A NOTE ON THE COMPLEXITY OF
GOEDEL NUMBERINGS AND ISOMORPHISMS

Jim Kadin

TR 81-466
August 1981

Department of Computer Science
Cornell University
Ithaca, New York  14853
A NOTE ON THE COMPLEXITY OF
GÖDEL NUMBERINGS AND ISOMORPHISMS

Jim Kadin

TR 81-466
August 1981

Department of Computer Science
Cornell University
Ithaca, New York 14853
ABSTRACT

Some problems involved in looking at recursive function theory and thinking about the complexity of computations are discussed. Complexity classes of Gödel numberings are studied where a Gödel numbering is in a given complexity class if every other Gödel numbering can be translated into it by functions in a given complexity class. In particular, we look at the class of numberings that can be translated into by polynomial time mappings (GNP) and the class that can be translated into by linear bounded automaton mappings (GNLBA). It is shown that polynomial time isomorphisms and LBA computable isomorphisms between two Gödel numberings relate the complexity of the syntax of the numberings. Since the classes GNP and GNLBA contain Gödel numberings with arbitrarily hard syntax, not all members of these classes are isomorphic by polynomial time or LBA mappings. LBA computable isomorphisms can be found between members of GNLBA whose syntax is LBA recognizable. A similar result holds for polynomial time isomorphisms and GNP if P=NP.

1. Introduction

We assume familiarity with the basic notions of recursive function theory and of computational complexity theory.

Rogers [4] defines (acceptable) Gödel numberings of partial recursive functions from a purely recursion theoretic view. The class of numberings that he defines forms the natural class of programming languages from the recursion theoretic viewpoint. Elegantly, all the elements of this class are recursively isomorphic and hence are all equivalent from this viewpoint. With respect to computational complexity, it is clear that all acceptable Gödel numberings are not equivalent since one can derive pairs of numberings for which any isomorphism between the pair is arbitrarily hard to compute. What class of Gödel numberings makes sense to be called the natural programming languages from the computer science viewpoint? In what sense are the members of this class equivalent? We discuss some of the issues involved in these questions.

In doing this we discuss the concepts of Gödel numbering, translation, and isomorphism. We give a definition of Gödel numbering that is equivalent to the definition found in recursive function theory but which we feel makes more intuitive sense from a computer science viewpoint. Our definition is not necessarily new; we discuss it and the notions of translation and isomorphism in detail because we feel there have been misunderstandings in the literature.
In short, we define a Gödel numbering (GN) to be a recursive set of strings \( L \) with a certain semantics (i.e. a partial recursive universal function) such that \( L \) is a universal programming language. Two GN's are isomorphic if there is a recursive mapping between the GN's that defines a 1-1 correspondence between the two sets of strings (programs) which preserves functional equivalence of the programs.

The basic idea we present is that for a GN to be natural or feasible, its programs must be easy to recognize. That is, it must have a natural, feasible syntax. In particular we show that a GN with an easy syntax (say polynomial time recognizable) cannot be isomorphic to a GN with a hard syntax by an easily computable isomorphism (polynomial time). Hence to define the class of feasible GN's so that all such GN's are isomorphic by feasibly computable functions, some restriction must be made on program syntax.

As an approximation to defining feasible GN's, Hartmanis and Baker [1] define the complexity of a GN \( L \) by the computational complexity of the translations from other GN's into \( L \). For instance, \( L \) is a polynomial time GN if every GN can be translated into \( L \) by a polynomial time bounded translation. They then define the class GNP to be the class of all polynomial time GN's. GNC is defined similarly for any complexity class C. Suppose a class C represents the class of feasibly computable functions. Then it seems that if a GN \( L \) is not a member of GNC, then \( L \) is not one of the natural GN's since common GN's such as Turing machines or Algol cannot be translated into \( L \) by feasible translations. Hence this concept of complexity of translations into a GN captures at least some of the important characteristics of nice GN's.

Machtley, Winklmann, and Young [3] study complexity classes of GN's further. They prove that all members of GNLBA (GN's into which every GN can be translated by a linear bounded automaton) are LBA-isomorphic (isomorphic by LBA-computable mappings). They also show that if P=NP, then all members of GNP are P-isomorphic (isomorphic by polynomial time mappings). We show that the isomorphisms produced in these results are not necessarily isomorphisms in the sense that they define a 1-1 correspondence between any two members of GNLBA or GNP. This was not obvious because Machtley, Winklmann, and Young use the technique of extending each GN \( L \) to the set of all strings over an alphabet by saying a string not in \( L \) is a program to compute the empty function. We show that this technique significantly alters languages, and that isomorphisms between the extended languages do not necessarily define 1-1 correspondences between the original languages.

More recently, Hartmanis [2] defines the class of P-natural GN's. A GN is P-natural if every other GN can be translated into it by a 1-1 polynomial time mapping whose inverse is also polynomial time computable. He proves that
all such GN's are P-isomorphic. Our results show that as in the case of GNP and GNLBA, the isomorphisms produced do not necessarily define 1-1 correspondences between arbitrary P-natural GN's. Note that in the version to be published, Hartmanis has corrected the definition of P-natural to include that the syntax be recognizable in polynomial time.

2. Goedel Numberings and Translations

Rogers defines a GN $\phi$ as a mapping from integers to partial recursive functions such that it is (1) effectively decidable whether an integer $i$ maps to a function, and (2) $\phi$ is effectively intertranslatable with some standard numbering (say lexicographically ordered Turing machines). A translation $t$ from $\phi$ to Turing machines is a recursive function mapping integers to integers such that if $i$ is an integer that $\phi$ maps to a function, then $\phi_i = M_t(i)$. Here we use the standard notation that $\phi_i$ represents the function that $\phi$ maps $i$ to. Similarly $M_j$ is the function computed by the $j^{th}$ Turing machine in the standard numbering. Note here that we only need to define $t$ on the set of integers that are "valid programs" for $\phi$, and since that set is recursive, given an arbitrary integer, we can tell whether or not to bother trying to translate it.

From a recursive function theory viewpoint one can equivalently define GN's as mappings from sets of strings (programs) to partial recursive functions. This definition is equivalent in recursive function theory since one can always define 1-1 recursive mappings from the set of integers onto any (infinite) recursive set of strings. As computer scientists, we are used to working with programs, and we tend to think of GN's as programming languages. Many of the common things done to programs, executing them, composing them, applying s-m-n functions to them, etc., are simple operations for most languages once we get hold of the programs to operate on. Hence it makes sense to think of GN's as mappings from strings (programs) rather than integers. In fact, in terms of computational complexity, the two definitions of GN's are not equivalent unless the mappings between integers and programs are easy to compute.

For instance, consider the standard lexicographic ordering of Turing machines. It is not clear presently whether given an integer $i$, one can generate the $i^{th}$ Turing machine (the $i^{th}$ Turing machine table) in polynomial time. If indeed this computation is not possible in polynomial time, then there is a big problem with this numbering. Basically the problem is that to a person in computational complexity theory concerned with feasible computations, the integer $i$ and the $i^{th}$ Turing machine table are not equivalent. To a person in recursion theory, the two are equivalent. In particular, given $i$, one may not be able to even begin to run the $i^{th}$ Turing
machine in a reasonable amount of time even if the \(i^{th}\) Turing machine runs very quickly. Similarly, it is not clear that one can compute the s-l-l function for Turing machines in polynomial time using this numbering. It seems that given integers \(i\) and \(x\), to compute the integer \(s(i,x)\) such that \(M_{s(i,x)}(y) = M_i(x,y)\) one must generate the \(i^{th}\) Turing machine table, add states to it to write \(x\) on its tape, and find the integer that is the name for the new Turing machine. The computation described certainly cannot be done in polynomial time unless the \(i^{th}\) Turing machine table can be generated in polynomial time. To escape these problems, we feel that it is much simpler to use each Turing machine table as a name (an index) for itself. Then there is no problem in mapping back and forth between indices and tables (programs).

In the general case, programming languages are sets of strings. Hence it seems to make more sense to use the programs themselves as names rather than assigning a number to each one. We will say a GN is a recursive set of strings \(L\) that has a partial recursive universal function and such that each Turing machine can be effectively translated to an equivalent string in \(L\). More formally:

**Definition:** A GN is a recursive set \(L\) and a partial recursive function \(\Phi_L\) such that \(\forall x \in L \exists y [\Phi_L(x,y)]\) is a partial recursive function, and \(\exists\) recursive \(f : TM \rightarrow L\) such that \(\forall M \in TM\), the function computed by \(M\) is \(\lambda y [\Phi_L(f(M),y)]\).

\(TM\) is the set of all one tape Turing machine tables written in some reasonable syntax in an alphabet of two or more symbols. Assuming Church's thesis that all partial recursive functions are Turing computable, the universal function for a GN \(L\) is Turing computable. Hence using the s-l-l function for \(TM\)'s we can easily translate \(L\) into \(TM\). Thus our definition is equivalent to saying that a GN is a programming language that is effectively intertranslatable with \(TM\). We will use the terms strings in \(L\), programs of \(L\), and valid programs in \(L\) interchangeably.

Note that the s-l-l function for \(TM\) can be made to run in linear time in both arguments, and it can be made \(l-l\), length increasing, and linear time reversible. Hence there exist very nice translations from any GN into \(TM\). In particular, \(TM\) is \(P\)-natural and is a member of the classes \(GNP\) and \(GNLBA\).

As explained above, a translation \(t\) from GN's \(L_1\) to \(L_2\) is a (recursive) mapping that preserves functional equivalence.

**Definition:** A recursive function \(t : \Sigma^* \rightarrow \Pi^*\) is a translation from \(L_1 \subseteq \Sigma^*\) to \(L_2 \subseteq \Pi^*\) if \(\forall x \in L_1 \lambda y [\Phi_{L_1}(x,y)] = \lambda y [\Phi_{L_2}(t(x),y)]\).
This is the definition from recursive function theory. Note that nothing is said about what \( t \) does to strings not in \( L_1 \). Since \( L_1 \) is recursive, \( t \) is free to detect strings not in \( L_1 \) and do anything to such strings. If the computational complexity of translations is of interest, this nice feature goes away. For instance, if \( L_1 \) is recognizable in polynomial time, and \( L_2 \) is not, then there is no polynomial time translation \( t : L_2 \rightarrow L_1 \) such that all strings not in \( L_2 \) are mapped to strings not in \( L_1 \). Every polynomial time translation from \( L_2 \) to \( L_1 \) maps invalid \( L_2 \) programs to valid \( L_1 \) programs (a Ptime translation that did map invalid programs to invalid programs would provide a way to recognize \( L_2 \) in Ptime). This observation is also true of LBA translations. It is this fact that essentially prohibits the existence of polynomial time or LBA isomorphisms between GN's with hard syntax and those with easy syntax.

3. Isomorphisms

Rogers [4] defines an isomorphism between two GN's to be a recursive 1-1 correspondence carrying one numbering onto the other. We give an equivalent definition that can easily be extended to talk about the complexity of isomorphisms.

**Definition:** Two GN's \( L_1 \) and \( L_2 \) are recursively isomorphic if \( \exists \) recursive functions \( f \) and \( g \) such that \( f \) restricted to \( L_1 \) is a 1-1 translation mapping \( L_1 \) onto \( L_2 \), and \( g \) restricted to \( L_2 \) is a 1-1 translation mapping \( L_2 \) onto \( L_1 \), and \( \forall x \in L_2 \ g( x ) = f^{-1}( x ) \).

We will call such a \( g f^{-1} \), but it is important to remember that \( g \) is not required to be \( f^{-1} \) on strings not in \( L_2 \). In fact \( f^{-1} \) may not be defined for strings not in \( L_2 \) as \( f \) may not map onto every string not in \( L_2 \).

Note that this is different from the notion of isomorphism used to talk about two sets of strings (say NP-complete sets) being isomorphic. For example, we say two NP-complete sets \( A \subseteq 2^x \) and \( B \subseteq 2^y \) are isomorphic via \( f \) if \( f \) maps \( 2^x \) 1-1 onto \( 2^y \) such that \( x \in A \iff f(x) \in B \). For isomorphisms between NP-complete sets, we are very much concerned about what happens to strings that are not in the sets.

**Definition:** GN's \( L_1 \) and \( L_2 \) are C-isomorphic for some complexity class \( C \) if there is an isomorphism \( f, f^{-1} \) between \( L_1 \) and \( L_2 \) such that \( f \) and \( f^{-1} \) are both in \( C \).

As with translations, when the computational complexity of isomorphisms is of interest, questions crop up as to what do isomorphisms do with invalid programs. If complexity is ignored, then isomorphisms can always check their inputs and map invalid strings to invalid strings. If the cardinalities of
the sets of invalid programs are the same, then a recursive isomorphism can be made 1-1 on all strings. Such an isomorphism can also be made to define a 1-1 correspondence between the sets of invalid programs since both these sets are recursive. Thus if there are no cardinality problems, a recursive isomorphism between two GN's can be extended to a recursive "set" isomorphism that preserves functional equivalence of valid programs. Thus for recursive isomorphisms, requiring that the isomorphisms be 1-1 only on valid programs is unnecessary.

On the other hand, it is clear that a P-isomorphism that is 1-1 for all strings must map all invalid strings to invalid strings. Hence there can be no such isomorphism between a Ptime recognizable GN and a GN that is not Ptime recognizable. It seems that the important thing about two GN's being isomorphic is the 1-1 correspondence between the two sets of valid programs. If this is all that matters, we could say that we will not make any requirements as to what an isomorphism does to invalid programs. We might not want to require that an isomorphism be 1-1 on invalid strings or map invalid strings to invalid strings. For what follows, we will not make any assumptions as to what isomorphisms do with invalid strings.

It is clear that if two sets (not GN's) are P-isomorphic, then the complexities of recognizing the two sets are related by a polynomial. The interesting thing is that even if we let isomorphisms do anything to invalid strings, then the complexities of recognizing the sets of valid programs of two P-isomorphic GN's are also related by a polynomial. We now show how this works.

\textbf{Lemma:} If GN's \( L_1 \) and \( L_2 \) are LBA-iso (P-iso), then \( L_1 \) is LBA (Ptime) recognizable if and only if \( L_2 \) is LBA (Ptime) recognizable.

\textbf{Proof:} Suppose that \( L_1 \) and \( L_2 \) are GN's that are isomorphic by LBA computable \( f \) and \( f^{-1} \). Suppose also that \( L_2 \) is recognizable by an LBA, we will show how an LBA can recognize \( L_1 \) (the other direction is symmetric).

Given \( x \), to decide if \( x \in L_1 \), compute \( f(x) \) counting steps and space used to be sure the computation only uses the space resources allowed by the LBA \( f \). Recall that an infinite loop in a fixed amount of space can be detected. If the computation uses too much space or gets into an infinite loop, then \( x \) is not in \( L_1 \) since \( f(y) \) is LBA computable \( \forall y \in L_1 \) (here we are assuming \( f \) may do weird things to strings not in \( L_1 \)). If the computation halts within the correct bounds with \( f(x) = z \), run the LBA recognizer for \( L_2 \) on \( z \). If \( z \) is not in \( L_2 \), then \( x \) is not in \( L_1 \) clearly. If \( z \in L_2 \), then \( x \) is not necessarily in \( L_1 \) since \( f \) may map invalid programs to valid ones. In this case, compute \( f^{-1}(z) \) (by an LBA). This is guaranteed to produce a valid \( L_1 \) program \( y \). Then since \( f, f^{-1} \) define a 1-1 correspondence between \( L_1 \) and \( L_2 \), \( y = x \) if
and only if \( x \) is valid (i.e. \( x \in L_1 \)).

The computation described can be carried out by an LBA since the class of LBA computable functions is closed under composition, addition, and the addition of clocks to count steps. Hence \( L_1 \) is LBA recognizable if \( L_2 \) is and vice versa. This computation works similarly for Ptime functions. QED

Note that this lemma is true for any class \( C \) that is closed under composition, addition, and "step counting".

So an LBA (Ptime) recognizable GN \( L_1 \) cannot be LBA (Ptime) isomorphic to a GN \( L_2 \) unless \( L_2 \) is also LBA (Ptime) recognizable. Since there exist members of GNLBA that are not recognizable by an LBA, all members of GNLBA are not LBA-isomorphic. Similarly for GNP and for the class of P-natural GN's.

**Theorem:** There exist members of GNLBA that are not LBA-isomorphic. There exist members of GNP that are not P-isomorphic. There exist P-natural GN's that are not P-isomorphic.

Proof: Recall that TM is a P-natural GN that is also in GNP and GNLBA. Let \( A \) be any recursive set such that \( A \) is neither Ptime nor LBA recognizable. Let \( L = \{<y,M> : y \in A \text{ and } M \in TM\} \) be a GN where the program \(<y,M>\) on input \( x \) computes \( M(x) \) for all \( y \in A \). \( L \) is P-natural and is in GNP and GNLBA since we can easily translate \( TM \) into \( L \) by picking some element \( y_0 \in A \) and mapping \( M \mapsto <y_0,M> \forall M \in TM \). We can easily translate any other GN \( L_0 \) into \( L \) by composing an easy translation from \( L_0 \) to \( TM \) with this translation from \( TM \) to \( L \). If \( L \) is in \( P \) or is LBA recognizable, then \( A \) is too since \( y \in A \) if and only if \(<y,M_0> \in L \) for some fixed \( M_0 \in TM \). Therefore \( L \) is not in \( P \), and \( L \) is not LBA recognizable. Therefore by the lemma, \( L \) is not P-isomnor LBA-iso to \( TM \). QED

Again this theorem is true for all classes \( C \) for which the lemma is true. For instance GNEXP, the class of exponential time GN's which is dealt with in [1] and [3], contains GN's that are not exponential time isomorphic.

We do not feel that this is a profound result that necessarily shows that the theory of feasible (say Ptime) functions is fundamentally different from the theory of computable functions. We feel that this just points out that we cannot define feasible GN's simply by looking at ease of translation. On the contrary, complexity of syntax is also very important.

Machtey, Winklmann, and Young [3] prove that all members of GNLBA are LBA-isomorphic. The isomorphisms they show exist are always between languages with an extended syntax and semantics, and hence our result does not contradict theirs. We summarize their proof in order to show how this
extension technique alters languages.

First of all, given any GN L written over an alphabet $\Sigma$, they extend L to $\Sigma^*$ by defining that any string in $\Sigma^* - L$ is a program to compute the everywhere undefined function. We will denote this extended version of L as $\hat{L}$. This extends the semantics and syntax of L. Every string in $\Sigma^*$ becomes a valid program of $\hat{L}$. This may mean the universal function for $\hat{L}$, $\overrightarrow{\lambda}$, will differ slightly from $\overrightarrow{\lambda}_L$ since $\lambda x[\overrightarrow{\lambda}(y,x)]$ is required to be undefined $\forall y \in \Sigma^* - L$. $\overrightarrow{\lambda}_L$ does not have to meet this requirement. From now on in the proof, Machtet, Winklmann, and Young only refer to the extended versions of GN's. Next they show that every member of GNLBA has LBA computable padding functions and can be translated into by 1-1 LBA computable translations. Using these two results, they show that translations can be made 1-1, length increasing LBA computable. Hence given $L_1, L_2 \in \text{GNLBA}$, there exist 1-1, length increasing LBA translations $f : \hat{L}_1 \rightarrow \hat{L}_2$ and $g : \hat{L}_2 \rightarrow \hat{L}_1$. Then by a Cantor-Bernstein argument, they prove the existence of a 1-1, LBA computable $h$ mapping $\hat{L}_1$ onto $\hat{L}_2$ that preserves functional equivalence ($h^{-1}$ is LBA computable). They conclude that all members of GNLBA are LBA-isomorphic.

The function $h$ produced by their proof is an isomorphism between $\hat{L}_1$ and $\hat{L}_2$. On the other hand, if exactly one of $L_1, L_2$ is LBA recognizable, then $h$ does not define a 1-1 correspondence between $L_1$ and $L_2$. By our lemma, $h$ must map some invalid $L_1$ programs to valid $L_2$ programs or vice versa. Hence by our definition $h$ is not an isomorphism between $L_1$ and $L_2$. $h$ does preserve functional equivalence between $\hat{L}_1$ and $\hat{L}_2$, and therefore has the nice property that $x \in L_1$ and $h(x)$ not in $L_2$ implies $x$ is a program for the empty function.

To see one way that $h$ will map an invalid program to a valid one, let $L_2$ be the only LBA recognizable language of the pair. $f$ is an LBA translation from $L_1$ to $L_2$, so it must map some invalid $L_1$ programs to valid $L_2$ programs. Recall that in the Cantor-Bernstein argument, $h(x)$ is defined to be either $f(x)$ or $g^{-1}(x)$ depending upon $x$. If $x$ is not in the image of $g$, then $h(x) = f(x)$. The easiest way to see how $h$ may not preserve program validity is to think of some invalid $L_1$ program $x$, not in the image of $g$, such that $f(x)$ is a valid $L_2$ program. For such an $x$, $h(x) = f(x)$, and hence $h$ will map $x$ to a valid program. $h$ cannot preserve program validity because $f$ cannot.

This phenomenon arises because $L$ and $\hat{L}$ are really different GN's, different programming languages. The syntax of $\hat{L}$ is trivial no matter how hard the syntax of $L$ is. This can be taken to the extreme if we let $L$ be a recursively enumerable set that is not recursive. For example, let $L = \{<M_i, M_j> : M_i$ and $M_j$ are in TM and $M_i$ halts on blank tape$\}$. $L$ is not recursive, yet it can be thought of as a GN by defining a program $<\overrightarrow{M_i}, \overrightarrow{M_j}>$ in $L$ to compute $M_j(x)$ on input $x$. Given a string $<M_k, M_j>$ that might be a
program in $L$, to run it on input $x$ we run $M_k$ on blank tape. If it halts, we then run $M_j$ on $x$. We can then extend $L$ to $\hat{L}$. $\hat{L}$ is a true GN that has a trivial syntax. It is as easy to translate into $L$ and $\hat{L}$ as it is to translate into $TM$ (if $t$ translates $L_2$ into $TM$, then $\lambda x[<M_0,t(x)>]$ translates $L_2$ into $L$ and $\hat{L}$ where $M_0$ is some $TM$ that we know halts on blank tape). Thus $\hat{L}$ and $TM$ are both in GNLBA and are LBA-isomorphic, yet we are not willing to say that that implies $L$ is LBA-isomorphic to $TM$.

In summary, Machtay, Winklmann, and Young's result can be restated as: for any two members of GNLBA $L_1, L_2$, $\hat{L}_1$ is LBA-isomorphic to $\hat{L}_2$. Their result that $P=NP$ implies all members of GNP are P-isomorphic can be restated as: $P=NP$ implies $\hat{L}_1$ is P-isomorphic to $\hat{L}_2 \ \forall L_1, L_2 \in GNP$. Hartmanis' result that P-natural GN's are P-isomorphic can be similarly restated.

4. Conclusion

If we want to define the class of feasible GN's so that all members of the class are feasibly isomorphic, we must require that the class contain only GN's with a feasible syntax. The classes GNP and GNLBA and the P-natural GN's contain languages with arbitrarily hard syntax and hence contain languages that are not feasible or natural. It is not really suprising then that the classes contain non-P-isomorphic and non-LBA-isomorphic GN's.

The results of Machtay, Winklmann, and Young that are mentioned here do go through nicely with essentially the same proofs if we look at classes of GN's which restrict translation complexity and syntax complexity.

**Definition:** A GN $L$ is in the class $P$-GNP if $L \in P$ and $L \in GNP$.

**Definition:** A GN $L$ is in the class LBA-GNLBA if $L$ is LBA recognizable and $L \in GNLBA$.

We can similarly define the class C-GNC for any complexity class C.

The members of the class LBA-GNLBA are all LBA intertranslatable by translations that map invalid programs to invalid programs. Hence the problem of translations not preserving program invalidity goes away. We can skip the step of extending each GN $L$ to $\hat{L}$ and reproduce their proof to get LBA-isomorphisms between any two LBA-GNLBA's. Similarly for $P$-GNP.

**Theorem:** All members of LBA-GNLBA are LBA-iso.

**Theorem:** $P=NP$ implies all members of $P$-GNP are P-iso.

Similarly, if we require P-natural GN's to be recognizable in polynomial time, then all the P-natural GN's can be shown to be P-isomorphic with
essentially the same proofs as in [2].

We have shown that not everything in GNP is P-isomorphic. Whether or not everything in P-GNP is P-isomorphic is still an open question.

Acknowledgement

The author thanks J. Hartmanis for many valuable discussions and insights.

References


