a function $f : A \to A$. Here, we speak of functions from the set $A$ to itself because we are interested in studying properties of calling these functions recursively, i.e. we intend to repeatedly apply these functions to their outputs on a given input until some stopping criteria is met (More precisely, we assume the classical definition of recursion, as given in [5]). Hence, we must make sure that the range is at least a subset of the domain, the easiest case being that the range and the domain are the same. Now, let $(f)_A$ represent the collection of all such functions for a given set $A$. For any element $x \in A$, assume a computational procedure, which evaluates $f$ using $x$ as an argument. The result of this computation is $f(x)$, which is passed directly, or after some modifications, to $f$ again. Let this new argument to $f$ be denoted by $g(f(x))$, where $g(x)$ is the *post-computational modification* done to the output of $f$ before recursion. If we assume that this modification is done for every computation of $f$, we can view $g(f(x))$ and $f(x)$ as one function, purely for the purpose of analysis here, and refer to this composition as just $f(x)$, without any loss of generalization. The advantage of this abstraction is to visualize a recursive call as consisting only an input $x$, a function $f$, and the output $f(x)$, which will be fed as it is to the function $f$ again, until some stopping criteria is met. This criterion is assumed to be contained in the definition of $f$ itself. The question of whether $f$ always meets this stopping criteria for all inputs still remains open, and we do not assume any prior knowledge in this regard.

With such an understanding of recursion, we can call the set $\{x, f(x), f(f(x)), \ldots\}$ an *orbit* of $x$ under the map $f$ in accordance with the definition of this term as given in [1]. Since we know that all elements in the sequence lie in the set $A$ itself, and that no two elements of this sequence can be interchanged, in general, to give another valid sequence under the same map, we can associate an order to these elements of $A$, as induced by the repeated application of $f$ on $x$. This order,
denoted by $\prec_f$, is said to be induced by $f$ on the elements of $A$ and we can give the following definition to this relation.

**Definition 2.1 (Function-order)** Given a set $A \subseteq \mathbb{N}$ and two elements $x, y \in A$, along with a function $f : A \rightarrow A$, define a relation $\prec_f$ between $x$ and $y$, denoted by $x \prec_f y$, and thereby read as $x$ is functionally less than $y$, if there exists a number $n \in \mathbb{N}$ such that $y = f^n(x)$. Call this relation a function-order induced on $A$ by $f$.

In other words, the above definition asserts $x \prec_f y$ if $y$ belongs to the orbit of $x$ under $f$. Thus, in general, we have $x \prec_f f(x) \prec_f f^2(x) \prec_f \ldots$ for every $x$ in $A$. It is easy to see that this relation is reflexive ($n = 0$) and transitive, i.e.

$$\forall x, y, z \in A : (x \prec_f y) \land (y \prec_f z) \rightarrow (x \prec_f z)$$

**Definition 2.2 (Function-equality)** Given a set $A \subseteq \mathbb{N}$ and two elements $x, y \in A$, along with a function $f : A \rightarrow A$, define a relation $=_f$ between $x$ and $y$, denoted by $x =_f y$, and thereby read as $x$ is functionally equal to $y$, if either $x = y$ (i.e. they are not distinct) or $x \prec_f y$ and $y \prec_f x$ hold simultaneously. Call this relation a function-equality induced on $A$ by $f$.

Clearly, $=_f$ forms an equivalence relation between the elements of $A$. Function-equality is important for functions whose iterations are periodic, i.e. whose orbits, for a given $x$, show periodic behavior. The reader can gain more insight into the nature of this periodicity, along with some nice examples, through [1]. We denote by $X$ the equivalence class of all elements of $A$, which are functionally-equal to $x$, i.e.

$$X = \{y : y \in A, x =_f y\}$$

Such equivalence classes may contain singleton elements when the corresponding orbits do not show any periodic behavior. With this picture in mind, let $A_f$ denote the collection of all equivalence classes that arise out of elements of $A$ due to the order induced by the function $f$. Let this collection be called the reduced domain of $f$. We can extend the definition of function-orders on these equivalence classes as:
Thus, if we know that some element in $X$ is functionally less than some element in $Y$, then we can assert this order on the corresponding equivalence classes as well. This makes it trivial to see how $\prec_f$ induces a partial order on the elements of $A_f$. Call this poset the *Induced Function Topology* of the function $f$ on the set $A$, denoted by $\tau_f(A)$. It has been named so keeping in mind the visualization expected out of this notion. If one draws the Hasse diagram corresponding to this poset, it appears as if the elements of the set $A_f$ have been *rearranged* in the form of a surface, with edges and contours, all induced by the map $f$. The diagram denotes the effect of application of this function on the elements of $A$ in a recursive manner, showing all possible orbits and equivalence classes. A study of this structure will help us come up with useful results, thus enabling us to see what happens when a particular function is used in a recursive call.

As an example, consider the function, $f : \{0, 1, 2, 3, 4, 5\} \rightarrow \{0, 1, 2, 3, 4, 5\}$, defined as below:

$$f(0) = 2, f(1) = 2, f(2) = 3, f(3) = 4, f(4) = 5, f(5) = 3.$$ 

The Hasse diagram corresponding to $\tau_f(\{0, 1, 2, 3, 4, 5\})$ is given as:

![Hasse diagram](image)

Here, $\{x\}$ represents the equivalence class for the element $x$ (which was earlier denoted by $X$). If this diagram is converted to a directed graph, with no new edges or vertices, and all existing edges pointing upwards, we get the structural equivalent of a tree (possibly with variable arity for its nodes). More precisely, the induced function topology of any function over a given set, when viewed as such a directed graph, forms a forest of such trees. The root of each tree is its greatest node and is the only node which may correspond to an equivalence class having more than one element. All other nodes in the tree must form singleton equivalence classes.

We call the collection of all roots of all trees in this forest (i.e. the set of maximal elements in the poset) the *Base Set*, for reasons that will be obvious shortly. Similarly, the trees also contain minimal nodes, from which the edges point in the outward direction, and that have no incoming edges. These nodes are necessarily either singleton equivalence classes or represent a complete
tree in them, and form part of a collection named as the *Generator Set*. The intersection of base set and the generator set forms the *Fixed Point Set*. It can be verified that the elements of the fixed point set form fixed points of the function, since they satisfy the condition $f(x) = x$ for every $x$ in this set. In case this set contains an equivalence class, say $X$, containing more than one element of $A$, we can view this as being equivalent to saying $f(X) = X$, which can stand for the fact that the domain and range of $f$ is $X$ and for every $x \in X$, there exists an $n \in \mathbb{N} \setminus \{0\}$ such that $f^{(n)}(x) = x$. This $n$ must be same for every element of $X$.

The recursive computation of $f$ starting at any member of the generator set takes us up the corresponding tree and it ends at some element of the corresponding base set. This path is along the members of the orbit of the element we started at. No computation is possible beyond the base set. Hence, if $f$ was used, directly or indirectly, to represent a recursive function, we would need to specify its value at all points in the base set because all inputs to $f$ would eventually require computing the value of $f$ at some element in this set. Hence, this set has been named appropriately as the Base Set. The set of values of $f$ at the elements in the base set comprises the set of Base Conditions for $f$ to terminate.

The generator set contains the smallest set of elements of $A$, which are enough to describe the value of $f(x)$ for all $x$ that belongs to $A$, hence its name. If we start computing $f(x)$ from any $x$ that belongs to the generator set, we move up $\tau_f(A)$ until we reach the base set. During this journey, we compute $f(y)$ for all $y$ that lie in our path. This way, starting at $x$, $f$ is computed for all $y$ that connects $x$ to the base set. For the example depicted through figure 1, the base set is $\{3, 4, 5\}$, the generator set is $\{0, 1\}$ and the fixed point set is empty, i.e. no fixed points of $f$ exist.

### III. The Fibonacci Function: An Example

The technique described above can be used in a slightly different way as well. Consider the well-known Fibonacci series, and the corresponding recursive relation:

$$f(x) = f(x-1) + f(x-2)$$

The two fixed points, usually stated for this relation, are at $x = 0$ and $x = 1$, i.e. $f(0) = 0$ and $f(1) = 1$. These also form the base conditions for $f$. If we want to visualize a recursive call of the kind where the function is called more than once for the computation to complete, we may have to modify the definition so that we have only one recursive call in the definition. This will simplify the analysis greatly. The domain of the Fibonacci relation, defined as above, is the set $\mathbb{N}$. To write $f(x)$ in terms of a single recursive call of $f$ would be a daunting task, and so far the
authors have not come across any definition of the Fibonacci relation, which has been written this way. However, we can try increasing the number of arguments to \( f \) and see if that helps [11].

Consider the following relation:

\[
g(x,y,z) = \begin{cases} 
x & \text{if } z = 0 \\
y & \text{if } z = 1 \\
g(y,x+y,z-1) & \text{otherwise}
\end{cases}
\]

We can compute \( f(x) \) using \( g(x,y,z) \) by observing that \( f(x) = g(0,1,x) \) for all \( x \). The advantage of using this modification is that we now have only one recursive call in the definition and that brings us back on track to analyze the Fibonacci function using its induced function topology. The domain of \( g(x,y,z) \) is \( \mathbb{N} \times \mathbb{N} \times \mathbb{N} \), which forms our set \( A \) in this case. Since we know that the third argument decreases till it reaches one, any 3-tuple \((x,y,z)\) that belongs to \( A \) must reach \((x',y',1)\) after \( z-1 \) recursive calls (We ignore the trivial case where \( z = 0 \)). The corresponding path must also compute the value \( g(x'',y'',u) \) for each \( u \) less than \( z \). Hence, if we only consider the argument \( z \) in the function \( g \), we are left with the following view of the function:

\[
g(*,z) = \begin{cases} 
x & \text{if } z = 0 \\
y & \text{if } z = 1 \\
g(*,z-1) & \text{otherwise}
\end{cases}
\]

Here, \( * \) has been used to denote variables which do not affect the structure of the Hasse diagram for \( \tau_s(A) \). The base set in this case is \( \{0,1\} \), which implies that the base conditions required to terminate this recursive call require us to specify the values of \( g(*,z) \) at \( z = 0 \) and \( z = 1 \), i.e. the value of \( x \) and \( y \). With this we can be sure that \( g(x,y,z) \), and consequently \( f(z) \), would terminate for all \( z \) as long as \( x \) and \( y \) are known.

The above formulation can be generalized for all linear recurrences, which can then be proved to terminate under certain conditions. Consider the linear recurrence for the function \( f: \mathbb{N} \to \mathbb{N} \) defined as:

\[
f(n) = c + \sum_{i=1}^{B} a_i f(n - i)
\]

The functions, defined this way, occur frequently in computer applications. Here, \( B \) represents the amount of memory this recurrence entails, i.e. the amount of dependency each computation of \( f(n) \) has on previously computed (and perhaps, stored in some form to improve efficiency)
values of $f(.)$ for smaller values of $n$. Similar to the Fibonacci function above, if we devise a new function $g(.)$ to convert $B$ recursive calls in $f(.)$ to a single recursive call, this would be given by the recurrence below.

$$g(d_0, d_1, ..., d_{B-1}, n) = \begin{cases} 
\text{d}_n & \text{if } 0 \leq n < B \\
\left( \sum_{i=1}^{B} d_{i-1} \right)_{n-1} & \text{if } n \geq B
\end{cases}$$

Similar to the previous case, we can view this formulation as:

$$g(\ast, n) = \begin{cases} 
\text{d}_n & \text{if } 0 \leq n < B \\
\left( g(\ast, n-1) \text{ if } n \geq B \right)
\end{cases}$$

Here, the set $D = \{d_0, d_1, ..., d_{B-1}\}$ forms the set of base conditions, which can also be verified by visualizing the Hasse diagram for $g\left( N^{B+1} \right)$. The base set is precisely the set D. Hence, specifying these $B$ values in the beginning would guarantee a linear time termination of $g(.)$ and consequently, for $f(n)$ for all values of $n$. In fact, we can argue similarly for all primitive recursive functions and $\mu$-recursive functions (defined in [5]) as well, using the least-number principal and mathematical induction. Interestingly, it turns out that the termination of all such functions can be proved using induced-topologies.

IV. Isomorphic Function Orders

A function, as we just saw, induces an order amongst the elements of its domain set. If we fix this set and try other functions on it, we get as many orders as the number of functions we study. For some of them, the case is similar to merely renaming the elements so that two orders, which would otherwise look different, essentially turn out to be the same. For some other cases, renaming the elements wouldn’t suffice, but the possibility of the existence of some bijection between the two posets under consideration is enough to view the associated orders similarly. The two cases indicate an isomorphism that may exist between two seemingly-different orders, induced by different functions over the same set.

**Definition 4.1 (Ordinal Isomorphism)** Let $f_1 : A \rightarrow A$ and $f_2 : A \rightarrow A$ be two functions, defined on a given set $A$. Let the orders induced by these functions be denoted by $\prec_1$ and $\prec_2$, respectively. The two posets $\left( A, \prec_1 \right)$ and $\left( A, \prec_2 \right)$ are said to be isomorphic if there exists an
order-preserving bijection \( f_3 : (A_f, \prec_1) \rightarrow (A_f', \prec_2) \) between them. In such a case, call \( f_1 \) and \( f_2 \) to be *Ordinally Isomorphic* to each other, with respect to the set \( A \). Also, call \( f_3 \) an *Ordinal Isomorphism* between \( f_1 \) and \( f_2 \).

As an example, consider the Hasse diagrams corresponding to two function orders, as shown below:

![Hasse diagrams](image)

Figure 2

The two functions are clearly different, and by no renaming of the elements of the set \( \{1,2,\ldots,8\} \) can we say that these two are similar. However, the reduced domains of the two functions seem to show some isomorphic nature. Structurally, the Hasse diagrams are exactly the same, except the names of the nodes. This naming can be dealt with using a renaming bijective function, which establishes the desired ordinal isomorphism between the two functions. What this bijection renames is not the elements in \( \{1,2,\ldots,8\} \) but the equivalence classes in this set, as formed by the orders induced by corresponding functions. Consequently, if we were to study recursive functions over \( \{1,2,\ldots,8\} \), which used either of these two functions, directly or indirectly, our analysis would be the same in all cases.

Without any loss of generalization, Def. 4.1 can be modified for functions defined on two different sets \( A \) and \( B \), instead of the same set \( A \), as:

**Definition 4.2** Let \( f_1 : A \rightarrow A \) and \( f_2 : B \rightarrow B \) be two functions, defined on two given sets, \( A \) and \( B \), which may not be isomorphic to each other. Let the orders induced by these functions be denoted by \( \prec_1 \) and \( \prec_2 \), respectively. The two posets \( (A_f, \prec_1) \) and \( (B_f, \prec_2) \) are said to be
**isomorphic** if there exists an order-preserving bijection \( f_3 : (A_{f_3}, \prec_1) \rightarrow (B_{f_2}, \prec_2) \) between them.

In such a case, call \( f_1 \) and \( f_2 \) to be **Ordinally Isomorphic** to each other, with respect to the sets \( A \) and \( B \). Also, call \( f_3 \) an **Ordinal Isomorphism** between \( f_1 \) and \( f_2 \).

This way, as long as \( \tau_{f_1} (A) \) and \( \tau_{f_2} (B) \) are isomorphic, our job is done and we can call the two functions ordinally isomorphic to each other. The presence of “extra” elements in either \( A \) or \( B \), which may be a possibility in such a case, will only show up as larger equivalence classes in the root nodes of the trees. Since these nodes refer to functionally-equal elements, we can assume them to be same, as far as our analysis is concerned. For both variants of the ordinal isomorphism, as given by Def. 4.1 and 4.2, our analysis is closer to treating \( f_3 \) as having the type \( (A \rightarrow A) \rightarrow (B \rightarrow B) \), rather than visualizing it as a function between two partially ordered sets. This is because we are trying to study properties of **similar order inducing** functions under one caption to avoid redundancy in our approach. In other words, the fact that \( f_1 \) and \( f_2 \) induce orders that are similar enough to be analyzed for termination using the same ranking function is more important than the structure of posets that are formed. The latter only provides a pictorial view of the same underlying fact as the former.

The importance of studying ordinally isomorphic functions is highlighted when we try to apply this concept to studying termination using the no-infinitely-descending-chain approach, as proposed by Floyd in [12]. For the sake of simplicity, let \([f]\) denote the collection of all functions that are ordinally isomorphic to \( f \). Thus, if \( f \) is ordinally isomorphic to another function \( g \), then \([f]\) and \([g]\) must be the same. Also include in \([f]\) all functions whose induced topologies are unions of countably many posets, each belonging to \([f]\). By countably many posets we mean that the set of these posets is countable. We use this notation extensively in the next section to tackle the problem of program termination using the theory developed so far.

## V. Program Termination

Let \( S : \mathbb{N} \rightarrow \mathbb{N} \setminus \{0\} \) represent the **Successor Function**, defined as \( S(x) = x + 1 \). It can be easily seen that \( s(N) \) is a total chain, whose generator set is \( \{0\} \). In fact, such an iterated application of \( S \) over the element 0 has been used as a fundamental concept in formal arithmetic to define natural numbers as \( 0, S(0), S(S(0)), \ldots \) by Dedekind and other famous mathematicians [3,4,5]. Also, let \( P : \mathbb{N} \setminus \{0\} \rightarrow \mathbb{N} \), defined as \( P(x) = x - 1 \), be the **Predecessor Function**, whose induced topology, \( \rho\left(\mathbb{N} \setminus \{0\}\right) \), is again a total chain, but this time, with the base set as \( \{0\} \).
It may seem at first that the two functions are ordinally isomorphic. However, they are not. The function $P$ has a base set but $S$ does not. Similarly, $S$ has a generator set but $P$ does not. There is no order-preserving map which can establish an ordinal isomorphism between $S$ and $P$. As a matter of fact, if such a map had existed, then we would not require the well-founded sets to prove termination at all, as long as we are sure that the iteration is along some chain isomorphic to either $S$ or $P$. In such a scenario, functions like $g(x) = g(x+2) + 3$ would also be shown to terminate because of the ordinal isomorphism between $h(x) = h(x+2)$ and $S$, and consequently, between $h$ and $P$. Surely, this could not be correct.

To further generalize the concept, let $[[f]]$ represent a broader collection, comprising of $[f]$ and all other functions for which the induced function topologies can be embedded in some member of $[f]$.

We can convince ourselves that $[P]$ and $[S]$ are disjoint collections. They contain all those functions, which induce total orders (in the form of unions of countably many infinite chains) upon their domain sets. But what about $[[P]]$ and $[[S]]$? Are they disjoint as well? The answer is no. All finite chains can be embedded in all elements of $[P]$ as well as $[S]$. These finite chains need not be from the elements of $N$ always. Def 4.2 allows us to find the required isomorphism when these domains are different from $N$. However, a question to ponder upon here is whether an infinite chain, which lies neither in $[P]$ or $[S]$, exists. The following lemma helps in this regard.

**Lemma 5.1** All total chains, which are unbounded from above and below, belong to $[S]$.

**PROOF:** For the proof of this lemma, we will use the fact that the successor function, $S':\mathbb{Z} \rightarrow \mathbb{Z}$ now defined over the set of all integers, induces a total order on $\mathbb{Z}$, but without a base set and a generator set. However, it still remains a bijection in the sense that a unique successor of every integer exists in $\mathbb{Z}$. Furthermore, any infinite chain of this sort is ordinally isomorphic to $S'(\mathbb{Z})$ and hence, proving that $S'[S]$ would be sufficient for the proof. To check this, let $f:A \rightarrow A$ denote a function for which $f(A)$ is a doubly unbounded infinite chain. For any element $x \in A$, one can view this chain as:

$$... \prec_f f^{(-2)}(x) \prec_f f^{(-1)}(x) \prec_f x \prec_f f^{(0)}(x) \prec_f f^{(1)}(x) \prec_f f^{(2)}(x) \prec_f ...$$

Thus, a desired ordinal isomorphism can be established taking the superscripts of $f$ in the chain above, i.e. using the function $:A \rightarrow \mathbb{Z}$, defined as $(y) = n \text{ if } y = f^{(n)}(x)$. Clearly, $A$ is ordinally isomorphic to $S(\mathbb{Z})$. Moving further with the proof of lemma 5.1, we would now like to split $S(\mathbb{Z})$ into two semi-infinite chains, each with a minimal element but no maximal
element. This split happens around a point, which we will call the pivot. The choice of pivot is completely arbitrary. For the sake of simplicity, let us take $x$ as our pivot. Perform this split operation using the function $g: \mathbb{Z} \rightarrow \mathbb{Z}$ as:

$$g(x) = \begin{cases} x + 1 & \text{if } x \geq 0 \\ x - 1 & \text{otherwise} \end{cases}$$

The topology, $\tau_g(\mathbb{Z})$, thus induced, is union of two semi-infinite chains with the minimal elements being 0 and -1. Hence, the generator set for $\tau_g(\mathbb{Z})$ is \{0, 1\}. No maximal elements exist. Each chain in $\tau_g(\mathbb{Z})$ belongs to [S], and hence, if we form a pair of values (0, -1), (1, -2), … and so on, we can see that the function $g(x)$ belongs to [S] as well. Consequently, the function $S'$ and $f(x)$ belong to [S], and this completes our proof.

We are now in a position to state the following theorem regarding functions that induce total orders on their domain sets.

**Theorem 5.1** Every function $f: A \rightarrow B$, where $A$ and $B$ are countably infinite sets and $B \subseteq A$, that induces a total order on the elements in $A$, belongs either to [P] or to [S], but not both.

**PROOF:** We start the proof by noticing that the total order induced by $f$ will either be a semi-infinite chain (with either a minimum element or a maximum element) or a doubly-infinite chain (with neither a minimum nor a maximum element). A finite chain is not possible because the set $A$ is countably infinite. In case of a semi-infinite chain with a minimum element, say $M$, we can define a bijection, $g: A \rightarrow \mathbb{N}$, as:

$$g(x) = n \text{ if } x = f(n)(M)$$

This establishes the required ordinal isomorphism between $f$ and $S$. Thus, $f$ belongs to [S]. A similar argument can be given if $f(A)$ had a maximum element instead. However, in this case, $f$ belongs to [P] instead. For the last case when $f(A)$ is a doubly infinite chain, lemma 5.1 establishes the proof that $f$ belongs to [S] in such cases.

A point to note here is that [P] represents all functions, which surely terminate in a finite number of iterations because every member of [P] contains a base set, which can be reached from any node in $\tau_P(\mathbb{N})$ in a finite number of steps. Thus, there cannot be an infinitely descending
sequence here. Similarly, \([S]\) represents all functions, which are surely non-terminating. There is no base set to converge to. Hence, all functions in \([S]\) must compute forever.

Coming back to \([\{P\}]\) and \([\{S\}]\), the common elements, or the finite chains (and their unions), represent functions whose reduced domains have a well-defined generator set and a well-defined base set. Such functions would surely terminate, if the base conditions were specified. This is not a daunting task once the base set is well defined. Hence, we can easily conclude that all functions that induce a finite total order on their domain sets can be made to terminate under all circumstances by specifying the base conditions for the elements in the base set.

**Definition 5.1 (Class of Terminating functions)** The class, Term, of all terminating recursive functions is the smallest class of functions satisfying the following:

1. All members of \([\{P\}]\) belong to Term, i.e. \([\{P\}] \subseteq \text{Term}\)
2. If \(g\) is a function that is ordinally isomorphic to some member in Term, then \(g \in \text{Term}\) as well.
3. If \(f\) is a function such that \(f\) is a poset (not necessarily a chain), with an enumerable base set, then \(f \in \text{Term}\).
4. If \(f\) can be expressed as a union of countably many subsets, each of which is ordinally isomorphic to some member in Term, then \(f \in \text{Term}\).

A brief explanation for the requirement of enumerability of base sets is required here. The idea is that for each element in the base set, we need to specify the base condition(s), which serve as stopping points for our computation. If this set was not enumerable, there would be no effective way of specifying the base conditions for all elements of the set. This can lead to either an ambiguity in determining whether the function really terminates, or to a non-determinism in our computation.

**VI. Ranking Functions**

A ranking function is used to prove termination of programs by mapping the set of program variables into a well-founded set so that there will not exist any infinitely-decreasing sequence of values as the program variables change during execution. The following lemma relates this classical concept to the theory developed so far.
Lemma 6.1 Every ranking function is an ordinal isomorphism between the predecessor function, \( P : \mathbb{N} \rightarrow \mathbb{N} \setminus \{0\} \) and some \( g \in \text{Term} \). In other words, every ranking function acts as an ordinal isomorphism between two terminating functions.

PROOF: To prove this lemma, let \( R : V \rightarrow W \) be the ranking function for a given program \( Q \). Here, \( R \) maps the set, \( V \), of program state variables into some well-founded set, \( W \). Let \( W' \) represent the identity function for the set \( W \). During the execution of \( Q \), the state variables make transitions from, say, \( v_1 \) to \( v_2 \), and then to \( v_3 \) and so on. For \( Q \) to terminate, we require the set \( \{ R(v_1), R(v_2), \ldots \} \) to be non-infinitely decreasing. Let \( f : V \rightarrow V \) be the function governing the state transitions of \( Q \), i.e. \( f(v_i) = v_{i+1} \) for all \( i = 1, 2, \ldots \). The knowledge of \( Q \) allows us to uniquely define such an \( f \). Hence, we have \( v_i \prec_f v_{i+1} \) for all \( i \), assuming that the states are distinct.

At the point of termination, which is guaranteed to be achieved since we know that a ranking function exists for \( Q \), the state reached is, say, \( v_M \). Then we have \( v_i \prec_f v_M \) for all \( i < M \) and consequently, we have \( f \in \lfloor [P] \rfloor \). Now, the set \( \{ R(v_1), R(v_2), \ldots \} \) can be rewritten as \( \{ R(v_1), R(f(v_1)), R(f(f(v_1))), \ldots \} \). Thus, \( R \) acts like a function between \( f(V) \) and \( W' \), each of which belongs to \( [P] \). This \( R \) is also an order preserving one-one function, and with the correct choice of \( W' \), we can make \( R \) an onto function as well. Thus, \( R \) acts as an ordinal isomorphism between \( f \) and \( W' \). The function \( W' \) can be replaced by any other terminating function because the induced topology of every terminating function is necessarily well founded (this is so because of the existence of a base set). Hence, the lemma is proved.

We can extend this analysis further by noticing that the arguments to \( R \) in its repeated application to the state variables are continually decreasing, which causes the computation of \( R \) to halt after a finite number of steps. Thus, \( R \) itself is terminating. We can then state the following theorem.

Theorem 6.1 Every ordinal isomorphism between two terminating functions must be a terminating function itself.

PROOF : The theorem follows directly from Lemma 6.1 and the terminating nature of all ranking functions, as shown above.

An important observation here is that any program for which a ranking function can be written, is terminating, can thus be used as a ranking function for some other program as well. A particularly interesting case would be if some program can act as a ranking function for itself, i.e. the program that can prove its own termination. Such a program would be complete in the sense
of being able to compute what is required of it as well as help in proving its own termination. Note that this is not equivalent to the program going through some extra computational steps to prove its termination, i.e. the program does not have a function like, `proveMyTermination()`, which it may call for the required purpose. The essence here is that the mathematical function, which the program simulates, is isomorphic to some “terminating modification” of the same program. The authors would like to express this thought in the form of the following proposition:

**Proposition 6.1** There exists a class of functions, say \( R \), each member of which can act as a Turing’s Ranking Function for itself. Call each function in this class, a *Self-Ranking Function*.

As a consequence of these observations, we can visualize an induced topology which is analogous to a superset of the induced topologies of all terminating functions. Assume a tree of arbitrarily large arity, but having no bound on the number of nodes. The root of this tree comprises the base set, while all other nodes are a part of some chain in the tree, which is also the path from that node to the root node. Thus, this tree can be represented as a union of all such chains, one for each of its elements, and is thus, isomorphic to a terminating function. A collection of infinite such trees is also isomorphic to some terminating function if the base set is countable. Such a collection is our desired generalized topology for a special terminating function, called the *Universal Terminating Function*, denoted by \( U \). Thus, any function for which the induced function topology can be embedded in \( U \), is necessarily terminating, and equivalently, and has an associated Turing’s Ranking Function.

**Proposition 6.2** There exists a *Universal Terminating Function*, \( U \), which induces a function order isomorphic to that of an infinite collection of infinite trees, with a countable base set, such that all terminating functions (over any domain set) induce orders which can be embedded in that induced by \( U \). Call \( \tau_u \) the *Universal Terminating Topology*.

In simpler words, all terminating functions are, in a way, *sub-functions* of the universal terminating function, \( U \), which then forms the largest class \([U]\) of all functions which halt deterministically for every input.

**VII. Conclusion**

Proving that a given program terminates, though undecidable, has never discouraged us from devising methods to study termination criterion for restricted instances of the generalized Halting Problem. Inspired by Floyd’s approach of using well-founded sets, this paper formalized the
notion of an order induced by a given function over the elements of its domain. Properties of this order were studied and various interesting observations were made on the isomorphic structures that arose. Finally, these were used to arrive at a generalization made about terminating recursive functions. The main result of this study is proving that every program which induces an order isomorphic to some terminating function, must terminate itself, and induce an order which can be embedded in the universal terminating topology.

VIII. References

[1] Alligood K. T., Sauer T. D., Yorke J. A. “Chaos : An Introduction to Dynamical Systems”, Springer-Verlag New York, Inc., 2008

[2] Davey B. A., Priestley H. A. “Introduction to Lattices and Order”, Second Edition, Cambridge University Press 1991, 2002

[3] Cantor G., “Contributions to the Founding of the Theory of Transfinite Numbers”, Dover Publications, Inc., New York

[4] Causey R. L., “Logic, Sets, and Recursion”, Second Edition, Jones & Barlett Publishers, Inc., 2010

[5] Odifreddi P., “Classical Recursion Theory”, Study in Logic and Foundations of Mathematics, Volume 125

[6] Blass A., Gurevich Y., “Program Termination and Well Partial Orderings”, ACM Transactions on Computational Logic, Vol. V, No. N, December 2006

[7] Boyer R. S., Moore J. S., “A Mechanical Proof of the Unsolvability of the Halting Problem”, ICSCA-CMP-28, July 1982

[8] Tarski A., “A Lattice-Theoretical Fixpoint Theorem and its Applications”, Pacific J. Math, 5(1955), 285-309

[9] Cook B., Podelski A., Rybalchenko A., “Proving Program Termination”, Communications of the ACM, Vol. 54, No. 5, May 2011

[10] Dershowitz N., Manna Z., “Proving Termination with Multiset Orderings”, Stanford Artificial Intelligence Laboratory, Memo AIM 3-10, March 1978

[11] McCarthy J., “A Basis for a Mathematical Theory of Computation”, In P. Braffort and D. Hirschberg, editors, Computer Programming and Formal Systems, pages 33-70, North-Holland, Amsterdam, 1963

[12] Floyd R. W., “Assigning Meanings to Programs”, Proc. Symp. In Applied Mathematics, Vol. 19, 1967
[13] Turing A., “Checking a Large Routine”, In Report of a Conference on High Speed Automatic Calculating Machines, 1949

[14] Lagarias, J. C., “The 3x+1 problem and its generalizations”, American Mathematical Monthly, 92:3-23, 1985

[15] Rosen, K. H., “Discrete Mathematics and its Applications with Combinatorics and Graph Theory”, 7th Edition, Indian Adaptation by Krithivasan, K., Tata McGraw Hill Education Pvt. Ltd., 2011