Accomplishments and New Directions for 2004:

Progress on Universally Programmable Intelligent Matter

UPIM Report 10

Technical Report UT-CS-04-352

Bruce J. MacLennan*

Department of Computer Science
University of Tennessee, Knoxville
www.cs.utk.edu/~mclennan

October 15, 2004

Abstract

We summarize progress to date on "universally programmable intelligence matter" (the use of general-purpose autonomous molecular computation [AMC] for nanostructure synthesis and control), and on this basis discuss directions for future research. Accomplishments include an analysis of extensions to combinatory logic required for nanotechnology and an investigation (through simulation) of the assembly of a variety of membranes, nanotubes, pores, channels, actuators, and other important structures. We also outline the requirements and possible solutions for molecular implementation of this computational model. In addition to discussing the successes and limitations of molecular combinatory computing, the approach that has been investigated to date, we also suggest an alternative approach to AMC that may be more suitable for nanotechnology due to its greater insensitivity to error and stochastic effects.

*This research is supported by Nanoscale Exploratory Research grant CCR-0210094 from the National Science Foundation. It has been facilitated by a grant from the University of Tennessee, Knoxville, Center for Information Technology Research.

1 Introduction

Although nanotechnology is advancing rapidly, many of the techniques are ad hoc in that they are limited to small classes of materials or applications. In some ways this is comparable to the early days of computing technology, when special-purpose computers were designed individually for particular applications. Computer technology began to advance much more quickly after the design of the general-purpose computer, which provided one device that could be easily programmed for a variety of applications. Our goal has been to use a similar idea to accelerate the development of nanotechnology. In addition to the current ad hoc design of materials, we are investigating the development of a technology for autonomous molecular computation that can be applied to the synthesis of nanostructures and to the control of nanoscale behavior.

We can put this idea more generally. In the broadest terms, computation is the use of physical systems to implement information representation and processing [Mac04b]. By choice (or design) of an appropriate physical system (e.g., a general-purpose electronic digital computer), we can so arrange it that the sequence of physical states of the system are determined by rules (i.e., an algorithm) that we have defined and that are represented in the system's state (i.e., a stored program). Ordinarily in computation we are more interested in the abstract information processing than in its physical representation, but we can also view computation in the opposite way, as a means for convenient, general control of the trajectory of a system's physical states. This observation is the basis of our approach: to use autonomous molecular computation as a means of nanostructure assembly and as a way to control nanoscale behavior. By use of an appropriate molecular computing medium, our goal is that we may program the assembly of a nanostructure much as we program the assembly of a data structure (which must be represented, it should be recalled, in a physical medium), and that we may program the nanoscale behavior of a material much as we program the behavior of a digital electronic device.

2 Background

Some forms of molecular computation require the external manipulation of environmental conditions such as temperature (e.g., to control stability of hydrogen bonds, as in denaturation and renaturation of DNA; to control activity of enzymes), light [KZvD$^+$99], and chemical concentrations (e.g., enzymes, ATP, DNA "fuel") [KdSS99, YTM$^+$00]. The pioneering work of Adleman [Adl94] and Lipton [Lip95] may be cited as examples. In contrast, in *autonomous* molecular computation (AMC), the computational process proceeds without external control. This is sometimes called "one pot" molecular computation: you put the reactants in the pot, provide proper (constant) environmental conditions (which might include continuous provision of fuel and elimination of waste), and the computational chemical reaction proceeds on its

own, generally to an equilibrium representing the result of the computation.

For application to nanostructure synthesis and control, it is essential that molecular computation be autonomous. The assembly of significant nanostructures may require very large numbers of individual computational operations (10^6, 10^9, or many more: consider the assembly of a 1 cm^2 membrane from 10^{13} molecular building blocks each 10 nm^2 in area), far too many steps to control externally. Rather, a principal advantage of molecular computation is that extremely large numbers of operations may proceed asynchronously and in parallel.

Many nanotechnology applications of autonomous molecular computation are *static*, by which we mean that the computation will assemble some nanostructure and then become inactive. In some respects this is similar to the work that has been done already on DNA self-assembly of nanostructures [Bas03, RLS01, WLWS98]. However, in all of these demonstrations, so far as we are aware, the computation has been *monotonic*, that is, as the computation proceeds towards termination, it is always adding to the structure, never deleting or replacing elements. Use of a more general model of autonomous molecular computation will permit modification of the entire structure throughout the computational process, much as a data structure is modified by an ordinary program. This will facilitate the synthesis of many structures for which a monotonic assembly is inefficient or impossible. Simple static applications of autonomous molecular computation include synthesis of membranes (of a desired architecture, possibly including pores), nanotubes, complex layout patterns (determined by a program), and three-dimensional structures such as sponges.

Other nanotechnology applications of autonomous molecular computation are *dynamic*, which means that they remain responsive or reactive in their operational environment. In effect, the computation proceeds to a temporary equilibrium state, but that equilibrium can be disrupted by changes in environmental conditions, which cause the material to compute some more, with further changes to the physical state of the system, until a new (temporary) equilibrium is reached. A simple example is a channel in a membrane, which can open or close depending on some triggering condition (presence of a chemical species of interest, electromagnetic radiation, etc.). Another example is a nanoactuator, which can retract or extend under computational control.

The proceeding applications of autonomous molecular computation can be classified as either assembly of nanostructures (static applications) or control of nanoscale behavior (dynamic applications), although the latter will involve some assembly as well. While these nanotechnology applications are the primary motivation for our research, we should not forget that autonomous molecular computation may be applied to more general nanocomputation, such as the massively parallel solution of NP-complete problems. This will be a great improvement over previous approaches, because it affords the ability to translate arbitrary computer programs to the molecular level, where they can execute with very high ("molar") degrees of parallelism. Nevertheless, nanocomputation per se is not the primary application focus of this

project.

We anticipate that the use of autonomous molecular computation in a nanotechnology application will involve the following steps:

1. expression of the desired nanostructure assembly or nanoscale behavior in terms of a conventional program in a high-level language, which is tested and debugged on a conventional computer in the ordinary way;
2. translation of high-level program into descriptions of molecular structures;
3. further testing and debugging of the (simulated) molecular program on a molecular reaction simulator, to evaluate the effects of errors and other factors peculiar to autonomous molecular computation;
4. synthesis of corresponding physical molecular structures from the description;
5. replication (e.g., by PCR) to achieve required concentrations;
6. supplying reaction resources ("fuel") to allow the computation to proceed;
7. allowing the reaction to reach equilibrium;
8. for a static application, the equilibrium state contains the desired nanostructure, represented in terms of the computational molecules;
9. if required by the application, computational groups may be replaced chemically by other chemical groups (e.g., metal complexes);
10. for a dynamic application, the equilibrium state is temporary, and the resulting (reactive) material may be deployed in the operational environment;
11. such an active material will be responsive whenever computational reaction resources are provided to it.

Some of these steps will become clearer is the context of molecular combinatory computing, discussed later (Sec. 2.1).

Many of the existing approaches to autonomous molecular computation have looked to the highly successful von Neumann model of computation. These approaches have focused on the binary representation of data and have proceeded bottom-up, beginning with the implementation of logic gates and simple arithmetic operations [LWR00, MLRS00, Rei02, WYS98, Win96]. These investigations are valuable for demonstrating how molecular processes (especially based on DNA) can be applied to computational ends, but we believe that it is necessary to broaden the investigation beyond the von Neumann model (which is oriented toward electronic switches) to other models of computation, which are more compatible with molecular processes.

Certainly, there has been some exploration of alternative models of computation for autonomous molecular computation. For example a book by Păun, Rozenberg, and Salomaa [PRS98] presents a number of theoretical results showing that various string transformation systems, theoretically implementable in DNA, are computationally universal (equivalent to a universal Turing machine in computational power); and there is also Winfree's research [WYS98, Win96]. This work is certainly important, but it does not seem to be directly applicable to nanostructure assembly (except for DNA strings) or to control of the nanobehavior of materials; that is, it is more applicable to nanocomputation. We believe that the needs of nanostructure synthesis and control are better achieved by graph-based models of computation, which may be implementable by substitution reactions in molecular networks. (Our own exploratory research in this direction is described in more detail below, Secs. 2.1 and 3.) So far as we can determine, none of these investigations (including our own) have dealt adequately with the problem of errors in the molecular processes implementing computations.

2.1 Molecular Combinatory Computing (MCC)

We are investigating a systematic approach to nanotechnology based on a small number of molecular building blocks (MBBs). To accomplish this we have used *combinatory logic* [CFC58], a mathematical formalism based on network (graph) substitution operations suggestive of supramolecular interactions. This theory shows that two simple substitution operations (known as S and K) are sufficient to describe any computable process (Turing-computable function) [CFC58, sec. 5H]. Therefore, these two operations are, in principle, sufficient to describe any process of nanoscale synthesis or control that could be described by a computer program. In a molecular context, several additional housekeeping operations are required beyond S and K, but the total is still less than a dozen.

An additional advantage of the combinatory approach is that computer scientists have known for decades how to compile ordinary programs into combinator programs, and so this approach offers the prospect of compiling computer programs into molecular structures so that they may execute at the molecular level and with "molar" degrees of parallelism. Further, the *Church-Rosser Theorem* [CFC58, ch. 4] proves that substitutions may be performed in any order or in parallel without affecting the computational result; this is very advantageous for molecular computation. (More precisely, the theorem states that *if* you get a result, you always get the same result. Some orders, however, may lead to nonterminating computations that produce *no* result. To date, we have found little need in molecular computing for such potentially nonterminating programs.)

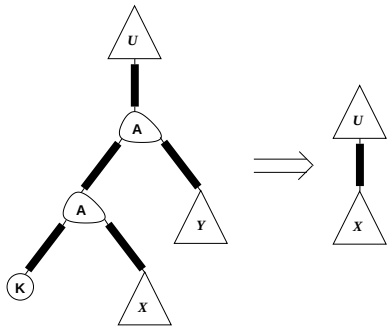

Figure 1: K combinator substitution operation. U, X, and Y represent any networks (graphs).

2.1.1 Molecular Combinatory Primitives

Molecular combinatory programs are supramolecular structures in the form of binary trees. The interior nodes of the tree (which we call A nodes) represent the application of a function to its argument, the function and its argument being represented by the two daughters of the A node. The leaf nodes are molecular groups that represent primitive operations. As previously remarked, one of the remarkable theorems of combinatory logic is that two simple substitution operations are sufficient for implementing any program (Turing-computable function). Therefore we use primarily the two primitive combinators, K and S (which exists in two variants Š and S proper).

To understand these operations, consider a binary tree of the form $((\mathsf{K}X)Y)$, where X and Y are binary trees (Fig. 1, left-hand side). (The A nodes are implicit in the parentheses.) This configuration triggers a substitution reaction, which has the effect:

$$((\mathsf{K}X)Y) \Longrightarrow X. \tag{1}$$

That is, the complex $((\mathsf{K}X)Y)$ is replaced by X in the supramolecular network structure; the effect of the operation is to delete Y from the network. The K group is released as a waste product, which may be recycled in later reactions. The tree Y is also a waste product, which may be bound to another primitive operator (D), which disassembles the tree so that its components may be recycled. The D primitive is the first of several house-keeping operations, which are not needed in the theory of combinatory logic, but are required for molecular computation. (Detailed descriptions can be found in a prior report [Mac02c].)

The second primitive operation is described by the rule:

$$(((\mathsf{S}X)Y)Z) \Longrightarrow ((XZ)(YZ)). \tag{2}$$

This rule may be interpreted in two ways, either as copying the subtree Z or as creating two links to a shared copy of Z (thus creating a graph that is not a tree). It can be proved that both interpretations produce the same computational result, but they have different effects when used for nanostructure assembly. For this reason we

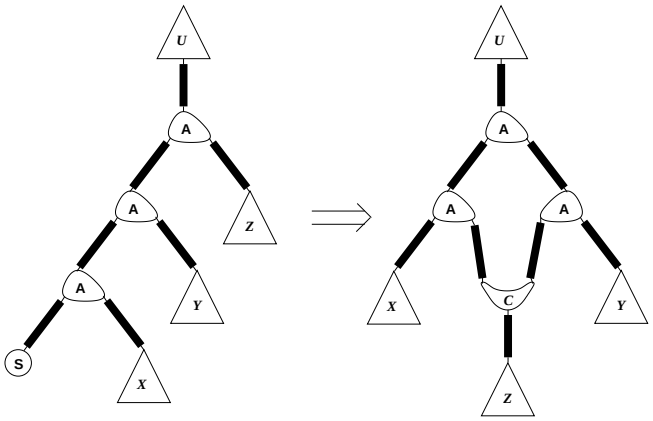

Figure 2: S and Š combinator substitution operations. $C = \mathsf{R}$ for S and $C = \mathsf{V}$ for Š. Note the reversed orientation of the rightmost A group.

need both variants of the operation, which we denote S (replicating) and Š (sharing). The molecular implementations of the two are very similar (see Fig. 2).

If $C = \mathsf{R}$ (a *replication node*), then other substitution reactions will begin the replication of Z, so that eventually the two links will go to two independent copies of Z:

$$(((\mathsf{S}X)Y)Z) \Longrightarrow ((XZ)(YZ')). \tag{3}$$

Here Z' refers to a new copy of the structure Z, which is created by a primitive replication (R) operation. The R operation progressively duplicates Z, "unzipping" the original and new copies [Mac02c]. The properties of combinatory computing allow this replication to take place while other computation proceeds, even including use of the partially completed replicate (a consequence of the Church-Rosser theorem).

The Š variant of the S operation is essential to molecular synthesis [Mac02c]. Thus, if $C = \mathsf{V}$ (a *sharing node*), then we have two links to a shared copy of Z (Fig. 2):

$$(((\check{\mathsf{S}}X)Y)Z) \Longrightarrow ((XZ^{(1)})(YZ^{(0)})). \tag{4}$$

The structure created by this operator shares a single copy of Z; the notations $Z^{(1)}$ and $Z^{(0)}$ refer to two links to a "Y-connector" (the V node), which links to the original copy of Z. (Subsequent computations may rearrange the locations of the two links.) The principal purpose of the Š operation is to synthesize non-tree-structured supramolecular networks; examples of its use are given in Sec. 3.1.2.

3 Progress to Date

In this section we describe the results of our NSF Nanoscale Exploratory Research grant.

3.1 MCC Applied to Nanotechnology

Before presenting examples of molecular combinatory computing applied to nanostructure synthesis and control, we will mention some of the preliminary investigations undertaken as a part of this project.

3.1.1 Preliminary Investigations

House-keeping Species: Although the S and K combinators and the binary interior nodes that connect them are all that is required for abstract combinatory computing, several additional nodes and substitutions are needed in a molecular context.

In addition to leaf nodes (S, K, etc.) and interior nodes (A, V, etc.), the edges between nodes must be represented by explicit *link groups*, which are asymmetric (since edges are directed) [Mac02c]. The length of these groups (distance between binding sites) determines the fundamental length scale of many nanostructures (see Sec. 3.1.2).

Since unused groups of various sorts must be available in solution in order to be able to assemble new networks, we determined that unused binding sites on these groups should be "capped" with inert groups, until the unused groups are required [Mac02c]. However, this problem might be solved a different way with the possible molecular implementation that we have investigated (Sec. 3.2 below).

Reaction waste products must be recycled or eliminated from the system, for several reasons. An obvious one is efficiency (decreasing resource utilization); another is to avoid the reaction space becoming clogged with waste. Less obvious is the fact that discarded molecular networks (such as Y in Eq. 1) may contain large executable structures; by the laws of combinatory logic, computation in these discarded networks cannot affect the computational result, but they can consume resources. Our approach is to attach a D group to a waste structure; the reactions associated with this group, which we have defined [Mac02c], lead to the structure's recursive disassembly; properly "capped" elementary nodes are released into solution for possible reuse.

As mentioned above (Sec. 2.1), we have developed "lazy replication" reactions, which allow replication of a structure, while, in parallel, both replicates can begin to be used [Mac02c]. This kind of asynchronous, parallel computation illustrates the value of combinatory logic as a basis for autonomous molecular computation.

Combinatory logic is traditionally defined in terms of trees, but for general nanostructure synthesis, combinatory programs need to be able to assemble arbitrary cyclic graphs (molecular networks). However, we cannot simply have unlimited pointers (links) to a shared structure, as we would on an ordinary computer; in a molecular context, there are a fixed number of binding sites on a group. Therefore we have investigated the use of a "sharing node" (called V), which often functions structurally as a complement of the A nodes [Mac02c]. (Recall that edges are directed. A has in-degree 1 and out-degree 2, whereas V has in-degree 2, but out-degree 1.) While

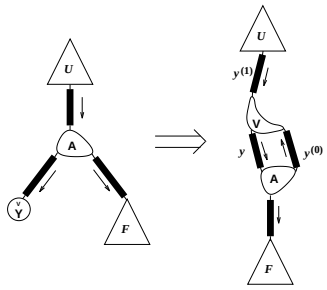

Figure 3: $\check{\mathsf{Y}}$ combinator primitive substitution operation. Arrows indicate link direction; note elementary cycle between A and V groups.

this approach has worked reasonably well, it is not without problems. For example, the V node is computationally inert and interferes with the recognition of the computational substitutions (K, S, etc.), so once such a node has been introduced, it may be difficult to manipulate (see Sec. 4.2.2 below). To date, the $\check{\mathsf{S}}$ operation has been adequate for defining non-cyclic, non-tree graphs, but its sufficiency has not been proved yet (see Sec. 4.2.2 below).

In the context of abstract tree substitution systems, there is no difference between a cyclic structure and a potentially infinite self-embedding tree. These can be constructed by a combinator called Y, which is definable in terms of S and K [CFC58, p. 178]. In implementations on conventional computers, this definition of Y can result in the creation of large trees, so it may be recognized as a special case and interpreted to create a cyclic structure [Mac90, pp. 525–535]. We have adapted this idea to molecular combinatory computing by defining a new primitive operation, the *sharing-*Y *combinator* $\check{\mathsf{Y}}$, which creates an elementary cyclic structure (see Fig. 3). It is defined [Mac02c]:

$$(\check{\mathsf{Y}}F) \Longrightarrow y^{(1)} \quad \text{where } y \equiv (Fy^{(0)}). \tag{5}$$

The operation creates an elementary cycle, but it may be expanded by computation in combinator tree F. Examples of the use of $\check{\mathsf{Y}}$ are given in Sec. 3.1.2.

Data Representation: In anticipation of writing molecular combinatory programs in a high-level functional language, we have selected combinatory representations for several common data types, including Booleans (truth values), Lisp-style lists, and integers [Mac02c].

Since molecular combinatory computing makes use of combinator trees rather than bits, Boolean (truth) values do not have a natural binary representation as they do on digital computers. Nevertheless, they can be represented as small structures that operate as selector functions (the primary operational meaning of **true** and **false**). Using them, it is simple to define a conditional (**if-then-else**) construct [Mac02c].

We anticipate that Lisp-style lists or sequences will be useful in molecular combinatory computing [Mac02d], so we have selected a convenient representation from those proposed in the literature and investigated in our own prior research [Mac02c].

Integers are useful in nanotechnology applications of molecular combinatory computing for counting assembly steps to create structures of desired dimensions (see Sec. 3.1.2 for examples). There are a variety of ways of representing integers in combinator structures, many of which represent the integer N with a structure (often a chain) of size proportional to N. (Of course, integers can be represented in binary as lists of Boolean values, but this is not so useful in a combinatory framework as in ordinary computers [Mac02c].) We have chosen a representation that is especially useful in combinator programming: *Church numerals* (named for Alonzo Church) represent the integer N by an operator that iterates the application of a given function N times (i.e., an integer is an active entity). This representation has, indeed, proved very useful in the synthesis applications that we have developed. Also, it facilitates the representation of large integers, since it allows the representation of N to be compressed successively, from a structure of size $\mathcal{O}(N)$, to structures of size $\mathcal{O}(\log N)$, $\mathcal{O}(\log \log N)$, etc. [Mac02c].

Molecular Combinator Reference Manual: To facilitate the application of autonomous molecular computation to nanotechnology, we have compiled a *Molecular Combinator Reference Manual* [Mac02b], which is regularly updated and maintained online [Mac03b]. For the primitive combinators and housekeeping operations (KSŠŸDR), the manual includes descriptions of reactions in chemical notation and in the notation of the Substitution Reaction Simulator input language [Mac04d] (see Sec. 3.1.3 below). For the non-primitive combinators, it includes reductions to the primitive combinators (analogous to those in the Appendix to this report). In addition, we have computed the size, in terms of elementary combinators, of the non-primitive combinators when expanded; this is useful for computing the size of molecular combinator programs. Finally, the manual incorporates a catalog of useful properties (theorems) pertaining to all the combinators. Most of these are collected from the literature [CFC58, e.g.], but some are original.

3.1.2 Examples of Nanostructure Assembly

We have developed a number of examples of nanostructure assembly by molecular combinatory computing [Mac02a, Mac02c, Mac03e]. In addition to demonstrating the potential of the technique, these exercises have revealed a variety of opportunities and issues.

General: Our reports [Mac02a, Mac03e] demonstrate several different approaches to developing a combinatory program to assemble a desired structure. In this way they lay the foundations of a systematic approach to nanotechnology based on molecular combinatory computing.

To date, we have not made use of the ability to program nanostructure assembly in a high-level functional programming language and to have it automatically translated to combinators; there are several reasons. One is that the assembly problems that we have addressed so far are relatively easy to express in the ordinary notation of combi-

natory logic. Although these can be opaque to readers unfamiliar with combinatory logic (see displayed equations below), they have the advantage that they are easy to manipulate mathematically. In principle, one could replace the single-letter combinator names with more descriptive English words, but in practice it doesn't improve readability much and it makes mathematical manipulation more difficult. Another reason that we have worked at the combinatory logic level rather than the high-level language level is that we have not determined the best way to express sharing and recursion at the high level (see Sec. 4.2.2 below).

In all but the simplest cases, our programs are accompanied with a proof of correctness, or they have been derived mathematically so that the proof is implicit in the derivation. These proofs are generally straight-forward inductions, since combinators behave like mathematical functions. In addition, most of our programs have been simulated with the tree-transformation system (described below, Sec. 3.1.3) so that their correctness is empirically verified.

Although all molecular combinatory programs reduce to a few elementary combinators ($\mathsf{KS\check{S}\check{Y}}$), programs are not generally written in such low level terms; rather, they are written using higher-level combinators, which are defined in terms of the elementary ones (see the Appendix to this report for examples; see previous reports [Mac02b, Mac03b] for comprehensive definitions). Therefore, while a program for nanostructure assembly may be written briefly (see below for examples), it may expand into a much larger tree of elementary combinators. Since the size of these trees is relevant to the molecular program representations that must be synthesized, we have computed the size, in terms of our molecular building blocks, of most of the programs we have developed. This has been accomplished by using our software tools (Sec. 3.1.3) to translate the high-level combinator programs into elementary combinators, and to compute the sizes of the resulting trees. Sometimes the size of a program has a polynomial dependence on certain parameters (e.g., the dimensions of a membrane to be assembled), in which case we have used difference equations to derive a polynomial expression for the size, which can then be proved inductively.

Membrane Architectures: Our first example is the synthesis of a cross-linked membrane, such as shown in Fig. 4. The structure depicted is produced by the combinator program $\mathsf{xgrid}_{3,4}\mathsf{NNN}$, where xgrid is defined:

$$\mathsf{xgrid}_{m,n} = \mathsf{B}(\mathsf{B}(\mathsf{Z}_{m-1}\mathsf{W}))(\mathsf{B}(\mathsf{Z}_{n-1}\mathsf{W})(\mathsf{Z}_m\check{\Phi}_n)), \tag{6}$$

which is an abbreviation for a large binary tree of A, K, S, and $\check{\mathsf{S}}$ groups. Unfortunately, space does not permit an explanation of this program or a proof of its correctness, both of which may be found in a technical report [Mac02a]. However, all the combinators that it uses are defined in the appendix to this report. In the formula $\mathsf{xgrid}_{m,n}XYZ$, the parameters m and n are the height and width of the membrane, respectively. X, Y, and Z are the terminal groups to be used on the left ends, right ends, and bottoms of the chains. (In the expression $\mathsf{xgrid}_{3,4}\mathsf{NNN}$, N is any inert group.)

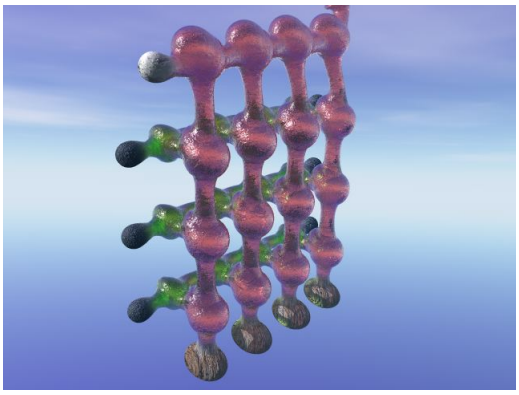

Figure 4: Visualization of cross-linked membrane produced by $\mathsf{xgrid}_{3,4}\mathsf{NNN}$. A groups are red, V groups green; other groups (N) are inert.

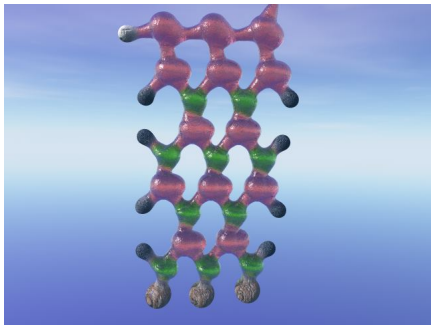

Figure 5: Visualization of small hexagonal membrane synthesized by $\mathsf{hgridt}_{2,3}\mathsf{N}$.

The size of $\mathsf{xgrid}_{m,n}$, the program structure to generate an $m \times n$ membrane, can be shown [Mac02a] to be $20m + 28n + 73$ primitive groups (A, K, S, Š). This does not seem to be unreasonable, even for large membranes, but it can be decreased more if necessary. For example, if $m = 10^k$, then Z_m in Eq. 6 can be replaced by $\mathsf{Z}_k\mathsf{Z}_{10}$, reducing the size of this part of the program from $\mathcal{O}(10^k)$ to $\mathcal{O}(k)$ (see [Mac02c] for explanation). Similar compressions can be applied to the other parts dependent on m and n. Therefore, by these recoding techniques the size of the program can be successively reduced to $\mathcal{O}(\log m + \log n)$, to $\mathcal{O}(\log\log m + \log\log n)$, etc. Furthermore, as will be explained later (p. 13), large membranes can be synthesized by iterative assembly of small patches.

Another synthesis example is the hexagonally structured membrane in Fig. 5 [Mac02a]. It is constructed by the combinator program $\mathsf{hgridt}_{2,3}\mathsf{N}$, where [Mac02a]:

$$\mathsf{Arow}_n = \mathsf{B}\check{\mathsf{W}}_{[n-1]} \circ \mathsf{B}^{[n]}, \tag{7}$$

$$\mathsf{Vrowt}_n = \check{\mathsf{W}}_{[n]} \circ \mathsf{KI} \circ \mathsf{K}_{(2n-2)} \circ \mathsf{B}^{[n-1]} \circ \mathsf{C}^{[n]}\mathsf{IN} \circ \mathsf{CIN}, \tag{8}$$

$$\mathsf{drowt}_n = \mathsf{Vrowt}_n \circ \mathsf{Arow}_n, \tag{9}$$

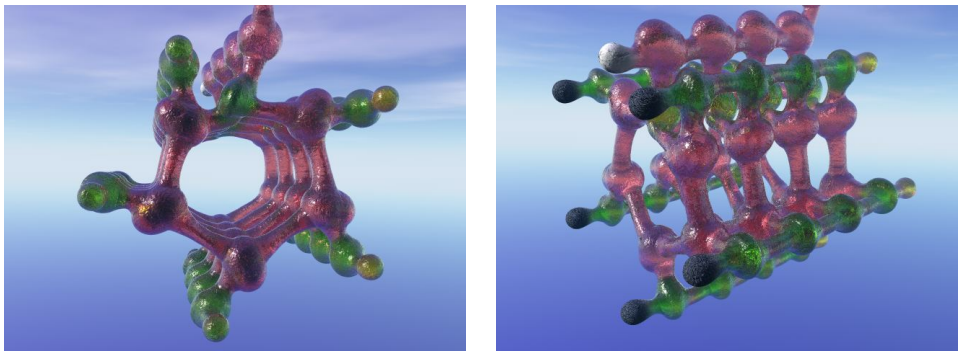

Figure 6: (A) Visualization of small nanotube, end view, produced by $\mathsf{xtube}_{5,4}\mathsf{N}$. (B) Side view.

$$\mathsf{hgridt}_{m,n} \quad = \quad \mathsf{Z}_{n-1}\mathsf{W}(\mathsf{Z}_m\mathsf{drowt}_n\mathsf{N}). \tag{10}$$

The size of the program is $\mathcal{O}(m+n)$ primitive combinators.

Nanotubes: A nanotube, such as shown in Fig. 6, can be synthesized by using the $\check{\mathsf{Y}}$ combinator to construct a cycle between the upper and lower margins of the cross-linked membrane (Fig. 4). The program is [Mac02a]:

$$\mathsf{xtube}_{m,n} = \hat{\mathsf{W}}^{m-1}(\mathsf{W}^{n-1}(\mathsf{Z}_m\check{\Phi}_n\mathsf{N})(\mathsf{B}^m\check{\mathsf{Y}}(\mathsf{C}_{[m]}\mathsf{I}))). \tag{11}$$

The size of $\mathsf{xtube}_{m,n}$ is $102m + 44n - 96$ primitive groups (A, K, N, S, $\check{\mathsf{S}}$, $\check{\mathsf{Y}}$).

Membranes with Non-unit Meshes: The membranes and nanotubes previously described are said to have a *unit mesh*, that is, the dimensions of the basic (square or hexagonal) cells are determined by the size of the primitive groups (A, V) and the links between them. It is relatively straight-forward to modify the preceding definitions to have larger mesh-dimensions (multiples of the cells); in addition, various pendant groups can be incorporated into the structure [Mac03e].

Patches: Large membranes may not be homogeneous in structure; often they will contain pores and active channels of various sorts embedded in a matrix. One way of assembling such a structure is by combining rectangular patches as in a patchwork quilt. To accomplish this we have defined a uniform interface for such patches [Mac03e]. Nanotubes can also be synthesized in patch format to allow end-to-end connection, and this operation can be iterated to assemble tubes of arbitrary length. Similarly, rectangular patches can be iteratively assembled, both horizontally and vertically, to hierarchically synthesize large, heterogeneous membranes. This allows us to build upon a basic library of elementary membrane patches, pores, and other nanostructural units.

Pores: We have developed molecular combinatory programs for rectangular *pores*, which are simply patches in which the interior is an open space [Mac03e]. These pores can be combined with other patches to create membranes with pores of a given size

and distribution (all in terms of the fundamental units, of course). Pores can be included in the surfaces of nanotubes as well.

Interface Protocol for Triggered Sensors: The primitive substitutions already mentioned ($\mathsf{K}, \mathsf{S}, \check{\mathsf{S}}, \check{\mathsf{Y}}$) are adequate for describing the assembly of static structures, but for dynamic applications we need additional operations that can respond to environmental conditions ("sensors") or have noncomputational effects ("actuators"). Many of these will be ad hoc additions to the basic computational framework, but we have developed general interface conventions to facilitate the development of a systematic nanotechnology [Mac03e]. For example, we may design a molecular group $\mathsf{K}_{-\lambda}$ that is normally inert, but is recognized (and therefore operates) as K in the presence of an environmental condition λ (e.g., light of a particular wavelength or a particular chemical species). Such a sensor may be used to control conditional execution, such as the opening or closing of a channel in a membrane [Mac03e]. However, this is not a complete solution to the problem of sensors and how they may be incorporated into the framework of molecular combinatory computing (see Sec. 4.2.2 below).

Simple Open-once Channels: Given such a sensor, channels that open and stay open were easy to implement, as described in a report [Mac03e]. In the former case, the sensor triggers the dissolution of the interior of its patch (perhaps using the deletion operator D to disassemble it). In the latter case, the sensor triggers a synthesis process to fill in a pore. Reusable channels (which open and close repeatedly) are more complicated, since, in order to reset themselves, they need a supply of sensor molecules that are protected from being triggered before they are used (see Sec. 4.2.2 below).

Computationally Implemented Actuators: Nano-actuators have some physical effect depending on a computational process. Certainly, many of these will be synthesized for special purposes. However, we have investigated actuators based directly on the computational reactions. To give a very simple example, we may program a computation that synthesizes a chain of some length; we may also program a computation that collapses a chain into a single link. The two of these can be used together, like opposing muscle groups, to cause motion under molecular program control. The force that can be exerted will depend on the bond strength of the nodes and links (probably on the order of 50 kJ/mol; see Sec. 3.2). However, these forces can be combined additively, as individual muscle fibers work cooperatively in a muscle.

3.1.3 Software Tools

In the course of this project we developed a number of useful software tools, which will be available on the project website. One of the most important tools is an interpreter for *abstract calculi* [Mac90, ch. 10], also known as *term-rewriting systems* and *tree-substitution systems*. The interactive program, written in LISP, allows the user to provide a set of tree-transformation rules, which it then applies repeatedly to any

tree provided by the user. This tool has many applications in molecular combinatory computing, including translation of high-level combinators to S and K, execution of combinator trees, and computation of combinator program size. All of these rules will be available on the project website.

The definitions of higher-level combinators in terms of the primitive combinators, such as we find in the Appendix to this report and in our reference manual [Mac03b], can be used almost unchanged as tree-transformation rules for translating any combinator program into the primitives (KSŠŸ). We have used it both to demonstrate the correctness of our molecular combinator programs and to compute their sizes. (See Secs. 3.1.1 and 3.1.2 above.)

The equations that define the primitive combinators (Eqs. 1 – 5) are also directly translatable to tree-transformation rules, and so it is easy to simulate the execution of molecular combinatory programs. It is important to observe, however, that this is not a complete simulation, because this tool is a *tree*-transformation system, so it does not deal with the issues of sharing that are essential in nanostructure assembly. These and other issues relevant to molecular computation are addressed by the Substitution Reaction Simulator discussed below.

For greater efficiency and convenience in simulations, there is no need to translate higher-level combinators to the primitives in order to execute them; their behavior can be described in terms of simple substitution rules (given in many sources [CFC58, Mac03b]), which are directly usable by the tree-transformation system. This is the approach we have used in most of our tests of molecular combinatory programs.

Finally, it is very easy to write tree-transformation rules that compute the size of SK trees, and we have used this tool for calculating the size of molecular combinator program trees (see examples in our reports [Mac03b, Mac02c, Mac02a]).

Substitution Reaction Simulator: Although the tree-transformation system has proved useful for many purposes, it ignores a number of issues that are important in molecular computation. For example, the "sharing" of structures, which is invisible in abstract combinatory computing, is essential to the construction of non-tree structured, and especially cyclic, structures. Another problem that must be investigated is the effect of error on the behavior of molecular combinatory computing. Therefore, we have developed a *Substitution Reaction Simulator* (SRS), programmed in Java, which allows us to address these issues [And04].

To keep the simulator general and extendable, we have not built specific chemical reactions into it, but we have defined a SRS input language [Mac04d], which allows the description, as abstract chemical equations, of the required substitution reactions. Examples of such reactions can be found in our reports [Mac02b, Mac02c, Mac03b]. The language allows additional parameters to be specified, such as the relative probability of a reaction taking place, and other parameters that may be used in the future (e.g., change of free energy).

A prototype of the SRS has been implemented in Java and tested on small examples [And04]. It provides the following basic functionality: (1) reading in the reaction

specifications, (2) reading in a description of the relative initial concentrations of specified molecular structures in the "soup," (3) simulation of the reactions for a specified number of substitutions, and (4) recording the final contents of the soup, or probing it with user-provided templates for particular structures.

3.2 Possible Molecular Implementation

Although this project was focused on an exploration of the potential of molecular combinatory computing for nanostructure synthesis and control, we have been sensitive to the fact that its advantages are illusory unless a molecular implementation of the combinatory operations can be developed. Therefore we have spent some time trying to develop at least one feasible molecular implementation. The two principal problems are: (1) How are the combinator networks represented molecularly? (2) How are the substitution operations implemented molecularly?

3.2.1 Combinator Networks

Requirements: Combinatory computing proceeds by making substitutions in networks of interconnected nodes. These networks constitute both the medium in which computation takes place and the nanostructure created by the computational process. Therefore it was necessary to consider the molecular implementation of these networks as well as the processes by which they might be transformed according to the rules of combinatory computing.

The first requirement is that nodes and linking groups need to be stable in themselves, but the interconnections between them need to be sufficiently labile to permit the substitutions. Second, the node types (A, K, S, etc.) need to identifiable, so that the correct substitutions take place. In addition, for more secure matching of structures, the link (L) groups should be identifiable. Further, it is necessary to be able to distinguish the various binding sites on a node. For example, an A node has three distinct sites: the result site, an operator argument, and an operand argument [Mac02c].

Hydrogen-Bonded Covalent Subunits: As a consequence of these consideration we have decided provisionally to implement the nodes and linking groups by covalently-structured molecular building blocks (MBBs) and to interconnect them by hydrogen bonds. We use a covalent framework for the nodes and links because they provide a comparatively rigid framework in which to embed hydrogen bonding sites, and because there is an extensive synthetic precedent for engineering molecules of the required shape and with appropriately located hydrogen bonding sites [AS03]. This is in fact the structural basis of both DNA and proteins (hydrogen bonding as a means of connecting and identifying covalently-bonded subunits).

Hydrogen bonds are used to interconnect the MBBs because they are labile in aqueous solution, permitting continual disassembly and re-assembly of structures. (H-bond strengths are 2–40 kJ/mol.) Further, other laboratories have demonstrated

the synthesis and manipulation of large hydrogen-bonded tree-like structures (*dendrimers*) [SSW91, ZZRK96]. Nevertheless, hydrogen bonds are not very stable in aqueous solution, so there may be a delicate balance between stability and lability.

It is necessary to be able to distinguish the "head" and "tail" ends [Mac02c] of the L groups (i.e., our graph edges are directed), and we estimate that two or three H-bonds are required to do this securely. (For comparison, thymine and adenine have two H-bonds, cytosine and guanine have three.) Therefore, if we take 20 kJ/mol as the strength of a typical H-bond, then the total connection strength of a link will be about 50 kJ/mol.

Hydrogen bonding can also be used for recognizing different kinds of nodes by synthesizing them with unique arrangements of donor and acceptor regions. Currently [Mac02c], we are using eleven different node types (A, D, K, L, P, Q, R, S, Š, V, Ÿ), so it would seem that arrangements of five H-bonds would be sufficient (since they accommodate 16 complementary pairs of bond patterns). (Actually, linear patterns of from one to four bonds are sufficient – 15 complementary pairs – but the slight savings does not seem worth the risk of less secure identification.)

As previously remarked, it is necessary to be able to distinguish the three binding sites of an A node. However, since the "tail" of any L group must be able to bind to either of the A's argument sites, they must use the same H-bond pattern. Therefore, we have concluded that at least part of the discrimination of the A's binding sites must be on the basis of the orientation of the A node. Fortunately, the orientation specificity of H-bonds allows this.

Estimation of Sizes of Building Blocks: A number of hydrogen-bonding sites can be located in a small area. For example, thymine (23 atoms) and adenine (26 atoms) have two H-bonds; amino acids are generally small, on the order of 30 atoms and as few as 10. On the basis of the above considerations, we have estimated — very roughly! — that our primitive combinators (K, S, Š, Ÿ) might be 90 atoms in size, L groups about 120, and ternary groups (A, V, R) about 150.

3.2.2 Substitution Reactions

Requirements: Our investigation has identified several requirements on a molecular implementation of the primitive combinator substitutions.

First, there must be a way of matching the network configurations that enable the substitution reactions. For example, a K-substitution (Eq. 1) is enabled by a leftward-branching tree of the form $((\mathsf{K}X)Y)$, and an S-substitution (Eq. 2) is enabled by a leftward-branching tree of the form $(((\mathsf{S}X)Y)Z)$ (see Figs. 1 and 2). So also for the other primitive combinators (D, R, Š, Ÿ).

Second, the variable parts of the matched structures (represented in the substitution rules by italic variables such as X and Y), which may be arbitrarily large supramolecular networks, must be bound in some way. Third, a new molecular structure must be constructed, incorporating some or all of these variable parts.

Further, reaction waste products must be recycled or eliminated from the system, for several reasons. An obvious one is efficiency; another is to avoid the reaction space becoming clogged with waste. Less obvious is the fact that discarded molecular networks (such as Y in Eq. 1) may contain large executable structures; by the laws of combinatory logic, computation in these discarded networks cannot affect the computational result, but they can consume resources.

Finally, there are energetic constraints on the substitution reactions, to which we now turn.

Fundamental Energetic Constraints: On the one hand, any system that is computationally universal (i.e., equivalent to a Turing machine in power) must permit nonterminating computations. On the other, a spontaneous chemical reaction will take place only if it decreases Gibbs free energy; spontaneous reactions tend to an equilibrium state. Therefore, molecular combinatory computing will require an external source of energy or reaction resources; it cannot continue indefinitely in a closed system.

Fortunately we have several recent concrete examples of how such nonterminating processes may be fueled. For example, Koumura et al. [KZvD$^+$99] have demonstrated continuous (nonterminating) unidirectional rotary motion driven by ultraviolet light. In the four-phase rotation, alternating phases are by photochemical reaction (uphill) and by thermal relaxation (downhill). Also, Yurke et al. [YTM$^+$00] have demonstrated DNA "tweezers," which can be cycled between their open and closed states so long as an appropriate "DNA fuel" is provided. Both of these provide plausible models of how molecular combinatory computation might be powered. As a consequence we have concluded that the individual steps of a combinator substitution must be either energetically "downhill" or fueled by external energy or reactions resources. In our case, the most likely sources of fuel are the various species of "substitutase" molecules (see next).

Use of Synthetic Substitutase Molecules: To implement the substitution processes we have investigated the use of enzyme-like covalently-structured molecules to recognize network sites at which substitutions are allowed, and (through graded electrostatic interactions) to rearrange the hydrogen bonds to effect the substitutions. Again, the rich synthetic precedent for covalently-structured molecules makes it likely that the required enzyme-like compounds, which we call *substitutase molecules*, can be engineered. Our research suggests the use of three kinds of substitutase molecules for each primitive combinator; they implement three stages in each substitution operation.

The first of these molecules, which we call *analysase*, is intended to recognize the pattern enabling the operation and to bind to the components of the matching subtree. For example, K-analysase binds to a structure of the form $((\mathsf{K}X)Y)$ (see Fig. 1), in particular to links to the variable components U, X, and Y.

The second stage, which is implemented by a *permutase* molecule, physically relocates some of the components to prepare them for the last stage. To this end, we have

investigated (inconclusively, to date) the use of graded electrostatic interactions to move the bound variable parts into position for the correct substitution product. The permutase molecule also includes any fixed combinators (e.g., R, V) that are required for the product, and which are bound to other product components at this time. At the end of this stage, the product network is essentially complete, but still bound to the permutase molecule.

The final molecule, a *synthesase*, recognizes the configuration created by the permutase, and binds to the waste structures, displacing and releasing the desired product from the permutase. For example, S- or Š-synthesase will remove the (S- or Š-) permutase and release the structure shown on the right in Fig. 2.

3.2.3 Guidelines for Use of Triggered Sensors

Due to the order-independence of combinatory computing, we have determined triggered sensors must be handled carefully [Mac03e]. In particular, all self-assembly of structures containing triggered sensors must be allowed to proceed to completion under conditions in which the triggering situations are impossible. Once the structure is complete (including all required replication of the sensors), it can be placed in its operational environment, in which the triggering condition may occur.

3.3 Application to Radical Reconfiguration of Computers

This year we conducted a preliminary investigation of a particular application of universally programmable intelligent matter (UPIM): the radical reconfiguration of computers [Mac04c]. There are many well-known reasons for designing computers to be reconfigurable, including flexibility, adaptability, and damage recovery in environments (such as earth orbit or interplanetary space) where the replacement of processors and transducers would be very expensive or impossible.

Ordinary approaches to reconfiguration are limited to changing connections among fixed components (e.g. gates). By *radical reconfiguration* we mean the ability to change the components themselves by rearranging the molecules of which they are composed. For example, the radical reconfiguration of transducers permits the creation of new sensors and actuators, and radical reconfiguration of processors permits the reallocation of matter to different components (e.g., memory or logic). Radical reconfiguration is valuable if a deployed system's mission changes or if it must recover from damage or serious failure.

UPIM is an especially promising approach to radical reconfiguration, because it provides a general-purpose mechanism for rearranging matter under program control. Thus, under either autonomous or external control, a program could be used to synthesize processor components and transducer structures with feature sizes on the order of hundreds of nanometers. It is anticipated that in future work we will explore this idea in more detail.

3.4 Publications and Presentations

To date, the publications resulting from this project include one journal article [Macss], three conference presentations with summaries in the proceedings [Mac02e, Mac03a, Mac03c], and eleven technical reports [And04, Mac02a, Mac02b, Mac02c, Mac02d, Mac03b, Mac03d, Mac03e, Mac04a, Mac04c, Mac04d]. In addition we have made four conference or workshop presentations of this work (IEEE Nano 2002 [Mac02e], IEEE Nano 2003 [Mac03c], 7th Joint Conference on Information Science [Mac03a], and an ARL Workshop on Reconfigurable Electronics). These are all available at the project website, <http://www.cs.utk.edu/~mclennan/UPIM>.

4 Future Directions

4.1 Relation to Prior Work

Our plans for the future are a continuation of what we have described, but also an expansion of its scope. Having established the potential of combinatory logic as a basis for autonomous molecular computation for nanotechnology, we intend to continue the development of molecular combinatory computing, both extending its range and resolving some of the issues that we have identified (see Sec. 4.2 below). However, we plan to expand the scope of this project to include other models of computation, which are also compatible with autonomous molecular computation, but may be preferable to combinatory logic (see Sec. 4.3 below).

In general, our objective is to investigate the use of autonomous molecular computation for nanostructure synthesis and control. In particular our investigation will focus on models of computation that are more compatible with molecular processes than are conventional models founded on sequential binary logic. At this time two models look most promising: combinatory computation, which we have investigated for several years, and lattice-swarm oriented models, which is a new direction for our research. However, as the investigation progresses, we expect that other models of computation may be considered. In the following sections we present our objectives in each of these research directions.

4.2 Continue Investigation of Molecular Combinatory Computing

4.2.1 Motivation for Continuing Investigation of MCC Approach

As described above (Sec. 3) we have made considerable progress in the investigation of molecular combinatory computing applied to nanostructure synthesis and control. Although a number of issues have been identified, the results are sufficiently promising, and the potential benefits sufficiently great, to encourage us to continue to pursue this approach and to resolve the issues.

These benefits (which may not pertain to alternatives, such as the lattice-swarm oriented model) include: (1) an existing technology for compiling from high-level programming languages into molecular programs, (2) the Church-Rosser theorem, which defines the conditions for consistent results in the presence of asynchronous, parallel computation at the molecular level, and (3) provable universality, that is, the guarantee that the model of computation is sufficiently powerful to accomplish anything that can be done on a digital computer.

4.2.2 Further Investigation of MCC Applied to Nanosynthesis and Control

Indefinite and Unlimited Iteration: The examples of nanoscale computing that we have developed so far have made use of *definite iteration*, that is, the repetition of a computation a pre-determined number of times (analogous to a **for**-loop in ordinary programming). For example, we may assemble an $m \times n$ membrane for specific values of m and n. While definite iteration is adequate for many purposes, in some applications we may need a computation that iterates an indefinite number of times (e.g., until some condition occurs), analogous to a **while**-loop. In other applications, such as channels that can open and close repeatedly, we require a computation to be able to take place an unlimited number of times (a **do forever** loop). To implement indefinite iteration and recursive processes, combinatory computing traditionally makes use of an operator Y (definable in terms of S and K), which is capable of producing unlimited copies of a program structure. Unfortunately, there are problems with applying this approach in a molecular context. The order-independence and parallelism of combinatory computing, which are generally so valuable in molecular computing, allow the Y operator to produce copies of a structure, regardless of whether they are needed. Indeed, such a process has the potential of consuming all the reaction resources and of clogging the computation space with unneeded structures. There are ways of implementing a "lazy Y," which replicates only when it is required to do so [Mac90, 525–30], but it needs to be determined if they can be adapted to molecular computation. If they cannot, then some alternative means of indefinite and unlimited iteration must be devised.

Temporal Control of Events: In conventional models of computation, operations proceed sequentially, and parallel computation has to be superimposed on an essentially sequential model. One advantage of the combinatory computation model is that it is inherently asynchronous, and operations may proceed (safely) in parallel. In general, this property is valuable in autonomous molecular computations, but when operations are required to proceed sequentially, or to be synchronized in some way, then a purely combinatory model is less suitable. Computer scientists who have investigated combinatory computing as a basis of massively parallel computer architectures have developed a number of solutions to this problem, but some of these depend on the use of conventional electronic computer technology. Therefore,

we must adapt these solutions or find new solutions appropriate to an autonomous molecular framework. We will mention several specific topics to be investigated.

A *resettable* or *multishot* channel requires, in effect, program code that is executed repeatedly (e.g., the code to open or close the channel). However, in combinatory computing, the code is consumed by the process of computation, and so combinatory computing traditionally makes uses of the Y operator to produce unlimited copies of a program structure. This approach might still be usable, if a solution can be found to the problem of indefinite and unlimited iteration. However, it can be argued that this is an inefficient way of implementing such channels, and that there might be a more direct approach. In the past, such ad hoc extensions have been made to functional programming languages; they are called *imperative* because they order an operation to take place at a specific time. But there remains the problem of how to control, in the context of autonomous molecular computation, the time at which such an operation takes place.

Even if something like the lazy Y operator can be adapted to molecular computation, there are problems with applying it to resettable sensors. In effect, Y creates multiple copies of a single program structure. If this structure contains triggerable sensors, then they may be triggered any time their corresponding triggering condition occurs. If this occurs in the structure being copied, then the triggered sensor will be copied, rather than an untriggered sensor. (Some partial solutions are discussed in a technical report [Mac03e].)

In general, this problem can be seen as symptomatic of an inability to control the times at which a sensor is responsive to its condition. One could instead use *polled sensors*, which are responsive only when they ordered to be so, but this requires some kind of time sequencing or control, which we have seen already to be problematic in the context of molecular combinatory computing.

A similar issue arises with nano-actuation and other effectors, since the action must take place at the correct time. This problem is not so difficult, however, since the rules of combinatory computing prevent a substitution from taking place unless the correct network configuration exists [Mac03e, Sec. 5]. However, even when the enabling configuration has formed, there is no guarantee that the substitution *will* take place within any specified time bound; the expected time depends on the reaction rate, which depends on the concentrations of reaction resources and many other factors. In some applications we will want the effect to take place within a bounded time after the enabling condition occurs. We do not know the best solution to this problem, but it is likely to be related to the approach we will develop to polled sensors.

Many useful applications (e.g., nano-motors, cilia, flagella) require regular repetition of some effect. One of the advantages of combinatory computing for many purposes (a consequence of the Church-Rosser property) is that multiple activities can proceed in parallel, to the extent permitted by the computation, but for cyclic effects we generally want the cycles to be non-overlapping. Our approach to the temporal control of events will have to include means for governing cyclic activities.

Synthetic Universality: We are uncertain of the ability of our current operator set (SŠKY̌) to produce arbitrary cyclic directed graphs, which is obviously essential for general nanostructure assembly. This may be surprising in light of the computational universality of the SK operators, but it must be remembered that this implies only the ability to program any computable function of SK *trees* (abstract "terms"). Non-trees arise only because the Church-Rosser property permits multiple instances of identical subgraphs to be represented by a single shared instance. Similarly, cyclic graphs correspond to potentially infinite trees with self-similar structure (mathematically, they are the limits — in a continuous lattice — of an infinite sequence of progressively larger self-similar trees). Although, in the context of abstract term-rewriting systems with the Church-Rosser property, sharing and replication are equivalent, the physical representations are different, and so for our purposes they represent different nanostructures. As already discussed, this is the reason that we have separate S and Š operators, which produce replicated and shared structures: physically different but mathematically equivalent. In combinatory programming, recursive structures are created by the Y operator. If the result is represented as a potentially infinite replicating structure, then it can be constructed by the standard Y, which is definable in terms of S and K (and thus need not be included as a primitive operator). However, if the result is represented as a cyclic structure, then a new operator Y̌ is required, which creates the cycle through a sharing-node (see Sec. 3.1.1 above). In traditional functional programming, the Y operator is sufficient to create arbitrary self-referential structures by means of mutually recursive definitions [Bur75, p. 21]. However, these techniques do not work directly in the context of molecular combinatory computing, since each instance of sharing, including that in cyclic structures, introduces a sharing-node, and these nodes interfere with the recognition and substitution processes essential for molecular computation. (In implementations of combinatory computing on conventional computers, in contrast, the addressing mechanism permits there to be an arbitrary number of pointers to the same structure, each invisible to the others.) As a consequence, while the currently specified Y̌ operator is sufficient for creating any structure with a single cycle, we have not yet found a means of using it to create arbitrary multiply-cyclic structures, except in specific cases. (One specific example: we can use Y̌ to create a single ring of a nanotube, and then use other combinatory operators to replicate this and assemble the rings into a nanotube [Mac02a, Mac03e], but we cannot express an alternative assembly method: synthesize a membrane and then create a nanotube by creating connections between its upper and lower borders, stitching them together by parallel cyclic connections.) Therefore, one objective of future research is either to find a way of creating arbitrary structures with the current operators, or to find an alternative to Y̌ that permits their creation.

As the foregoing discussion makes clear, we cannot depend on existing proofs of the computational universality of SK to guarantee the ability to assemble arbitrary structures. Therefore, once we have found a plausible solution to the construction of

arbitrary structures, we intend to develop a formal proof of that fact.

As previously discussed, one of the advantages of using combinatory logic as a medium of computation is the extensive research by computer scientists into the translation of high-level functional programming languages into combinatory structures (specifically SK trees). This offers the real possibility of being able to program the assembly and behavior of nanostructures in a high-level programming language, thus improving the facility and reliability of nanotechnology. To date, however, the molecular programs that we have developed have been expressed directly in terms of combinatory operators (see Sec. 3.1.2 and our progress reports [Mac02a, Mac03e] for examples). While convenient for mathematical proof and analysis, the notation is low-level and comparable to programming in assembly language. However, in the context of molecular programming, there are additional complications arising from the different handling of shared structures, as described in connection with Y̌. Again, the sharing nodes block the substitution reactions. At this time, we are planning on designing high-level ways of expressing and manipulating shared structures, which will have the advantages of high-level programming while remaining compatible with the constraints of molecular computing.

Develop Chain-to-tree Reactions: Our model for molecular computation (Sec. 2) calls for:

1. translation of high-level program into combinator trees;
2. flattening combinator tree into a fully parenthesized linear string;
3. synthesis of corresponding chain polymer (e.g., DNA strand);
4. massive replication of the chain (e.g., by PCR);
5. translation of chains into molecular combinator trees, ready for computation.

The last stage, which amounts to the translation of a fully parenthesized string into a tree, is conceptually simple, but it needs to be investigated in detail to give a complete account of molecular combinatory computing. One approach is to devise reactions that walk down the string (rather like a ribosome) and synthesize the tree. Such reactions are not much more complicated than the network replication and deletion operations [Mac02c, Mac03b], but they do add additional complexity to the system as a whole. If the molecular implementation described above (Sec. 3.2) proves feasible, then it should also be able to be used for this translation task.

A different approach depends on the fact that the string-to-tree translation is a (relatively simple) computable function, and therefore can be expressed as a combinatory program. To accomplish this it is sufficient to: (1) be able to synthesize molecular program chains in one of the combinator representations of LISP-style lists [Mac02c]; (2) be able to attach a fixed molecular network representing the translation program to the appropriate end of the chain. Since, at this time, we do not know which approach is better, we plan to develop both.

Synthesis Test Cases: We have developed a simple means of assembling an *open-once channel*, that is, a channel that, in response to a triggering conditions, opens and stays open thereafter (Sec. 3.1.2 above). It operates as follows: upon the triggering condition, the sole connection to a subnetwork of a membrane is destroyed, which makes that subnetwork subject to disassembly; in effect, a well-defined region of the membrane dissolves and opens up. Preliminary investigations suggest that a *close-once channel* could be implemented in an analogous way: upon the triggering condition the channel synthesizes a membrane patch to cover a previously open pore in the membrane. Although the idea is simple, the details have not been worked out, but they should.

A *resettable* or *multishot channel* can be opened and closed many times. We intend to develop a program for a resettable channel, but it presupposes a solution to the problems of temporal control (especially of sensors) discussed above.

The channels that we have already developed, and the resettable channel, which we are proposing to develop, are all *passive*; that is, they depend on osmotic pressure to transport molecules through the membrane opening. Thus, their selectivity is limited primarily to the size of the molecules. Greater selectivity can be achieved with active transport, which can bind a specific molecular species on one side of a membrane, transport it to the opposite side, and release it. Active transport depends on the identification or synthesis of specific molecules that will bind the species of interest, which is outside the scope of this project. Nevertheless, we intend to investigate and define general interface requirements so that specific recognition molecules can be integrated in the molecular combinatory framework. In addition, since an active channel needs to be able to operate repetitively, it will depend on developing an approach to resettable channels, as described above.

Many nano-actuators will be the result of ad hoc chemical synthesis (e.g., molecules with conformational changes controlled by molecular combinatory computing). However, we have also been investigating the use of the molecular computing process to implement nano-actuators. For example, a computation condition can trigger the assembly of a linear chain; similarly, a condition can trigger the collapse of a chain. The two processes can be used in tandem, like opposed muscle groups, to create a nano-actuator capable of linear motion. The general approach is clear, but we intend to develop some specific nano-actuators of this kind.

Cilia are important in biological systems for controlling the movement of fluids across membranes and through tubes, as well for providing and controlling motility of organisms. Therefore an artificial cilium is a good test case for molecular combinatory computing. The cilium has to be capable of oscillatory motion under control of a molecular computation; thus it integrates solution to several of the issues described above (temporal control, cyclic operation, nano-actuation).

While a linear-motion nano-actuator is important for many applications (e.g. artificial muscles), rotary motion is required for others. Therefore, we intend to investigate the implementation of a rotary nano-actuator by molecular combinatory

computing. We anticipate a three- or four-pole design, perhaps using the nano-ring architecture already designed [Mac03e]. Implementation of a rotary actuator presupposes a solution to the problem of controlling cyclic processes (see p. 21).

A rotating flagellum stands as an important challenge for molecular combinatory computing. It should be capable of rotating clockwise or counter-clockwise under program control. Again, it builds upon the solutions to many of the issues already discussed (such as the rotary actuator).

4.2.3 Continue Development of Simulation Software

The current prototype of the Substitution Reaction Simulator is needlessly inefficient, which limits its use on large programs. Correcting this problem should be straightforward. The current (largely non-graphical) interface is also poorly adapted to large problems, so we intend to investigate more usable alternatives. Further, this simulator has been tested only on relatively small programs, and before it is put into production use, it needs to be tested on much larger problems (hundreds or thousands of combinators), and any latent bugs repaired. Further development of the simulator will be delegated to a GRA.

4.2.4 Investigate Error Control

Identify Expected Error Modalities: Combinatory programs are similar to conventional programs in that they are very sensitive to error. Unfortunately, while errors are rare on electronic digital computers, they are to be expected in molecular computation. Also, prior simulation studies have shown that a significant fraction of random combinator trees produce nonterminating computations in which the trees grow without bound [Mac97, Yar00]. While there may be some applications for such processes, if they are unintended they run the risk of consuming all the reaction resources and congesting the reaction space with unwanted structures. Therefore some means must be found to control the effects of errors.

A first step is identification of possible error modalities. This presupposes some idea of the molecular implementation of the combinatory substitutions, which is an ongoing subject of research. Nevertheless, the following error modalities are apparent: (1) template mismatch (configurations that do not completely match a reaction template may nevertheless trigger the reaction), (2) unreliable linkage (hydrogen bonds are weak, 2–40 kJ/mol., and may be easily broken; conversely, there will be some unwanted "stickiness" between hydrogen bonding sites that are not intended to match), (3) reversible reactions (all chemical reactions are to some extent reversible, and so we must ensure that temporary reversals do not block or misguide the overall forward progress of computation), (4) metastable states (substitutions will progress through unstable intermediate states, which may nevertheless persist long enough to cause undesirable results), and perhaps others.

Simulation Studies: Once likely error modalities have been identified, and as they continue to be identified, we can incorporate them into the Substitution Reaction Simulator, which will also be one vehicle for investigating various means of error control and recovery. Simulation studies will be supplemented by mathematical analysis, where possible. (Since we consider the management of error to be a central issue in molecular computation, we also intend to investigate alternate models of computation that are less sensitive to errors than combinatory computing; see below on alternate models.)

4.2.5 Investigate Steric Constraints

Combinatory logic treats program and data structures as abstract graphs, but in molecular combinatory computing these graphs are represented as physical molecular networks. Thus the nodes and links have definite sizes and may be in varying physical orientations. To date, our investigation has ignored such issues, except insofar as they affect the architecture assembled by a molecular computation [Mac03e]. Nevertheless, these issues must be addressed. First, the bonds between the nodes and links of the network must be reasonably flexible in order to allow the assembly of a variety of architectures, which means that mathematically equivalent network configurations can occur with physically different relative orientations of their components. On the other hand, each substitution molecule will be designed to recognize one or at most a few physical configurations, so a reaction may be delayed until the network assumes the correct conformation, or it may be blocked entirely if the network has become relatively rigid (e.g., when assembly of a structure is well advanced). Second, the networks may get quite dense, especially when they are in the later stages of assembly, and so we must consider whether the substitution molecules will find their way to the sites on which they are intended to operate.

At this time we intend to address the problem of steric constraints on molecular combinatory computing with analytic techniques, addressing the geometry of the networks. Eventually we may want to extend the reaction simulator, or develop a new simulator, to address the stochastic aspects of the problem. Even in this case, however, we hope to maintain a higher level of abstraction (and thus a lower level of complexity) than would be the case with detailed molecular modeling.

4.2.6 Determine Feasibility of Molecular Implementation of SK Reduction

Specific molecular implementation is not a primary research topic for this project, except so far as necessary to establish the probable feasibility of molecular combinatory computing. Nevertheless, we intend to continue to devote some effort to investigating possible implementations, both for the sake of feasibility, but also with the aim of eventual collaboration with chemists. Such collaboration will be easier to establish if we have shown that the molecular computing approach to nanotechnology is both valuable and plausible.

As already mentioned, the molecular implementation that looks most promising — covalently-structured molecular building blocks connected with hydrogen bonds — includes DNA as a specific case. Therefore it is sensible to consider the rapidly developing technology of DNA computation and nano-assembly as a basis for molecular combinatory computing. Although much of this research to date has been directed toward the assembly of some nanostructure as an equilibrium state, there is continuing work on non-equilibrium systems (such as molecular nano-motors), which could be adapted for autonomous molecular computation [KZvD$^+$99, YZSS02, YYD$^+$ed, YTM$^+$00]. Therefore, we plan a detailed investigation of the techniques of DNA technology to the problems of molecular combinatory computing. In addition to the DNA-oriented investigation, we intend to continue our exploration of the synthesis of enzyme-like "substitutase" molecules suitable for implementing the substitution reactions.

4.3 Investigate Lattice-Swarm Oriented Model

Processes at the molecular level do not proceed with the same sort of precision that we expect from digital electronic circuits. This suggests that for a model of autonomous molecular computation we might look toward self-organizing systems in nature, which accomplish their purposes in spite of noise, imprecision, damage, and other disruptive influences [Mac04b]. Such systems are stochastic, but lead to a result or behavior that is determinate in some relevant way. Many of these systems have generalized equilibria to which they return after perturbation; by a generalized equilibrium I mean either an (ordinary) static equilibrium, or a *stationary state*, which is a temporarily stable ensemble of (thermodynamically) non-equilibrium states. This property provides a basis for error correction and repair.

One model of self-organization, which we have begun investigating as a basis for autonomous molecular computation, is the *lattice-swarm* model of wasp nest building developed by Theraulaz and Bonabeau [TB95a, TB95b] (see also [BDT99, ch. 6] and [CLDF$^+$01, ch. 19]). In this model, simple agents detect local configurations of blocks in a (cubic or hexagonal) lattice structure, and deposit building blocks of a type determined by the configuration. Depending on the template-deposition rules, such systems have been shown to be capable of assembling a variety of complex structures, some similar to natural wasp nests, other not. Although the model is stochastic, some sets of rules lead to a determinate structure, others lead to a variety of results, which are generically similar, but differ in specific detail. This reliability is in part a consequence of the use of *stigmergy*, which refers to the principle by which an incomplete structure governs its own future development by means of the agents.

Extend lattice-swarm model to allow replacement and removal as well as addition: Although the lattice-swarm model is promising as a basis for molecular self-assembly, it is inadequate as a basis for nanotechnology by autonomous molecular computation, because its agents are capable of adding blocks to the structure, but not removing or replacing them. (Certainly, one may implement autonomous molecular

computation by self-assembly, as has been demonstrated in DNA computing [LWR00, MLRS00, Rei02, WYS98, Win96], but such an approach is both wasteful, for it retains all intermediate results in an ever-growing transcript of the computation, and it is inadequate as a basis for dynamic applications, which may be activated and compute any number of times.) Therefore we propose to extend the lattice-swarm model to include the ability to replace and remove building blocks, which we call a *lattice-swarm oriented (LSO) model*. We are unaware if anyone has investigated this extension, but in either case we will.

Explore ability of LSO approach to deal with errors: A principal goal of the LSO approach is the robust synthesis and control of nanostructures in the presence of errors and other stochastic effects. Therefore, after identifying typical error modalities, we will conduct systematic experiments on the effects of errors on the assembled nanostructures. We will attempt also to outline design principles that will minimize the undesirable effects of errors (some examples of which already exist in the lattice-swarm literature [BDT99, ch. 6] [CLDF$^+$01, ch. 19] [TB95a, TB95b]).

Provide proof of computational universality: One of the advantages of the combinatory computing approach is that we have proofs that certain sets of operators (e.g., S and K) are computationally universal. This gives us a certain (albeit theoretical) guarantee that we will have a sufficiently powerful computational model for our future needs, unanticipated as well as anticipated. Therefore, it is desirable to have the same confidence with the LSO approach. It seems intuitively plausible that the addition, deletion, and replacement operations that we have described should be sufficient to simulate a Turing machine, but this intuition needs to be backed up by a formal proof, which is therefore an objective of this project. (Perhaps existing proofs of the universality of DNA computing can be adapted.) The proof itself may illuminate some requirements for LSO molecular computing.

Explore general principles for obtaining desired structures: One difficulty with the LSO approach is that self-organizing processes in biology, such as wasp nest building, are a result of evolution by natural selection. Models such as the lattice-swarm explain how such processes take place, but not how to design them. Although there has been some attempt to discover general engineering principles [Mas02], self-organization as a technology is still poorly understood. Therefore, we intend to devote a part of our effort to the investigation of such principles, at least in the context of LSO models. Part of the work will be empirical, comparable to the modeling of biological systems: we will observe the sorts of structures assembled by various sets of rules, select the ones with most potential use, and investigate variations of those rules. From these experiments we hope to identify principles that will guide the systematic application of LSO models to nanostructure synthesis and control. Unfortunately, the LSO approach lacks one of the advantages of the combinatory approach, namely the existence of a well developed technology for translating high-level languages into molecular programs.

Develop simulation software: Due to the stochastic nature of the LSO ap-

proach, simulation-based empirical investigations will be essential. Fortunately, the model is quite simple, so developing an efficient simulator should not be difficult. (Perhaps we may even get permission to modify Theraulaz and Bonabeau's program.) In addition to the stochastic encounter of operator molecules with possible activity sites, the simulator will have to provide means for simulating various error modalities (e.g., misidentification of configurations).

Since the outcome of a molecular process is not entirely determined, but may be quite complex, it is valuable to be able to visualize the result. Since LSO-based assembly takes place on a fixed lattice, it appears that it will be relatively straightforward to automatically translate the result of a simulation into the input of a ray-tracing visualization program (such as POV-ray). By this means we hope to get informative visualization without a large investment in software development.

Determine feasibility of DNA-based implementation: We plan a feasibility study of possible molecular implementations of the LSO approach. As for the combinatory programming approach, we expect to use covalently-structured molecular building blocks identified and connected by hydrogen bonds. However, a molecular implementation of the LSO approach should be simpler, since the operators are more local in the patterns to be matched and in the substitutions to be effected. Our investigation will include, specifically, a DNA-based approach, using ligation and cleavage by restriction enzymes for replacement of building blocks.

4.3.1 Explore Other Suitable Models of Computation

Although the combinatory and LSO approaches seem to be most promising at this time, we intend to remain open to other models of computation that may be compatible with the molecular context [PRS98]. If any of these can be identified, we will devote some time to their preliminary investigation. If they are sufficiently promising, we may redirect a more substantial fraction of our effort to them from the combinatory or LSO investigations.

5 Concluding Summary

By way of summary, we would like to call attention to some of the novel characteristics of this research, past and projected.

First, our project is oriented toward two nontraditional models of computation. Although computer scientists have already investigated combinatory logic as a model of computation (and we are exploiting that research), it was in the context of conventional (albeit parallel) digital computation. In contrast, we are investigating it as a model of autonomous molecular computation, especially applied to nanotechnology. Similarly, the lattice-swarm model has been used on conventional digital computers as a model of wasp nest building, but we intend to extend it into a model of autonomous molecular computation suitable for nanotechnological applications. In both cases we

are focusing on nontraditional models because they are a better match with molecular processes.

Second, our research is interdisciplinary in that it builds on theories from mathematical logic (combinatory logic), chemical nanotechnology (molecular building blocks, enzymes), DNA technology (hydrogen bond-based recognition, ligation, and cleavage), and biology (lattice-swarm model of wasp nest building). In particular, we believe the technology of autonomous molecular computation has much to learn from processes of self-organization, asynchronous parallelism, and robust behavior in complex systems, especially in colonies or other groups of organisms. In our continuing investigation of molecular combinatory computing, we will use concepts from cell biology, while in the proposed investigation of the lattice-swarm oriented model, we will use theories of self-organized assembly in social insects.

This research is admittedly high-risk; both approaches depend on our eventual ability to synthesize the appropriate molecular building blocks and enzyme-like compounds. There are also critical issues to be resolved (as described in the body of this report). However, the risks are justified by the potential benefit, for either approach would allow the synthesis of materials, with a desired structure or behavior, almost as easily as a program can be written. Nanotechnology would be based on a much more systematic foundation than it has been.

A Appendix

For completeness, we include the definitions of all combinators used in this report (except the primitives K, N, S, $\check{\mathsf{S}}$, and $\check{\mathsf{Y}}$). For additional explanation, see the combinatory logic literature (e.g., [CFC58]) as well as our previous reports [Mac02d, Mac02b, Mac03b]. For convenience, the composition operator may be used as an abbreviation for the B combinator: $X \circ Y = \mathsf{B}XY$. In combinatory logic, omitted parentheses are assumed to nest to the left, so for example $\mathsf{S(KS)K} = ((\mathsf{S(KS)})\mathsf{K})$.

$$\begin{aligned}
\mathsf{B} &= \mathsf{S(KS)K} \\
\mathsf{C} &= \mathsf{B(BS)} \\
\mathsf{I} &= \mathsf{SKK} \\
\check{\mathsf{S}}_1 &= \check{\mathsf{S}} \\
\check{\mathsf{S}}_{n+1} &= \mathsf{B}\check{\mathsf{S}}_n \circ \check{\mathsf{S}} \\
\mathsf{W} &= \mathsf{CSI} \\
\check{\mathsf{W}} &= \mathsf{C}\check{\mathsf{S}}\mathsf{I} \\
\mathsf{Z}_0 &= \mathsf{KI} \\
\mathsf{Z}_{n+1} &= \mathsf{SBZ}_n \\
\check{\Phi}_n &= \check{\mathsf{S}}_n \circ \mathsf{K}
\end{aligned}$$

For any X,

$$\begin{aligned}
X^1 &= X \\
X^{n+1} &= X \circ X^n \\
X_{(0)} &= X \\
X_{(n+1)} &= \mathsf{B}X_{(n)} \\
X_{[1]} &= X \\
X_{[n+1]} &= \mathsf{B}X_{[n]} \circ X \\
X^{[1]} &= X \\
X^{[n+1]} &= X \circ \mathsf{B}X^{[n]}
\end{aligned}$$

References

[Adl94] L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 226:1021–1024, 1994.

[And04] Alex Andriopoulos. Development of a simulator for universally programmable intelligent matter: Progress on universally programmable intelligent matter — UPIM report 8. Technical Report CS-04-519, Dept. of Computer Science, University of Tennessee, Knoxville, 2004.

[AS03] Damian G. Allis and James T. Spencer. Nanostructural architectures from molecular building blocks. In William A. Goddard, Donald W. Brenner, Sergey Edward Lyshevski, and Gerald J. Iafrate, editors, *Handbook of Nanoscience, Engineering, and Technology*, chapter 16. CRC Press, 2003.

[Bas03] Rashid Bashir. Biologically mediated assembly of artificial nanostructures and microstructures. In William A. Goddard, Donald W. Brenner, Sergey Edward Lyshevski, and Gerald J. Iafrate, editors, *Handbook of Nanoscience, Engineering, and Technology*, chapter 15. CRC Press, 2003.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Intelligence*. Santa Fe Institute Studies in the Sciences of Complexity. Addison-Wesley, New York, 1999.

[Bur75] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, Reading, 1975.

[CFC58] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic, Volume I*. North-Holland, Amsterdam, 1958.

[CLDF+01] Scott Camazine, Jean Louis Deneubourg, Nigel R. Franks, James Sneyd, Guy Theraulaz, and Eric Bonabeau. *Self-Organization in Biological Systems*. Princeton University Press, Princeton, 2001.

[KdSS99] T. Ross Kelly, Harshani de Silva, and Richard A. Silva. Unidirectional rotary motion in a molecular system. *Nature*, 401:150–2, 1999.

[KZvD+99] Nagatoshi Koumura, Robert W. J. Zijlstra, Richard A. van Delden, Nobuyuki Harada, and Ben L. Feringa. Light-driven monodirectional molecular rotor. *Nature*, 401:152–5, 1999.

[Lip95] R. J. Lipton. Using DNA to solve NP-complete problems. *Science*, 268:542–545, 1995.

[LWR00] Thomas H. LaBean, Eric Winfree, and John H. Reif. Experimental progress in computation by self-assembly of DNA tilings. In Eric Winfree and David K. Gifford, editors, *Proc. DNA Based Computers V*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Providence, RI, 2000. American Mathematical Society.

[Mac90] Bruce J. MacLennan. *Functional Programming: Practice and Theory.* Addison-Wesley, Reading, 1990.

[Mac97] Bruce J. MacLennan. Preliminary investigation of random SKI-combinator trees. Technical Report CS-97-370, Dept. of Computer Science, University of Tennessee, Knoxville, 1997.

[Mac02a] Bruce J. MacLennan. Membranes and nanotubes: Progress on universally programmable intelligent matter — UPIM report 4. Technical Report CS-02-495, Dept. of Computer Science, University of Tennessee, Knoxville, 2002.

[Mac02b] Bruce J. MacLennan. Molecular combinator reference manual — UPIM report 2. Technical Report CS-02-489, Dept. of Computer Science, University of Tennessee, Knoxville, 2002.

[Mac02c] Bruce J. MacLennan. Replication, sharing, deletion, lists, and numerals: Progress on universally programmable intelligent matter — UPIM report 3. Technical Report CS-02-493, Dept. of Computer Science, University of Tennessee, Knoxville, 2002.

[Mac02d] Bruce J. MacLennan. Universally programmable intelligent matter (exploratory research proposal) — UPIM report 1. Technical Report CS-02-486, Dept. of Computer Science, University of Tennessee, Knoxville, 2002.

[Mac02e] Bruce J. MacLennan. Universally programmable intelligent matter: Summary. In *IEEE Nano 2002 Proceedings*, pages 405–8. IEEE Press, 2002.

[Mac03a] Bruce J. MacLennan. Combinatory logic for autonomous molecular computation. In *Proceedings, 7th Joint Conference on Information Sciences*, pages 1484–1487. 2003 JCIS/Association for Intelligent Machinery, 2003.

[Mac03b] Bruce J. MacLennan. Molecular combinator reference manual. Technical report, Dept. of Computer Science, University of Tennessee, Knoxville, 2003. Latest edition available at http://www.cs.utk.edu/~mclennan/UPIM/CombRef.ps.

[Mac03c] Bruce J. MacLennan. Molecular combinatory computing for nanostructure synthesis and control. In *IEEE Nano 2003 Proceedings*, page 13_01. IEEE Press, 2003.

[Mac03d] Bruce J. MacLennan. Molecular implementation of combinatory computing for nanostructure synthesis and control: Progress on universally programmable intelligent matter — UPIM report 6. Technical Report CS-03-506, Dept. of Computer Science, University of Tennessee, Knoxville, 2003.

[Mac03e] Bruce J. MacLennan. Sensors, patches, pores, and channels: Progress on universally programmable intelligent matter — UPIM report 5. Technical Report CS-03-513, Dept. of Computer Science, University of Tennessee, Knoxville, 2003.

[Mac04a] Bruce J. MacLennan. Accomplishments and new directions for 2004: Progress on universally programmable intelligent matter — UPIM report 10. Technical Report CS-04-532, Dept. of Computer Science, University of Tennessee, Knoxville, 2004.

[Mac04b] Bruce J. MacLennan. Natural computation and non-Turing models of computation. *Theoretical Computer Science*, 317(1–3):115–145, June 2004.

[Mac04c] Bruce J. MacLennan. Radical reconfiguration of computers by programmable matter: Progress on universally programmable intelligent matter — UPIM report 9. Technical Report CS-04-531, Dept. of Computer Science, University of Tennessee, Knoxville, 2004.

[Mac04d] Bruce J. MacLennan. Simulator input language: Progress on universally programmable intelligent matter — UPIM report 7. Technical Report CS-04-518, Dept. of Computer Science, University of Tennessee, Knoxville, 2004.

[Macss] Bruce J. MacLennan. Combinatory logic for autonomous molecular computation. *Bio-inspired Computing*, in press.

[Mas02] Zachary Mason. Programming with stigmergy: Using swarms for construction. In Standish, Abbas, and Bedau, editors, *Artificial Life VIII*, pages 371–374. MIT Press, Cambridge, MA, 2002.

[MLRS00] Chengde Mao, Thomas H. LaBean, John H. Reif, and Nadrian C. Seeman. Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. *Nature*, 407:493–496, 2000.

[PRS98] G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms*. Springer-Verlag, Berlin, 1998.

[Rei02] John H. Reif. DNA lattices: A method for molecular-scale patterning and computation. *Computing in Science and Engineering*, 4(1):32–41, January/February 2002.

[RLS01] J. H. Reif, T. H. LaBean, and N. C. Seeman. Challenges and applications for self-assembled DNA nanostructures. In Anne Condon and Grzegorz Rozenberg, editors, *Proc. Sixth International Workshop on DNA-Based Computers, Leiden, The Netherlands, June, 2000*, Lecture Notes in Computer Science Vol. 2054, pages 173–198, Berlin, 2001. Springer.

[SSW91] Michel Simard, Dan Su, and James D. Wuest. Use of hydrogen bonds to control molecular aggregation. Self-assembly of three-dimensional networks with large chambers. *Journal of American Chemical Society*, 113(12):4696–4698, 1991.

[TB95a] Guy Theraulaz and Eric Bonabeau. Coordination in distributed building. *Science*, 269:686–688, 1995.

[TB95b] Guy Theraulaz and Eric Bonabeau. Modelling the collective building of complex architectures in social insects with lattice swarms. *Journal of Theoretical Biology*, 177:381–400, 1995.

[Win96] Eric Winfree. On the computational power of DNA annealing and ligation. In Richard Lipton and Eric B. Baum, editors, *Proc. DNA-Based Computers: April 4, 1995*, DIMACS Series in Discrete Mathematics and Computer Science Vol. 27, pages 199–211, Providence, RI, 1996. American Mathematics Society.

[WLWS98] E. Winfree, F. Liu, L. A. Wenzler, and N. C. Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394:539–544, 1998.

[WYS98] E. Winfree, X. Yang, and N. C. Seeman. Universal computation via self-assembly of DNA: Some theory and experiments. In Laura F. Landweber and Eric B. Baum, editors, *Proc. DNA Based Computers II, June, 1996*, DIMACS Series in Discrete Mathematics and Computer Science Vol. 44, pages 191–213, Providence, RI, 1998. American Mathematics Society.

[Yar00] Asim YarKhan. An investigation of random combinator soups. Technical report, Dept. of Computer Science, University of Tennessee, Knoxville, 2000. Unpublished report.

[YTM+00] Bernard Yurke, Andrew J. Turberfield, Allen P. Mills Jr, Friedrich C. Simmel, and Jennifer L. Neumann. A DNA-fuelled molecular machine made of DNA. *Nature*, 406:605–8, 2000.

[YYD+ed] Peng Yin, Hao Yan, Xiaoju G. Daniell, Andrew J. Turberfield, and John H. Reif. An autonomous DNA motor with unidirectional linear movement. *Science*, submitted.

[YZSS02] Hao Yan, Xiaoping Zhang, Zhiyong Shen, and Nadrian C. Seeman. A robust DNA mechanical device controlled by hybridization topology. *Nature*, pages 62–5, 2002.

[ZZRK96] S. C. Zimmerman, F. W. Zeng, D. E. C. Reichert, and S. V. Kolotuchin. Self-assembling dendrimers. *Science*, 271:1095, 1996.