# New Computing Paradigms Suggested by DNA Computing: Computing by Carving


Vincenzo MANCA
Università degli Studi di Pisa
Dipartimento di Informatica
Corso Italia 40, 56125 Pisa, Italy
mancav@di.unipi.it

Carlos MARTÍN-VIDE
Research group in Mathematical Linguistics and Language Engineering
Rovira i Virgili University
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
cmv@astor.urv.es

Gheorghe PĂUN
Institute of Mathematics of the Romanian Academy
PO Box 1-764, 70700 Bucureşti, Romania
gpaun@imar.ro



**Abstract.** Inspired by the experiments in the emerging area of DNA computing, a somewhat unusual type of a computation strategy was recently proposed by one of us: generate a (large) set of candidate solutions of a problem, then remove the non-solutions such that what remains is the set of solutions. This has been called a *computation by carving.* This idea leads both to a speculation with possible important consequences and to interesting theoretical computer science (formal language) questions. The speculation is that in this way we can "compute" non-recursively enumerable languages, because the family of recursively enumerable languages is not closed under complementation. The formal language theory questions concern sequences of languages with certain regularities, needed as languages to be extracted from the total language of candidate solutions of a problem. Specifically, we consider sequences of languages obtained by starting from a given regular language and iteratively applying to it a given finite state sequential transducer (a gsm). Computing by carving with respect to such a sequence of languages can identify all context-sensitive languages and can also lead to non-recursively enumerable languages; in fact, exactly the complements of recursively enumerable languages can be obtained in this way.

Then, we investigate the hierarchy of language families defined by the number of states of the used gsm's. We find that this hierarchy collapses: three states are enough. We also consider the more general problem of characterizing the recursively enumerable languages by an iterated use of a finite sequential transducer and we investigate the corresponding hierarchy of languages.

# 1 Foreword[1]

This paper deals with *dry DNA computing*, with DNA computing *in info* (as opposed both to *in vivo* and to *in vitro*).

It starts from the authors' belief/hope that one of the possible ways by which DNA computing will contribute to the computer science is by contributing to the theory of computing, by suggesting new computability paradigms and tools: new data structures (the double strand is the main one), new operations with these structures or with usual data structures, new computing models, and new computing strategies. The classic computer science is grounded on automata and grammars (or other quite similar devices: Post systems, Markov algorithms, Thue systems), all of them based on *rewriting* (in a fair understanding of the term). The nature does not "compute" by rewriting, but by crossing-over, annealing, insertion-deletion, etc. All these operations can lead to devices which are computationally complete (see, e.g., [14]). That is, the computability theory can be reconstructed by taking as basic operations such "natural" operations. Can this be of any practical interest ? What about DNA-like computing implemented in silicon (or other "classic") media ? We do not have answers to such questions, but it is clear that they are a challenge for the theorist (for the "theoretical DNA computer science" and for [theoretical] computer science in general).

We call here attention to a computing strategy which is common to most (if not all) actual or proposed experiments in DNA computing and which enables us to compute beyond Turing by using tools much weaker than Turing machines. (This is realistic exactly as much as Turing machines and Turing computability are realistic.) At least philosophically and mathematicaly, this should be obviously interesting. But, of course, this is just DNA computing *in info*...

# 2 From DNA Computing to Computing by Carving[2]

Let us recall, in a few words, the first successful experiment in the emerging area of DNA computing, that reported in [1], where a small instance of the Hamiltonian Path Problem in a graph (known to be an NP-complete problem) has been solved by purely biochemical techniques: the vertices of the graph were encoded as single stranded DNA molecules of length 20; an edge from a vertex $i$ to a vertex $j$ was encoded as a single stranded DNA molecule, also of length 20, consisting of the Watson-Crick complements of the last 10 nucleotides in the code of node $i$ and of the first 10 nucleotides in the code of node $j$. A huge number of copies of these single stranded molecules were placed in the same test tube and let to anneal. With a high enough probability, the double stranded molecules obtained in this way described all possible paths in the graph (of a length bounded by the amount of available material), including paths which were not Hamiltonian (not visiting all nodes, or visiting some nodes more than once). A "carving" phase has followed, when L. Adleman has filtered the solution obtained at the previous phase in such a way to get rid of all garbage and to conclude that at least a molecule describing a

[1]Mainly for the reviewer who failed to understand the aim of this paper.
[2]The first three sections of this paper closely follows the technical report [13].

Hamiltonian path has remained in the test tube.

Still more visible than in Adleman's case is this type of computation in the case of the DNA computation reported in [11]. The considered problem was the determination of the size of the maximal clique in a graph. (Note the qualitative difference from the problem dealt with in [1]: now one looks for a number, not for the answer to an yes-or-no question.) Making again use of the huge parallelism of DNA, a pool of molecules were created describing all subgraphs of a given graph (such a subgraph is encoded as a number over the binary alphabet, with 1 in a place identifying a vertex in the subgraph and with 0 otherwise; see details in [11]). In [11], this is explicitely called *the complete data pool*. Then, the complementary graph was considered (the graph with two nodes connected if and only if they were not connected in the original graph). Two steps were then performed. We quote from [11]: (i) "We eliminate from the complete data pool all numbers containing connections in the complementary graph"; (ii) "We sort the remaining data pool to find the data containing the largest number of 1's".

The *carving* is visible end explicit: the non-solutions are removed from the complete data pool until only solutions remain[3].

There are three important steps here: (1) creating the set of all candidate solutions, (2) creating/identifying non-solutions, (3) removing non-solutions from the set constructed at step (1). Steps (2), (3) are repeated a number of times. In the mentioned experiments, the control of these steps is ensured by a human, manually, hence the efficiency of the whole procedure essentially depends on the efficiency (and the accuracy) of each step. However, what is crucial here is that all these steps are done by biochemical means. For instance, the creation of the "complete data pool" in Adleman's case is an annealing operation, which proceeds in a highly parallel manner, and makes use of the two basic feature of DNA: the Watson-Crick complementarity and the high parallelism. (We agree with the observation in [17] that DNA is intrinsically computationally complete just because of the complementarity, which is provided by the very way the DNA molecules are build.) These two features – complementarity and huge parallelism – are also essential for the other phases of the computation.

A similar strategy is followed in other DNA computing procedures, for instance, in [2], [3], [10].

We want to point out here the fact that phase (1) is an explicit part of the procedure. This differs from previously considered procedures of "computing by filtering", by removing non-solutions. The most illustrative example is the Erathostene's sieve for "generating" the set of prime numbers. The "complete data pool", that is the (infinite) set of natural numbers, is not constructed explicitely in the procedure (but it is indeed used as an input given "for free").

[3] In a recent paper, [5], where a double stranded approach to the Hamiltonian Path Problem is proposed, T. Head, explicitely speaks about two phases of the computation: "(1) the *generation* of DNA moolecules that encode candidate solutions and (2) the *elimination* of those DNA molecules that encode non-solutions." The computing by carving is again explicit.

# 3 Computing by Carving; A Way to Compute Beyond RE ?

The previous considerations suggest a mode of computation which is not very usual in computer science or in computability theory: produce a "complete data pool" and filter it iteratively in such a way that only the solutions of the problem remain.

In formal language theory terms, this can be formulated in the following quite new way of identifying a language $L$: start from a superlanguage of $L$, say[4] $M$, large and easy to be obtained, then remove from $M$ strings or sets of strings, iteratively, until obtaining $L$. Contrast this strategy with the usual "positivistic" grammatical approach, where one produces the strings in $L$, at the end of successful derivations, discarding the "wrong derivations" (derivations not ending with a terminal string), and constantly ignoring/avoiding the complement of $L$.

In particular, $M$ above can be $V^*$, the total language over the alphabet $V$ of the language $L$ ($V^*$ is the language of all strings of symbols in $V$; the empty string is denoted by $\lambda$ and the set of all non-empty strings over $V$ is denoted by $V^+$; the length of a string $x \in V^*$ is denoted by $|x|$; for formal language theory notions and results which we will use here we refer to [18]). Therefore, as a particular case, we can try to produce $V^* - L$, the complement of $L$, and to extract it from $V^*$. As one knows, the family of recursively enumerable languages is not closed under complementation. Therefore, *by carving we can "compute" non-recursively enumerable languages* !

This is just another formulation of the fact that the family of recursively enumerable languages is not closed under complementation. But this reformulation is very natural for DNA computing, because of the following two facts:

1. We can generate any recursively enumerable language by using one of the many generative devices based on DNA-type of operations which were already investigated. This is the case with various types of controlled or distributed H systems (based on the operation of *splicing*, which describes the recombinant behavior of DNA molecules under the influence of restriction enzymes), with insertion-deletion systems, with sticker systems of various forms (based on the operation of annealing), etc. A comprehensive source of information is [14]. In particular, we can generate (in an easy way) the language $V^*$ (the "complete data pool" for our case), or any needed regular language.

2. A procedure to implement the difference of two languages in DNA terms can be easily imagined (at the theoretical level, assuming that the operations are going in the ideal fashion, without errors, mismatching, etc), and such operations were already used in the mentioned experiments, [1], [11], and are involved in the procedures proposed in [2], [3], [10], too.

   Let us imagine a general procedure of this type, for "computing" the difference $L_1 - L_2$ of two given languages $L_1, L_2$ consising of DNA double stranded molecules. More specifically, assume that both $L_1, L_2$ consist of molecules codifying some information in such a way that in the upper strands of molecules one uses only the nucleotides A and C and in the lower

[4] From "marble", to fit the idea of "computing by carving"...

strands one uses the complementary nucleotides T and G. (This gives us the possibility to speak about the *upper* and the *lower* strands of a molecule also after denaturing the molecules and separating the strands.) Moreover, let us add an occurrence of T in between each two nucleotides in the upper strand, and hence A in between each two nucleotides in the lower strand. Let us also concatenate the molecules in the two languages with some blocks $\alpha$ and $\beta$ which never appear as substrings of the strings in $L_1, L_2$. (The alternating occurrences of T in the upper strands and of A in the lower strands make this possible.) The language $L_1$ bounded in this way is placed in a test tube. We separate the strands (by denaturing the DNA) and we remove all lower strand sequences (for instance, we can pour the solution over a solid support having the upper strand of $\alpha$ bound to it; the lower strands will anneal and will remain bound to the solid support, the upper strands can be washed out). We make a copy of the tube. We do the same with the language $L_2$, in another test tube, preserving only the lower strands. We merge the two tubes and let the strands to anneal, then we remove all molecules which are not complete duplexes (having sticky ends or loops, hence nucleotides not paired with a complement); this can again be done by using solid support techniques as above. What remains is the intersection of the two languages. We melt again and remove the upper strands, then we merge the obtained solution with the copy of the first tube. After annealing, we remove all complete molecules and keep all upper strands which were not paired with their complementary lower strands. This is the difference $L_1 - L_2$ (if needed, by polymerisation/amplification we can return to double stranded molecules).

Of course, from a biochemical point of view, this is a completely idealized procedure (not to speak about the fact that we have dealt tacitly with infinite languages codified as DNA molecules and handled in test tubes, which, for a while are nothing else than finite objects...). However, this is the same kind of idealization as when passing from a computer to a Turing machine as a computing model, with an infinite tape, working an unbounded number of steps, etc. Important is that the difference and the complementation are "DNA-like operations".

In sum: we can produce $V^*$, we can produce[5] any given recursively enumerable language $L$ (for instance, using a controlled H system, see [14]), and we can compute the difference $V^* - L$. In this way, we can step outside the family of recursively enumerable languages !

The above described procedure can hardly be considered an algorithm, in any broad meaning of the term, hence we cannot infer that in this way we violate the Church-Turing Thesis. However, in some sense (because of the lack of precise definitions of the involved notions), this is a matter of terminology: what an algorithm is, what the Church-Turing Thesis says, when crossing the barrier of recursive enumerability means violating the Church-Turing Thesis ? We do not insist in this speculative direction, but we want to look for more precise definitions of the "computing by carving" in formal language theory terms.

[5]We agree with one of the referee who said that we will never be able to produce $V^*$ – or any other infinite language – by using DNA techniques. Because the universe is finite, everything is finite (including the test tubes)... The only trouble is that we cannot count on that in computability...

## 4 Identifying Languages by Complementation

Identifying a language $L$ by generating first its complement, $H = V^* - L$, and then computing the difference $V^* - H$ is quite a rough procedure. Let us consider a more subtle one, where $L$ is obtained at the end of several difference operations. The "most subtle" case is that when we have an infinite sequence of such operations: we construct $V^*$, as well as certain languages $L_1, L_2, \ldots$, and we iteratively compute $V^* - L_1$, $(V^* - L_1) - L_2, \ldots$, such that $L$ is obtained at the limit. In total, this means

$$L = V^* - (\bigcup_{i \geq 1} L_i).$$

We have returned again to the complement of only one language, the union $\bigcup_{i\geq 1} L_i$. Moreover, the languages $L_i, i \geq 1$, can be rather simple while their union can be arbitrarily complex. For example, take $L_i$ the $i$th string in the lexicographic order in a given language, which can be of any complexity in the Chomsky hierarchy. Each $L_i$ is a singleton, the union of these languages can be non-recursively enumerable.

The problem is still ill formulated. *The sequence of languages $L_i, i \geq 1$, must be defined in a regular, finite, way.* Here is a proposal:

A sequence $L_1, L_2, \ldots$ of languages over an alphabet $V$ is called *regular* if $L_1$ is a regular language and there is a gsm $g$ such that $L_{i+1} = g(L_i), i \geq 1$[6].

Thus, $L_1$ is given, all other languages are iteratively constructed by applying the gsm $g$ to $L_1$. In this way, we can identify the sequence by its generating pair $(L_1, g)$.

Then, a language $L \subseteq V^*$ is said to be *C-REG computable* if there is a regular sequence of languages, $L_1, L_2, \ldots$, and a regular language $M \subseteq V^*$ such that $L = M - (\bigcup_{i\geq 1} L_i)$.

Note that when defining C-REG computable languages we also allow the use of a specified initial (regular) language $M$, which is not necessarily $V^*$ as in the previous discussion.

We can denote by $g^i$ the $i$th iteration of $g, i \geq 0$, and then $L_{i+1} = g^i(L_1), i \geq 0$ (taking $g^0(H) = H$ for all $H \subseteq V^*$). Denoting by $g^*(L_1)$ the union $\bigcup_{i\geq 0} g^i(L_1)$, we can write $L = M - g^*(L_1)$. We are again back to the complement with respect to $M$ of a single language, $g^*(L_1)$, but this language is the union of a regular sequence of languages (the languages in the sequence are defined by finite tools: a regular grammar – or a finite automaton – for the language $L_1$, and the gsm $g$).

The regular sequences of languages deserve a more detailed study. We present here only a few properties of them.

First, it is easy to see that the union of a regular sequence of languages can be of a high complexity, in spite of the fact that each *finite* union is a regular language. For instance, it is easy to see that the following sequence is a regular one:

$$L_i = \{a^i b^i c^i\}, \ i \geq 0.$$

[6]Please distinguish between "a sequence of regular language" and "a regular sequence of languages", as defined here. Every regular sequence of languages is a sequence of regular languages, but the converse is not true.

The union is a non-context-free language.

Also the sequence $L_i = \{a^{2^i}\}$, $i \geq 1$, is regular: start ¿from $L_1 = \{a^2\}$ and take the gsm which doubles each occurrence of $a$.

However, not all languages can be written as the union of a regular sequence. We can find counterexamples by using the following necessary condition, whose proof is omitted:

**Lemma 1.** *If $(L_1, g)$ identifies an infinite language $L = g^*(L_1)$, then there is a constant $k$ such that for each $x \in L$ we can find $y \in L$ such that $|x| < |y| \leq k|x|$.*

For instance, the language

$$L = \{a^{2^{2^n}} \mid n \geq 1\}$$

does not have the property in Lemma 1, hence it cannot be obtained as the union of a regular sequence of languages.

On the other hand, the family of languages which can be obtained in this way is not at all a small one:

**Theorem 1.** *Every recursively enumerable language $L \subseteq T^*$ can be written in the form $L = g^*(\{a_0\}) \cap T^*$, where $g$ is a gsm – depending on $L$ – and $a_0$ is a fixed symbol not in $T$.*

Proofs of this result and of similar ones can be found in [12], [15], [20].

This result has an important consequence: because we have

$$V^* - (g^*(\{a_0\}) \cap T^*) = (V^* - g^*(\{a_0\})) \cup (V^* - T^*),$$

and $V^* - T^*$ is a regular language, it follows that $V^* - g^*(\{a_0\})$ is not necessarily a recursively enumerable language: if $V^* - (g^*(\{a\}) \cap T^*)$ is not recursively enumerable (and this can be the case, because $g^*(\{a_0\}) \cap T^*$ can be any recursively enumerable language – Theorem 1), then $V^* - g^*(\{a_0\})$ is not recursively enumerable either. This statement is worth formulating as a theorem:

**Theorem 2.** *There are C-REG computable languages which are not recursively enumerable languages.*

Two questions appear here in a natural way: (1) How powerful is the computing by carving (when using regular sequences of languages) ? (2) What about other definitions of regular sequences of languages, leading at the limit to simpler languages than those obtained in the case discussed above ?

In view of their connections with the DNA computing and the speculations related to the Church-Turing Thesis, these questions deserve a close investigation. The second question remains *open*. The first one is completely answered below.

**Theorem 3.** *A language is C-REG computable if and only if it can be written as the complement of a recursively enumerable language.*

*Proof.* Consider a language $L \subseteq T^*$ such that the language $T^* - L$ is recursively enumerable. Consider a type-0 Chomsky grammar $G = (N, T, S, P)$ for the language $T^* - L$.

We construct the regular sequence of languages starting with

$$L_1 = \{S\},$$

and using the gsm

$$g = (K, V, V, s_0, \{s_1\}, R),$$

with the following components:

$$\begin{aligned}
K &= \{s_0, s_1\} \cup \{[x] \mid x \in (N \cup T)^+, xy \to v \in P, \text{ for some } y \in (N \cup T)^+\},\\
V &= N \cup T,\\
R &= \{s_0\alpha \to \alpha s_0 \mid \alpha \in N \cup T\}\\
&\quad \cup \{s_0\alpha_1 \to [\alpha_1],\ [\alpha_1]\alpha_2 \to [\alpha_1\alpha_2], \ldots, [\alpha_1 \ldots \alpha_{i-2}]\alpha_{i-1} \to [\alpha_1 \ldots \alpha_{i-2}\alpha_{i-1}],\\
&\qquad [\alpha_1 \ldots \alpha_{i-1}]\alpha_i \to vs_1 \mid \alpha_j \in N \cup T, 1 \leq j \leq i, i \geq 2, \alpha_1 \ldots \alpha_i \to v \in P\}\\
&\quad \cup \{s_0\alpha \to vs_1 \mid \alpha \to v \in P, \alpha \in N\}\\
&\quad \cup \{s_1\alpha \to \alpha s_1 \mid \alpha \in N \cup T\}.
\end{aligned}$$

It is easy to see that $g^*(L_1) \cap T^* = L(G)$ (at each iteration of $g$ one simulates the application of a rule in $P$).

Therefore, for the regular language $M = T^*$ we obtain $M - g^*(L_1) = T^* - L(G) = T^* - (T^* - L) = L$. In conclusion, $L$ is a C-REG computable language.

Conversely, assume that $L \subseteq T^*$ is a C-REG computable language, hence $L$ can be written in the form $L = M - g^*(L_1)$, for some regular sequence of languages defined by $(L_1, g)$ and a regular language $M \subseteq T^*$. Let us consider the complement of $L$, that is $T^* - L$. We have

$$T^* - L = T^* - (M - g^*(L_1)) = (T^* - M) \cup g^*(L_1).$$

The language $T^* - M$ is regular (the family of regular languages is closed under difference) and $g^*(L_1)$ is recursively enumerable. Consequently, $T^* - L$ is a recursively enumerable language.

**Corollary 1.** (i) *Every context-sensitive language is C-REG computable.* (ii) *The family of C-REG computable languages is incomparable with the family of recursively enumerable languages.*

*Proof.* Assertion (i) follows from the fact that the family of context-sensitive languages is closed under the complementation, see [9], [19].

Assertion (ii) is a consequence of the non-closure of the family of recursively enumerable languages under the complementation.

We do not know whether or not Theorem 3 (and assertion (i) in Corollary 1) remains true for other, more restrictive, definitions of a regular sequence of languages. What about restricted forms of gsm mappings ? In particular, what about deterministic gsm's ? A related question is dealt with (and solved) in the following sections: does the number of states of the gsm used in the definition of regular sequences of languages induce an infinite hierarchy of C-REG computable languages ? (The answer will be negative.)

## 5 Iterated Finite State Sequential Transducers

Let us first introduce a general setting-up for our investigations, by defining the notion of an *iterated finite state sequential transducer* (in short, an IFT), as a language generating device. Among the reasons for considering this notion are the facts that, according to Theorem 1, in order to characterize $RE$ by iterated gsm's we can start from a single symbol (not from a regular language, as in the definition of regular sequences of languages), but we also need an intersection with a language $T^*$. Such an intersection can easily be done by a gsm (for instance, using a special state for this purpose), so we introduce this feature "inside" the gsm work. Because we explicitely provide the starting symbol $a_0$ in the machinery, we prefer to use a new terminology, speaking of IFT's and not of gsm's of a special form and with a special functioning.

An IFT is a construct $\gamma = (K, V, s_0, a_0, F, P)$, where $K, V$ are disjoint alphabets (the set of *states* and the *alphabet* of $\gamma$), $s_0 \in K$ (the *initial state*), $a_0 \in V$ (the *starting symbol*), $F \subseteq K$ (the set of *final states*), and $P$ is a finite set of *transition rules* of the form $sa \rightarrow xs'$, for $s, s' \in K, a \in V, x \in V^*$ (in state $s$, the device reads the symbol $a$, passes to state $s'$, and produces the string $x$).

For $s, s' \in K$ and $u, v, x \in V^*, a \in V$ we define

$$usav \vdash uxs'v \text{ iff } sa \rightarrow xs' \in P.$$

This is a *direct transition step* with respect to $\gamma$. We denote by $\vdash^*$ the reflexive and transitive closure of the relation $\vdash$.

Then, for $w, w' \in V^*$ we define

$$w \Longrightarrow w' \text{ iff } s_0 w \vdash^* w's, \text{ for some } s \in K.$$

We say that $w$ *derives* $w'$; note that this means that $w'$ is obtained by translating the string $w$, starting from the initial state of $\gamma$ and ending in any state of $\gamma$, not necessarily a final one. We denote by $\Longrightarrow^*$ the reflexive and transitive closure of the relation $\Longrightarrow$.

If in the writing above we have $s \in F$ (we stop in a final state), then we write $\stackrel{f}{\Longrightarrow}$ instead of $\Longrightarrow$: $w \stackrel{f}{\Longrightarrow} w'$ iff $s_0 w \vdash^* w's$, for some $s \in F$.

The *language generated* by $\gamma$ is

$$L(\gamma) = \{w \in V^* \mid a_0 \Longrightarrow^* w' \stackrel{f}{\Longrightarrow} w, \text{ for some } w' \in V^*\}.$$

Therefore, we iteratively translate the strings obtained by starting from $a_0$, without care about the states we reach at the end of each translation, but at the last step we necessarily stop in a final state.

The IFT's as defined above are *nondeterministic*. If for each pair $(s, a) \in K \times V$ there is at most one transition $sa \rightarrow xs'$ in $P$, then we say that $\gamma$ is *deterministic*.

We denote by $IFT_n, n \geq 1$, the family of languages of the form $L(\gamma)$, for nondeterministic $\gamma$ with at most $n$ states; when using deterministic IFT's we write $DIFT_n$ instead of $IFT_n$. The union of all families $IFT_n, n \geq 1$, is denoted by $IFT$; similarly for $DIFT$.

We are now going to compare these families with the families in the Chomsky hierarchy (especially $CF$ = context-free, $CS$ = context-sensitive, $RE$ = recursively enumerable), including the family of recursive languages, and with families in the Lindenmayer hierarchy, $0L$ and $E0L$ (of languages generated by interactionless L systems and by extended interactionless L systems, respectively); $D0L, DE0L$ are the families corresponding to $0L, E0L$ generated by deterministic L systems. We refer to [16] and [18] for details about these hierarchies.

**Convention.** When comparing the power of two generative devices, $G_1, G_2$, the empty string is ignored: $G_1$ is equivalent to $G_2$ if and only if $L(G_1) - \{\lambda\} = L(G_2) - \{\lambda\}$.

The following relations are direct consequences of the definitions:

**Lemma 2.** (i) $IFT_n \subseteq IFT_{n+1}, DIFT_n \subseteq DIFT_{n+1}$, $n \geq 1$. (ii) $DIFT_n \subseteq IFT_n$, $n \geq 1$, $DIFT \subseteq IFT$. (iii) $IFT \subseteq RE$.

Because IFT's with one state are equivalent to finite substitutions (the only state is both initial and final), while deterministic IFT's with only one state are equivalent to morphisms, we have:

**Lemma 3.** $IFT_1 = 0L,\ DIFT_1 = D0L$.

It is also easy to prove the next result:

**Lemma 4.** $E0L \subseteq IFT_2$.

The hierarchy $IFT_n, n \geq 1$, is not an infinite one. This fact is not unexpected: we know that iterated gsm's characterize the family $RE$, which can be stated as $RE \subseteq IFT$; start the proof of this inclusion from a *universal type-0 grammar* (such a grammar is constructed, for instance, in [4]); in this way we obtain an IFT with a fixed number of states; by slightly modifying it, we can obtain an IFT generating any given recursively enumerable language (at the first step of the work of our IFT we introduce the code of the particular grammar which is used by the universal one in order to simulate the particular grammar; then, our IFT works as the universal grammar, hence it generates the language of the particular grammar). This argument only shows that the hierarchy $IFT_n, n \geq 1$, collapses. Using the construction in [4], we get a huge number of states in our "universal" IFT. However, this hierarchy collapses at a surprisingly low number of levels: four.

**Lemma 5.** $RE = IFT_4$.

*Proof.* We only have to prove the inclusion $RE \subseteq IFT_4$.

Consider a language $L \in RE, L \subseteq T^*$, and take a grammar $G = (N, T, S, P)$ in the Geffert normal form generating it, [7]. That is, we have $N = \{S, A, B, C\}$ and the rules in $P$ are of the following forms: $S \to x$, for $x \in (N \cup T)^+$, and $ABC \to \lambda$. (Note that there are exactly four nonterminals, all context-free rules have $S$ as their left hand member, and that there is only one non-context-free rule, the erasing rule $ABC \to \lambda$.)

Consider two new symbols, $a_0, a_1$, and construct the IFT

$$\gamma = (\{s_0, s_1, s_A, s_B\}, V \cup \{a_0, a_1\}, s_0, a_0, \{s_1\}, P'),$$

where

$$\begin{aligned}
P' = & \{s_0a_0 \to a_1Ss_0\} \\
& \cup \{s_0\alpha \to \alpha s_0 \mid \alpha \in N \cup T \cup \{a_1\}\} \\
& \cup \{s_0S \to xs_0 \mid S \to x \in P\} \\
& \cup \{s_0A \to s_A,\ s_AB \to s_B,\ s_BC \to s_0\} \\
& \cup \{s_0a_1 \to s_1\} \\
& \cup \{s_1a \to as_1 \mid a \in T\}.
\end{aligned}$$

At the first step we introduce the axiom of $G$, marked to the left with the symbol $a_1$. At the subsequent translations, the context-free rules $S \to x$ of $P$ are simulated by using only the state $s_0$. The deletion of a substring $ABC$ is performed as follows:

$$s_0x_1ABCx_2 \vdash^* x_1s_0ABCx_2 \vdash x_1s_ABCx_2 \vdash x_1s_BCx_2 \vdash x_1s_0x_2 \vdash^* x_1x_2s_0.$$

Of course, none or several applications of rules in $P$, irrespective whether or not they are context-free or non-context-free, can be simulated at the same translation, but this does not change the generated language.

We can reach the final state $s_1$ only by erasing the symbol $a_1$ by a step $s_0a_1w \vdash s_1w$; then, only terminal symbols can be parsed, hence the string $w$ obtained in this way is a terminal one. After erasing the symbol $a_1$, the final state cannot be reached once again, hence new rules in $P$ cannot be simulated in a successful way.

Consequently, $L(\gamma) = L(G)$, that is $RE \subseteq IFT_4$.

As a particular case, the construction above gives a new proof of the inclusion $CF \subseteq IFT_2$, which also follows from Lemma 4, because $CF \subset E0L$ (if the rule $ABC \to \lambda$ is missing, then we simulate a context-free grammar and the states $s_A, s_B$ are no longer necessary).

We do not know whether or not the result above is optimal. Anyway, IFT's with three states are very powerful:

**Lemma 6.** *The family $IFT_3$ contains non-recursive languages.*

*Proof.* Consider two $n$-tuples of non-empty strings, $x = (x_1, \ldots, x_n), y = (y_1, \ldots, y_n)$, over the alphabet $\{a, b\}$. Let $a', b'$ be two new symbols associated with $a, b$, respectively. Denote by $h$ the coding defined by $h(a) = a', h(b) = b'$, and by $mi(x)$ the mirror image of $x$, for any string $x$. We construct the IFT

$$\gamma = (\{s_0, s_1, s_2\}, \{a, b, a', b', a_0, c, d\}, s_0, a_0, \{s_0\}, P),$$

with the following rules:

1. $s_0a_0 \to cx_id\ mi(h(y_i))cs_0$, for $1 \leq i \leq n$,

2. $s_0\alpha \to \alpha s_0$, for $\alpha \in \{a, b, a', b', c\}$,

3. $s_0 d \rightarrow x_i d\ mi(h(y_i))s_0$, for $1 \leq i \leq n$,

4. $s_0 d \rightarrow s_0$,

5. $s_0 a \rightarrow s_1,\ s_1 a' \rightarrow s_0$,

6. $s_0 b \rightarrow s_2,\ s_2 b' \rightarrow s_0$.

It is easy to see that after replacing $a_0$ at the first step with a string $cx_i d\ mi(h(y_i))c$, for some $1 \leq i \leq n$, at the next iterations one replaces the occurrence of $d$ by a string of the form $x_j d\ mi(h(y_j))$, for $1 \leq j \leq n$. In this way, we obtain all strings of the form $cx_{i_1}x_{i_2}\ldots x_{i_r}d\ mi(h(y_{i_r}))\ldots mi(h(y_{i_2}))\ mi(h(y_{i_1}))c$, for some $r \geq 1, 1 \leq i_j \leq n, 1 \leq j \leq r$.

Then, after removing the symbol $d$, by the rule of type 4, the following iterations either leave the string unchanged (by using rules of type 2), or a pair $aa'$ or $bb'$ is deleted, via states $s_1, s_2$, respectively (by using rules of types 5 and 6). If we arrive in $s_1$ (after erasing an occurrence of the symbol $a$) and the next symbol is not $a'$, then the work of $\gamma$ is blocked (at least the rightmost occurrence of the symbol $c$ has remained non-scanned). The same happens if we arrive in $s_2$ (after deleting an occurrence of the symbol $b$) and the next symbol is not $b'$.

Therefore, we can erase all symbols in between the two occurrences of the marker $c$ if and only if $x_{i_1}x_{i_2}\ldots x_{i_r} = h^{-1}(mi(mi(h(y_{i_r}))\ldots mi(h(y_{i_2}))\ mi(h(y_{i_1}))))$. Because $mi(mi(h(y_{i_r}))\ldots mi(h(y_{i_2}))\ mi(h(y_{i_1}))) = h(y_{i_1})h(y_{i_2})\ldots h(y_{i_r}) = h(y_{i_1}y_{i_2}\ldots y_{i_r})$, and $h$ is one-to-one, this means that $x_{i_1}x_{i_2}\ldots x_{i_r} = y_{i_1}y_{i_2}\ldots y_{i_r}$, that is, the Post Correspondence Problem for the $n$-tuples $x, y$ (abbreviated, $PCP(x,y)$) has a solution. Consequently, the string $cc$ belongs to the language $L(\gamma)$ if and only if $PCP(x,y)$ has a solution. This is undecidable, hence the language $L(\gamma)$ is not recursive.

Synthesizing the results above, and using the known relations among Chomsky and Lindenmayer families, we obtain:

**Theorem 4.** (i) $IFT_1 = 0L \subset E0L \subseteq IFT_2 \subseteq IFT_3 \subseteq IFT_4 = RE$. (ii) $CF \subset IFT_2$. (iii) $IFT_3 - CS \neq \emptyset$.

We do not know which of the non-proper inclusions in this theorem are proper.

Also the relationships between the family $CS$ and the families $IFT_2, IFT_3$ are open. According to the following result, if $CS \subseteq IFT_2$, then $RE \subseteq IFT_3$ (which however is not very plausible).

**Theorem 5.** *If $CS \subseteq IFT_n$, for some $n \geq 1$, then $RE \subseteq IFT_{n+1}$.*

*Proof.* Consider a recursively enumerable language $L \subseteq T^*$. Take a type-0 grammar $G = (N, T, S, P)$ generating $L$ and a new symbol, $c$. We construct the length-increasing grammar

$$G' = (N, T \cup \{c\}, S, P'),$$

with

$$\begin{aligned} P' &= \{u \rightarrow v \mid u \rightarrow v \in P, |u| \leq |v|\} \\ &\cup \{u \rightarrow vc^s \mid u \rightarrow v \in P, |u| = |v| + s, s \geq 1\} \\ &\cup \{c\alpha \rightarrow \alpha c,\ \alpha c \rightarrow c\alpha \mid \alpha \in N \cup T\}. \end{aligned}$$

It is easy to see that $L(G') \subseteq L \text{ Ш } \{c\}^*$ and that for each string $w \in L$ there is a string $w' \in L(G'), w' \in \{w\} \text{ Ш } \{c\}^*$. (We have denoted by Ш the shuffle operation, defined by $x \text{ Ш } y = \{x_1y_1x_2y_2 \dots x_ny_n \mid x = x_1x_2 \dots x_n, y = y_1y_2 \dots y_n, n \geq 1, x_i, y_i \in V^*, 1 \leq i \leq n\}$, for $x, y \in V^*$, and extended in the natural way to languages.) Indeed, the length-decreasing rules of $P$ are replaced by monotone rules which introduce occurrences of the symbol $c$; this symbol is moved freely to right and to left, making room for left hand sides of rules in $P$.

Consider now an IFT $\gamma = (K, V, s_0, a_0, F, P'')$ generating the context-sensitive language $L(G')$ and having $card(K) = n$. Let $a_0', c_1, c_2, c_3$ be new symbols and $s_1$ a new state, and construct the IFT

$$\gamma' = (K \cup \{s_1\}, V \cup \{a_0', c_1, c_2, c_3\}, s_0, a_0', \{s_1\}, P'''),$$

where $P'''$ contains all transition rules in $P''$ as well as the following transitions:

1. $s_0a_0' \to c_1xc_2s$, for all $s_0a_0 \to xs \in P''$ with $x \in V^*, s \in K$,

2. $s_0c_1 \to c_1s_0$,
   $sc_2 \to c_2s$, for all $s \in K$,

3. $sc_2 \to c_3s$, for all $s \in F$,

4. $s_0c_1 \to s_1$,
   $s_1c \to s_1$,
   $s_1c_3 \to s_1$,
   $s_1a \to as_1$, for all $a \in T$.

After replacing $a_0'$ by a string introduced by $\gamma$ when scanning $a_0$, bounded by the markers $c_1, c_2$, the IFT $\gamma'$ simulates the work of $\gamma$, modulo the passing over $c_1$ and $c_2$ and leaving them unchanged. The symbol $c_3$ cannot be scanned in a state of $K$. This symbol can be introduced only from a state in $F$, by a rule $sc_2 \to c_3s, s \in F$. If such a rule is used, then the obtained string, of the form $c_1wc_3$, cannot be further translated by the rules in $P''$. This means that a rule of the form $sc_2 \to c_3s, s \in F$, is used only once, at the end of simulating in $\gamma'$ a sequence of translations in $\gamma$.

On the other hand, the symbol $c_2$ cannot be scanned in the state $s_1$, hence the work of $\gamma'$ cannot be finished as long as the symbol $c_2$ is present in the string. Thus, a rule $sc_2 \to c_3s, s \in F$, must be used. This ensures the fact that the work of $\gamma$ ends in a final state, as imposed by $F$ (note that the states in $F$ are no longer final states in $\gamma'$).

A string of the form $c_1wc_3$ can be processed on the path from $s_0$ to $s_1$; all symbols $c_1, c_3, c$ are erased, all symbols in $T$ are left unchanged. This is the only way of reaching the end of a string in the final state $s_1$. Therefore, if $w$ above is a string from $L(G')$ such that $w \in \{z\} \text{ Ш } \{c\}^*$, for $z \in L$, then the string generated by $\gamma'$ is exactly $z$. In conclusion, $L(\gamma') = L$, that is, $L \in IFT_{n+1}$.

Some indication on the size of the family $IFT_3$ is also shed by the following result, whose proof is omitted and which shows that this family includes the largest family of L languages generated without interaction, $ET0L$ (the family of languages generated by extended tabled interactionless L systems). It is known that this family is strictly included in $CS$.

**Theorem 6.** $ET0L \subseteq IFT_3$.

## 6 Back to Computing by Carving

The main result of the previous section is the fact that non-deterministic iterated finite state sequential transducers with four states characterize the recursively enumerable languages. This implies that the number of states does not induce an infinite hierarchy of languages which are C-REG computable: in view of Theorem 3, a language is C-REG computable if and only if it is the complement of a language in $RE = IFT_4$. Thus, if we take into account the number of states of an IFT generating the complement of a C-REG computable language, then we can have at most four levels in the hierarchy of C-REG computable languages. If we take into account the number of states of the gsm $g$ describing a regular sequence of languages, identified by a pair $(L_1, g)$, then the hierarchy is still lower: there are at most three levels. To this aim, we will repeat the proof of one implication in Theorem 3, making use of the basic idea used in the proof of Lemma 5.

Let us denote by $CREG_n, n \geq 1$, the family of languages of the form $M - g^*(L_1)$, for $M, L_1$ regular languages and $g$ a gsm with at most $n$ states; by $CREG$ we denote the union of all these families, that is, the family of all C-REG computable languages.

**Theorem 7.** $CREG_1 \subset CREG_2 \subseteq CREG_3 = CREG$.

*Proof.* The inclusions $\subseteq$ are obvious. According to Theorem 3, each language $L \subseteq T^*$, $L \in CREG$, has the complement in $RE$. Take such a language $L$ and consider a grammar $G = (N, T, S, P)$ for the language $T^* - L$ in the Geffert normal form, that is with $N = \{S, A, B, C\}$ and with $P$ containing context-free rules $S \to x, x \in (N \cup T)^*$, and the non-context-free rule $ABC \to \lambda$.

Consider $L_1 = \{S\}$ and the gsm

$$g = (K, V, V, s_0, \{s_0\}, R),$$

with the following components:

$$\begin{aligned} K &= \{s_0, s_A, s_B\} \\ V &= N \cup T, \\ R &= \{s_0\alpha \to \alpha s_0 \mid \alpha \in N \cup T\} \\ &\quad \cup \{s_0 S \to x s_0 \mid S \to x \in P\} \\ &\quad \cup \{s_0 A \to s_A,\ s_A B \to s_B,\ s_B C \to s_0\}. \end{aligned}$$

It is easy to see that $g^*(L_1) \cap T^* = L(G)$ (at each iteration of $g$ one can simulate the application of a rule in $P$; if several rules are simulated at the same iteration, then this does not change the generated language). We proceed now as in the proof of Theorem 3: we take $M = T^*$ and we obtain $M - g^*(L_1) = L$. In conclusion, $L \in CREG_3$.

The inclusion $CREG_1 \subset CREG_2$ is clearly proper, because by iterating a gsm with only one state we get a language in $IFT_1 = 0L$.

The proof before does not imply that $RE = IFT_3$, because one state has been saved here by using the language $M$, of a specific form, which makes non-necessary the use of a special (final) state in order to check whether or not the string produced by iterating the gsm is terminal.

The previous theorem shows that in order to classify the C-REG computable languages we need other parameters describing the complexity of the regular sequences of languages, the number of states is not sufficiently sensitive. Such more sensitive parameters remain to be found.

Also the case of deterministic iterated transducers remains to be investigated.

An important research problem here is the "applicative" one: Are there (mathematical or practical) problems which can be solved by computing by carving in an easier way than by "classic" methods ? Can such solutions be implemented in a biochemically feasible and computationally efficient way ?

At least from a theoretical point of view, it seems that the computing by carving is useful: as it was mentioned to us by G. M. Germano (Pisa University) at the beginning of April 1998, by carving it is rather easy to construct computable unary functions, a task by no means trivial when using classic methods, see, e.g, [6] and its references.

**Acknowledgement.** This work has been supported by CNR, Gruppo Nazionale per l'Informatica Matematica, Italy, and by the Direcció General de Recerca, Generalitat de Catalunya (PIV)

## References

[1] L. M. Adleman, Molecular computation of solutions to combinatorial problems, *Science*, 226 (1994), 1021 – 1024.

[2] M. Amos, *DNA Computation*, PhD Thesis, Univ. of Warwick, Dept. of Computer Sci., 1997.

[3] M. Amos, A. Gibbons, D. Hodgson, Error-resistant implementation of DNA computations, in *Proc. of the Second Annual Meeting on DNA Based Computers*, Princeton, 1996.

[4] C. Calude, Gh. Păun, Global syntax and semantics for recursively enumerable languages, *Fundamenta Informaticae*, 4, 2 (1981), 245 – 254.

[5] T. Head, Hamiltonian paths and double stranded DNA, in *Computing with Bio-Molecules. Theory and Experiments* (Gh. Păun, ed.), Springer-Verlag, Singapore, 1998 (to appear).

[6] G. M. Germano, S. Mazzanti, General iteration and unary functions, *Annals of Pure and Applied Logic*, 54 (1991), 137 – 178.

[7] V. Geffert, Normal forms for phrase-structure grammars, *RAIRO. Th. Inform. and Appl.*, 25 (1991), 473 – 496.

[8] F. Guarnieri, M. Fliss, C. Bancroft, Making DNA add, *Science*, 273 (1996), 220 – 223.

[9] N. Immerman, Nondeterministic space is closed under complementation, *SIAM J. Comput.*, 17, 5 (1988), 935 – 938.

[10] R. J. Lipton, DNA solution of hard computational problems, *Science*, 268 (1995), 542 – 545.

[11] Q. Ouyang, P. D. Kaplan, S. Liu, A. Libchaber, DNA solution of the maximal clique problem, *Science*, 278 (1997), 446 – 449.

[12] Gh. Păun, On the iteration of gsm mappings, *Revue Roum. Math. Pures Appl.*, 23, 4 (1978), 921 – 937.

[13] Gh. Păun, *(DNA) Computing by carving*, Technical Report TCS-97-17, Center for Theoretical Study at Charles Univ. and the Academy of Sciences of the Czech Republic, Prague, 1997.

[14] Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Heidelberg, 1998 (to appear).

[15] B. Rovan, A framework for studying grammars, *Proc. MFCS 81, Lect. Notes in Computer Sci.* 118, Springer-Verlag, 1981, 473 – 482.

[16] G. Rozenberg, A. Salomaa, *The Mathematical Theory of L Systems*, Academic Press, New York, 1980.

[17] G. Rozenberg, A. Salomaa, Watson-Crick complementarity, universal computations and genetic engineering, *Techn. Report* 96-28, Dept. of Computer Science, Leiden Univ., October 1996.

[18] G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, 3 volumes, Springer-Verlag, Heidelberg, 1997.

[19] R. Szelepcsenyi, The method of forced enumeration for nondeterministic automata, *Acta Inform.*, 26, 3 (1988), 279 – 284.

[20] D. Wood, Iterated a-NGSM maps and Γ-systems, *Inform. Control*, 32 (1976), 1 – 26.