A Formal Model for Databases in DNA

Joris J.M. Gillis* and Jan Van den Bussche

Hasselt University and Transnational University of Limburg,
Agoralaan Gebouw D, 3590 Diepenbeek, Belgium

Abstract. Our goal is to better understand, at a theoretical level, the database aspects of DNA computing. Thereto, we introduce a formally defined data model of so-called *sticker DNA complexes*, suitable for the representation and manipulation of structured data in DNA. We also define DNAQL, a restricted programming language over sticker DNA complexes. DNAQL stands to general DNA computing as the standard relational algebra for relational databases stands to general-purpose conventional computing. The number of operations performed during the execution of a DNAQL program, on any input, is only polynomial in the *dimension* of the data, i.e., the number of bits needed to represent a single data entry. Moreover, each operation can be implemented in DNA using a constant number of laboratory steps. We prove that the relational algebra can be simulated in DNAQL.

Keywords: DNA Computing, Formal Model, Relational Algebra.

1 Introduction

In DNA computing [16,3], data are represented using synthetic DNA molecules in vitro. Operations on data are performed by biotechnological manipulations of DNA that are based on DNA self-assembly (Watson–Crick base pairing) or on explicit effects upon DNA by specific enzymes. In the original approach to DNA computing, which we could call the Adleman model [2,5,18], one uses a more or less standard repertoire of operations on DNA, where each operation corresponds to a fixed number of steps in the laboratory. (These steps could be performed by a human or by a robot.)

In more recent years, research in DNA computing is largely focusing on the goal to let an entire computation happen by self-assembly alone, without (or with minimal) outside intervention, e.g., [23,22,11]. Whereas the pure self-assembly model is very attractive, it is harder to achieve in practice, and indeed this is the subject of a lot of current research in the area of DNA nanotechnology.

Meanwhile, the original Adleman model deserves further study, and in this paper we have a renewed look at the Adleman model, specifically from the perspective of databases. Indeed, DNA computing is very attractive from the database perspective: the nanoscale and robustness of DNA molecules are promising from

* Ph.D. fellowship of the Research Foundation - Flanders (FWO)

K. Horimoto, M. Nakatsu, and N. Popov (Eds.): ANB 2011, LNCS 6479, pp. 18–37, 2012.

a data storage point of view, and the highly parallel operations of DNA computing correspond well with the bulk data processing nature typical of database query processing [1]. Most earlier theoretical work on the possibilities of DNA computing has focused either on the ability to mimick classical models of computation such as finite automata or Turing machines, or on the relationship with parallel computation, but this always from a general-purpose computing perspective.

In contrast, in database theory, one considers restricted models of computation, limited in computational power but still with sufficient expressiveness for structured database manipulation. The classical model is the relational algebra for relational databases [1]. This algebra consists of six operations on relations (database tables): union; difference; cartesian product; selection; projection; and renaming. These operations can be composed to form expressions. These express database queries, and the relational algebra can express precisely those database queries that can be defined in first-order logic, thus providing a well-delineated restriction in computational power.

The benefit of restricted computational models is that they facilitate the identification of optimisation strategies for more efficient processing; hence there exists a large body of techniques for database query processing, e.g., [17]. From the point of view of theoretical science, an added benefit of a restricted computational model is that it allows us to study and attempt to characterise the precise computational abilities of the computational systems that are being modeled (such as relational database systems).

Motivated by the above considerations, in this paper, we want to propose a solution to the following equation:

$$\frac{\text{relational databases and relational algebra}}{\text{general-purpose conventional computing}} = \frac{?}{\text{DNA computing (Adleman model)}}$$

We define a formal data model of *sticker complexes*, which represent complexes of DNA molecules. Our complexes are general enough to serve as data structures for structured data such as found in relational databases. At the same time, however, sticker complexes are restricted so that we avoid the complications connected to the difficult secundary structure prediction problem of general DNA complexes [14]. Indeed, our main contribution consists in formally defining a well-behaved family of DNA-complex data structures, with an accompanying set of operations on these data structure that preserve the well-behavedness restrictions. We fit the operations into a first-order query language, called DNAQL, with a formal operational semantics. We thus propose the sticker complex data model, together with DNAQL, as the DNA computing analogues of the relational database model and the accompanying relational algebra. Restrictive as sticker complexes and DNAQL may be, we prove that they can still simulate the relational data model and the relational algebra. At the same time, we stress that our new DNA database model should also be appreciated in its own right as a restricted model of DNA computing specialised to database manipulation.

This paper is organised as follows. Section 2 discusses related work. Section 3 defines the data model. Section 4 introduces important operations on sticker

complexes. Section 5 discusses the representation of structured data using complexes. Section 6 discusses the implementation in DNA of the operations. Section 7 defines the query language DNAQL. Section 8 presentes the simulation of the relational algebra in DNAQL. We conclude in Section 9.

2 Related Work

Our work can be seen as a followup of Reif's original work [18] on relating DNA computing with conventional parallel computing. Indeed, Reif also formalized DNA complexes and considered similar operations. Our model specializes Reif's model to a database model. For example, it is well known [1,12] that the data complexity of the relational algebra (first-order logic) belongs to the parallel circuit complexity class AC_0, denoting constant-depth, polynomial-size circuits with unbounded fan-in. Likewise, DNAQL programs execute a number of operations on complexes that are largely independent of the data size, except for a polynomial dependence on the number of bits needed to represent a single data entry, a number we call the *dimension* of the data. Moreover, as usual for DNA computing, each operation works in parallel on the different DNA strands present in a complex, and each operation can be implemented in real DNA in a constant number of laboratory steps.

Our work also clearly fits in a recent trend in DNA computing to identify specialised computational models within the general framework of DNA computing. This trend is nicely exemplified by the work by Cardelli [6] and Majumder and Reif [15], where the specialised computational model is that of process algebras; in our work, it is that of databases.

While our work is not the first to relate the relational algebra with DNA computing, we are the first to do it formally and in detail. An abbreviated account of achieving relational algebra operations through DNA manipulation was given recently by Yamamoto et al. [26], but unfortunately that paper is too sketchy to allow any comparison with our approach. In contrast, our own methods are fully formalised, and importantly, our work identifies restrictions on DNA computing within which relational algebra simulation remains possible. More influential to our work is the older work by Arita et al. [4] demonstrating how one can accomplish concatenation and rotation of DNA strands. Such manipulations, which involve circular DNA, are crucial in our model, and indeed were already crucial to Reif [18].

Finally, we mention the earlier works on DNA memories [19,7], which, while having a database flavor, are primarily about supporting *searching* in sets of DNA strands and largely ignore the more complex operations of the relational algebra such as difference, projection, cartesian product, and renaming.

3 The Sticker-Complex Data Model

In this section we formally define a family of data structures which we call *sticker complexes*. They are an abstraction of complexes of DNA strands. Reif

[18] already defined a similar data structure, but our definition introduces several limitations so as to avoid unrealistic or otherwise complicated and unmanageable secundary structures. The adjective 'sticker' points to our restriction of hybridization to short primers (which we call "negative" strands) for the recognition and splicing of the strands carrying the actual data (called the "positive" strands).

Basically, we assume the following disjoint, finite alphabets: Λ of *atomic value symbols*; Ω of *attribute names*; and $\Theta = \{\#_1, \#_2, \#_3, \#_4, \#_5, \#_6, \#_7, \#_8, \#_9\}$ of *tags*. The union of these three alphabets is denoted by Σ and called the *positive alphabet*.

Furthermore, we use a *negative alphabet*, denoted $\overline{\Sigma}$, disjoint from Σ, defined as $\overline{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$. Thus there is a bijection between Σ and $\overline{\Sigma}$, which is called *complementarity* and is denoted by overlining a symbol; we set $\bar{\bar{a}} = a$.

We will first define *pre-complexes* that contain the overall structure of sticker complexes. Sticker complexes will then be defined as pre-complexes satisfying various restrictions. A pre-complex is a finite, edge-labeled directed graph where the edges represent bases in strands, and the nodes represent the endpoints between the bases in a strand. Moreover, a pre-complex is equipped with a matching, representing base pairing, and two predicates. One predicate indicates which bases are "immobilized", i.e., do not float freely and can be separated from solution in a controlled manner; the other predicate indicates which bases are "blocked", i.e., cannot participate in base pairing. Formally, a pre-complex is a 6-tuple $(V, E, \lambda, \mu, \mathit{immob}, \mathit{blocked})$ such that:

1. V is a finite set of nodes,
2. $E \subseteq V \times V$ is a finite set of directed edges without self-loops,
3. $\lambda : E \to \Sigma \cup \overline{\Sigma}$ is a total function labeling the edges,
4. $\mu \subseteq [E]^2 = \{\{e, e'\} \mid e, e' \in E \text{ and } e \neq e'\}$ is a partial matching on the edges, i.e., each edge occurs in at most one pair in μ,
5. $\mathit{immob} \subseteq E$,
6. $\mathit{blocked} \subseteq E$.

Let C be a pre-complex as above. We introduce the notion of "strand" and "component" of C as follows. A *strand* of C is simply a connected component of the directed graph (V, E). Furthermore, we say two strands s and s' are *bonded* if there exists some edge e in s and some edge e' in s' with $\{e, e'\} \in \mu$. When two strands are connected (possibly indirectly) by this bonding relation, we say they belong to the same component. Thus, a *component* of a pre-complex is a substructure formed by a maximal set of strands connected by the bonding relation.

A *sticker complex* now is a pre-complex satisfying the following restrictions:

1. There are no isolated nodes, i.e., each node occurs in at least one edge.
2. Each node has at most one incoming and at most one outgoing edge. Thus, each strand has the form of a chain or a cycle.
3. The labels on a chain are "homogeneous", in the sense that either all edges are labeled with positive symbols or all edges are labeled with negative

symbols. Naturally, a strand with positive (negative) symbols is called a positive (negative) strand.
4. Negative strands are severely restricted: specifically, every negative strand must be a chain of one or two edges.
5. Matchings by μ can only occur between complementarly labeled edges.
6. An edge can be immobilized only if it is the sole edge of a negative strand.
7. Edges in *blocked* do not occur in μ.
8. Each component can contain at most one immobilized edge.

Henceforth, for simplicity, we will refer to sticker complexes simply as "complexes".

We remark that the predicate *blocked* and the matching μ serve to abstract two different features of double-strandedness. The matching μ is used to make explicit where the stickers (short negative strands) pair with the positive strands. The predicate blocked represents longer stretches of double strands. As in the work by Rozenberg and Spaink [20], blocking is used to restrict the places where hybridization can still occur.

We also remark that it is not necessary to require that edges matched by μ run in opposite directions (in accordance with the opposite 5′–3′ and 3′–5′ directions of double-stranded DNA). This is because stickers of length one can trivially be placed in the desired direction, and stickers of length two can always fold so as to be again in the desired direction. The latter is illustrated in Figure 1.

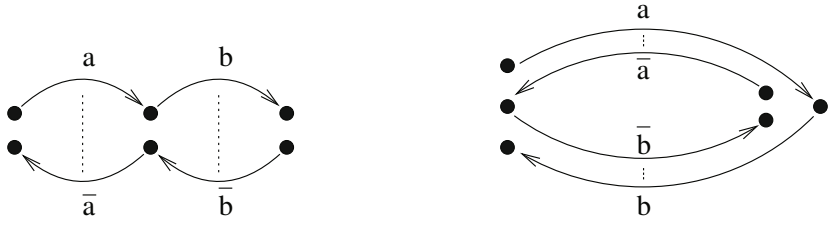

Fig. 1. On the left, a complex with two strands spelling the words ab and $\bar{b}\bar{a}$ and the expected complementary base pairing. On the right, a complex with two strands spelling the words ab and $\bar{a}\bar{b}$ and a "folded" base pairing. Dotted lines denote edges matched by μ.

Redundancy in complexes. In practice, a test tube will contain many duplicate strands, and indeed this multiplicity is typically crucial for DNA computing to work. Accordingly, in our model, each component of a complex stands for possibly multiple occurrences. (This important issue is not addressed in Reif's formalisation of complexes [18].) In order to formalize this, we define the notions of subsumption, equivalence, redundant extension, and minimality.

A complex C' is said to *subsume* a complex C if for each component D of C, there exists an isomorphic component D' in C'. Two complexes C and C' are

said to be *equivalent* if they subsume each other. When C' is equivalent to C and an extension of C, we call C' a *redundant extension* of C.

A component D of a complex C is called redundant if some other component of C is isomorphic to D. Note that removing a redundant component from C yields a complex that is still equivalent to C. A complex that has no redundant components is called *minimal*. Naturally, each complex C has a unique (up to isomorphism) minimal complex C' that is equivalent to C; we call C' the *minimization* of C.

4 Operations on Complexes

In this section, we formally define a set of operations on complexes that are rather standard in the DNA computing literature, except perhaps the difference. But what is interesting, however, is that we have defined sticker complexes in such a way that each operation always result in a sticker complex when applied to sticker complexes. Moreover, the difference operation imposes additional restrictions on its input so as to guarantee effective implementability in real DNA (discussed in Section 6).

As a general proviso, in the following definitions, a final minimization step should always be applied to the result so as to obtain a mathematically deterministic operation. In the following definitions we keep this implicit so as not to clutter up the presentation. Also, it is understood that the result of each operation is defined up to isomorphism.

Union. Let $C_1 = (V_1, E_1, \lambda_1, \mu_1, \mathit{immob}_1, \mathit{blocked}_1)$ and $C_2 = (V_2, E_2, \lambda_2, \mu_2, \mathit{immob}_2, \mathit{blocked}_2)$ be two complexes. W.l.o.g. we assume that V_1 and V_2 are disjoint. Then the union $C_1 \cup C_2$ equals $(V_1 \cup V_2, E_1 \cup E_2, \lambda_1 \cup \lambda_2, \mu_1 \cup \mu_2, \mathit{immob}_1 \cup \mathit{immob}_2, \mathit{blocked}_1 \cup \mathit{blocked}_2)$.

Difference. Let C_1 and C_2 be two complexes that satisfy the following conditions:

1. $\mu_1 = \mathit{immob}_1 = \mathit{blocked}_1 = \emptyset = \mu_2 = \mathit{immob}_2 = \mathit{blocked}_2$, i.e., all components in C_1 and C_2 are single strands.
2. All strands of C_1 and C_2 are positive, noncircular, and all have the same length.
3. Each strand of C_2 ends with $\#_4$ and does not contain $\#_5$.

Then the difference $C_1 - C_2$ equals the union of all strands in C_1 that do not have an isomorphic copy in C_2. If C_1 and C_2 do not satisfy the above conditions then $C_1 - C_2$ is undefined.

Hybridize. Let $C = (V, E, \lambda, \mu, \mathit{immob}, \mathit{blocked})$ and $C' = (V', E', \lambda', \mu', \mathit{immob}', \mathit{blocked}')$ be two complexes. We say that C' is a *hybridization extension* of C if $V = V'$, $E = E'$, $\lambda = \lambda'$, $\mathit{immob} = \mathit{immob}'$, $\mathit{blocked} = \mathit{blocked}'$ and μ' is an extension of μ. Beware that a hybridization extension must satisfy all conditions from the definition of sticker complex. A complex C' is said to have *maximal matching* if the only hybridization extension of C' is C' itself.

The notion of hybridization extension is not sufficient, however, since we want to allow duplicate copies of components in C to participate in hybridization. (This important issue is glossed over in Reif's formalisation [18].) To formalize this behavior, let us call C' (with matching μ') a *multiplying hybridization extension (MHE)* of C if C' is a hybridization extension, with maximal matching, of some redundant extension C'' of C. Moreover, we call a component D of an MHE *unfinished* if there exist another MHE in which D occurs bonded within a larger component. We then call an MHE *saturated* if it has no unfinished components. This is illustrated in Figure 2.

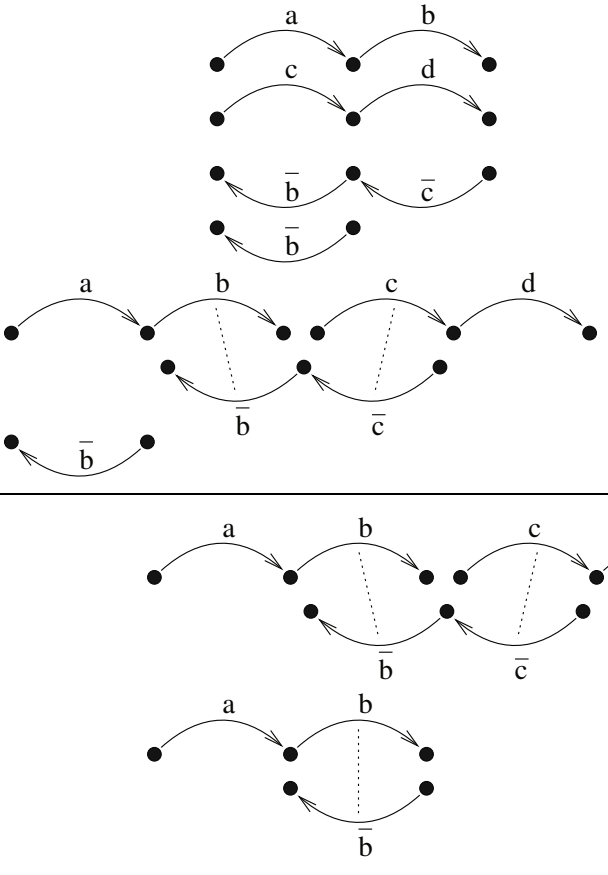

Fig. 2. Left: a complex C; top right: a hybridization extension of C with maximal matching, but not saturated in view of the MHE of C shown bottom right; that MHE is saturated. Dotted lines denote edges matched by μ.

Finally we say that C has *recursion-free hybridization* if there exists only a finite number of saturated hybridization extensions of C.

On the other hand, we do not want hybridization to go off into an uncontrolled chain reaction. Indeed, our very goal in this paper is to explore a "first-order" or "recursion-free" version of DNA computing, in line with the first-order nature of the relational algebra [1]. Thus we want to stay away from recursive self-assembly DNA computations. Formally, we want to rule out the situations where there are infinitely many possible non-equivalent MHE's. Such situations are very well possible. Consider, for a simple example, the complex C consisting of two non-circular strands spelling out the words ab and $\bar{a}\bar{b}$. Taking n copies of ab and n copies of $\bar{a}\bar{b}$, we can form arbitrary long non-equivalent MHE's of C. An illustration for $n = 3$ is given in Figure 3.

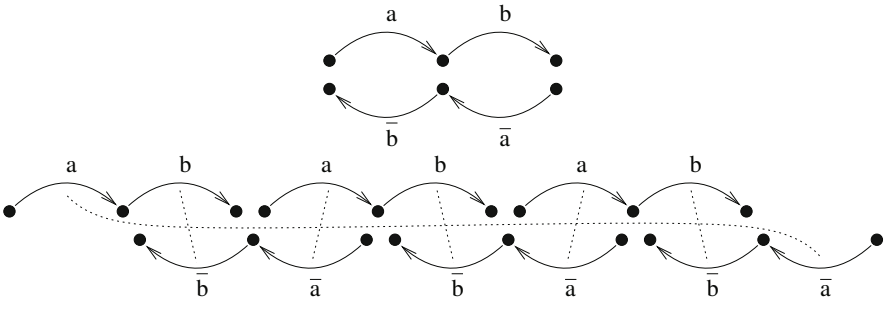

Fig. 3. A complex (top) and one of its MHE's (bottom). Dotted lines denote edges matched by μ. Note that the MHE forms a ring structure.

Formally, we say that C has *recursion-free hybridization* if their are only finitely many saturated MHE's of C. If this is the case, we define $\texttt{hybridize}(C)$ to equal the disjoint union of all saturated MHE's of C. If C does not have recursion-free hybridization, we consider $\texttt{hybridize}(C)$ to be undefined. For example, it can be verified that the complex from Figure 2 has recursion-free hybridization.

Ligate. The ligate operator concatenates strands that are held together by a sticker. Formally, define a *gap* as a set of four edges $\{e_1, e_2, e_3, e_4\}$ such that $\{e_1, e_4\} \in \mu$; $\{e_2, e_3\} \in \mu$; e_1 and e_2 (in that order) are consecutive edges on a negative strand; e_3 is the last edge on its (positive) strand; and e_4 is the first edge on its (positive) strand. By *filling a gap* we mean modifying the complex so that the endnode of e_3 and the startnode of e_4 are identified. We now define $\texttt{ligate}(C)$ as the complex obtained from C by filling all gaps.

Flush. Quite simply $\texttt{flush}(C)$ equals the complex obtained from C by removing all components that do not contain an immobilized edge.

Split. Consider a node u in some complex C. By *splitting C at u*, we mean the following.

- If u has an incoming (outgoing) edge, denote it by e_1 (e_2).
- If both e_1 and e_2 exist, then replace u by two nodes u_1 and u_2, letting e_1 arrive in u_1, and letting e_2 start in u_2.
- Furthermore, if there exists a node u' with incoming edge e_4 and outgoing edge e_3, such that $\{e_1, e_3\} \in \mu$ or $\{e_2, e_4\} \in \mu$, then u' is also split in an analogous manner.

Also, an edge is called *interacting* if it neither occurs in *blocked* nor in μ.

Now consider the set of triples shown in Table 1. Each triple is called a *split-point* and has the form (*label*, *interacting*, *place*). By splitting C at such a split-point, we mean splitting C at all startnodes (if *place* is 'before') or endnodes (otherwise) of edges labeled *label*, on condition that the edge is interacting (or noninteracting, depending on the boolean value *interacting*). The result is denoted by $\texttt{split}(C, \mathit{label})$.

Table 1. The allowed split points

Label	Interacting	Place
$\#_2$	*false*	before
$\#_3$	*false*	before
$\#_4$	*false*	after
$\#_6$	*true*	after
$\#_8$	*true*	before

Blocking. There are two blocking operations. Here we assume that C is "saturated" in the sense that C is equivalent to $\texttt{hybridize}(C)$; if this condition is not satisfied then the blocking operations on C are considered to be undefined.

The simplest operation is $\texttt{block}(C, \sigma)$, for any $\sigma \in \Sigma$, which equals the complex obtained from C by adding all edges labeled σ to *blocked*.

For the other operation, let again be $\sigma \in \Sigma$, and consider any contiguous substrand s in C. We call s a *σ-blocking range* if it satisfies three conditions. Firstly, all edges of the substrand are interacting (in the sense of the previous paragraph). Secondly, either the substrand contains the first edge of its strand, or the edge preceding the first edge of the substrand is blocked. Thirdly, the last edge of the substrand is labeled with σ. Now we define $\texttt{blockfrom}(C, \sigma)$ to be the complex obtained from C by adding to *blocked* all edges appearing in some σ-blocking range.

Cleanup. The cleanup operator undoes matchings and blockings and removes all strands except for the longest positive strands. Here we assume the condition that every positive strand in C is at least three long, and has at least one interacting edge; if C does not satisfy this condition, $\texttt{cleanup}(C)$ is not defined. Otherwise, $\texttt{cleanup}(C)$ equals the union of all positive strands of C of maximal length; there are no matched and no blocked edges in $\texttt{cleanup}(C)$.

5 Data Representation

When we want to represent structured data as sticker complexes, the symbols from the alphabet $\Sigma = \Lambda \cup \Omega \cup \Theta$ will be used in different ways. Attributes (Ω) will be used to indicate the structure of the data; tags (Θ) will be used as separators and auxiliary markers in data manipulation. Atomic value symbols (Λ) will be used to represent the actual data entries. However, since Λ is just a finite alphabet typically of small size, we will need to use strings (or vectors) of atomic value symbols to represent data entries, just like words of bits are used in conventional computing to represent data entries like characters or integers. In analogy to the word length of a conventional computer processor, in our approach we assume some *dimension* ℓ, a natural number, is known. Then every data entry is encoded by an ℓ-vector of atomic data symbols.

Formally, we say that a sticker complex C *has dimension* ℓ if every edge e labeled by some (positive) atomic value symbol is part of a sequence $(e_0, e_1, \ldots, e_\ell, e_{\ell+1})$ of $\ell + 2$ consecutive edges, where e_0 is labeled $\#_3$; each e_i for $i = 1, \ldots, \ell$ is labeled with a positive atomic value symbol; and $e_{\ell+1}$ is labeled $\#_4$. So, e is one of the e_i's with $i \in \{1, \ldots, \ell\}$. We call $(e_0, e_1, \ldots, e_\ell, e_{\ell+1})$ an *ℓ-vector* in C. A complex of dimension ℓ is also called an *ℓ-complex*.

We also introduce an additional blocking operator on ℓ-complexes. Let n be a natural number and let C be a complex satisfying the following conditions:

1. C is an ℓ-complex with $\ell \geq n$;
2. in every ℓ-vector in C, either all edges are blocked or no edge is blocked;
3. C is equivalent to $\texttt{hybridize}(C)$.

Then $\texttt{blockexcept}(C, n)$ equals the complex obtained from C by blocking, within each ℓ-vector $(e_0, e_1, \ldots, e_\ell, e_{\ell+1})$ that is not yet blocked, all edges except e_n. If (C, n) does not satisfy the conditions above, then $\texttt{blockexcept}(C, n)$ is undefined.

6 Implementation in DNA

In this section, we argue that the abstract sticker complexes and the operations on them presented above can be implemented by real DNA complexes. Our discussion remains theoretical as we have not performed laboratory experiments. On the one hand, our main purpose is to make the abstract model plausible as a theoretical framework in which the possibilities and limitations of DNA computing as a database model; on the other hand, we use only rather standard biotechnological techniques.

Each component of an abstract complex is represented by a large surplus of duplicate copies in DNA. Each positive alphabet symbol from Σ is implemented by a strand of (single-stranded) DNA, such that the resulting set of DNA strands forms a set of DNA codewords [8,21,24]. If the DNA strand for symbol $a \in \Sigma$ is w, then the DNA strand for the complementary symbol $\bar{a}$, is, naturally, the Watson-Crick complementary strand to w. Then, matching of edges by μ in an

abstract complex is implemented by base pairing in the DNA complex. We will see below how blocking is implemented. Immobilization is implemented as is standard in DNA computing by attachment to surfaces [13] or magnetic beads.

The union operation amounts to mixing two test tubes together.

The difference $C_1 - C_2$ of complexes can be implemented by a subtractive hybridization technique [10]. Let C_1 (C_2) be stored in test tube t_1 (t_2). Because all strands in t_2 end in $\#_4$, we can easily append $\overline{\#_5}$ to them. Next we add to t_2 an abundance of immobilized short primers $\overline{\#_5}$. Using polymerase we obtain complements to all strands in t_2, still immobilized, so that it is now easy to separate them. It remains to use these complements to remove all strands from t_1 that occured in t_2. Since all strands have the same length, partial hybridization, leading to false removals, can be avoided by using a very precise melting temperature based on the precise length of the strands.

Hybridization happens naturally and is merely controlled by temperature. Still, we must argue that the result still satisfies the definition of sticker complex. The only peculiarity in this respect is the requirement that each component can contain at most immobilized edge. Since immobilized edges are implemented by strands affixed to surfaces, implying some minimal distance between such strands, it seems reasonable to assume that the large majority of hybridization reactions will occur among freely floating strands, or between freely floating and immobilized ones.

Recursion-free hybridization is very hard to control by nature. It will be the responsability of the algorithm designer to design DNAQL programs (see Section 7) that, on the intended inputs, will apply `hybridize` only to inputs that have recursion-free hybridization. Our simulation of the relational algebra in DNAQL (see Section 8) is well-defined in this sense.

Splitting is achieved as usual by restriction enzymes. A feature of our abstract model is that we require only five recognition sites (Table 1). Of course, these recognition sites will have to be integrated in the DNA codeword design.

Blocking is implemented by making strands double-stranded, so that they cannot be involved in later hybridizations. The ordinary `block` operation can be implemented by adding the appropriate primer which will anneal to the desired substrands thus blocking the corresponding edges. As in the Sanger sequencing method, however, the base at the $3'$ end of the primer is modified to its dideoxy-variant. In this way unwanted interaction with polymerase from possible later `blockfrom` operations is avoided. Indeed, `blockfrom` is implemented using polymerase.

For the `blockexcept` operation to work, we need to adapt the implementation of ℓ-vector strands $\#3v_1 \ldots v_\ell\#_4$, with $v_i \in \Lambda$ for $i = 1, \ldots, \ell$, by introducing additional markers ϕ_i, so that we get $\#_3\phi_1 v_1 \ldots \phi_\ell v_\ell \#_4$. These ℓ additional markers must be part of the set of codewords. We can then implement $\texttt{blockexcept}(., n)$ by the composition $\texttt{block}(., \#_3); \texttt{blockfrom}(., \phi_{n-1}); \texttt{block}(., \phi_{n+1}); \texttt{blockfrom}(., \#_4)$.

The cleanup operation starts by denaturing (warming up) the tube. Immobilized strands are removed from the tube. Next a gel electrophoresis is carried out

to separate the longest DNA molecules from the other molecules. Thanks to the conditions we have imposed on inputs to cleanup, the result of this separation is either empty or consists of positive DNA molecules.

7 DNAQL

In this section we define a limited functional programming language, DNAQL, for expressing functions from ℓ-complexes to ℓ-complexes. A crucial feature of DNAQL is that the same program can be applied uniformly to complexes of any particular dimension ℓ. DNAQL is not computationally complete, as it is meant as a query language and not a general-purpose programming language. The language is based on the operations on complexes introduced earlier, and adds to this the following features: some distinguished constants; an emptiness test (if-then-else); let-variable binding; counters that can count up to the dimension of the complex; and a limited for-loop for iterating over a counter.

The syntax of DNAQL is given in Figure 4. Note that expressions can contain two kinds of variables: variables standing for complexes, and counters, ranging from 1 to the dimension. Complex variables can be bound by let-constructs, and counters can be bound by for-constructs. The free (unbound) complex variables of a DNAQL expression stand for its inputs. A DNAQL *program* is a DNAQL expression without free counters. So, in a program, all counters are introduced by for-loops.

$$
\begin{aligned}
\langle \mathit{expression} \rangle &::= \langle \mathit{complexvar} \rangle \mid \langle \mathit{foreach} \rangle \mid \langle \mathit{if} \rangle \mid \langle \mathit{let} \rangle \mid \langle \mathit{operator} \rangle \mid \langle \mathit{constant} \rangle \\
\langle \mathit{foreach} \rangle &::= \texttt{for}\ \langle \mathit{complexvar} \rangle := \langle \mathit{expression} \rangle\ \texttt{iter}\ \langle \mathit{counter} \rangle\ \texttt{do}\ \langle \mathit{expression} \rangle \\
\langle \mathit{if} \rangle &::= \texttt{if empty}(\langle \mathit{complexvar} \rangle)\ \texttt{then}\ \langle \mathit{expression} \rangle\ \texttt{else}\ \langle \mathit{expression} \rangle \\
\langle \mathit{let} \rangle &::= \texttt{let}\ x := \langle \mathit{expression} \rangle\ \texttt{in}\ \langle \mathit{expression} \rangle \\
\langle \mathit{operator} \rangle &::= ((\langle \mathit{expression} \rangle) \cup (\langle \mathit{expression} \rangle)) \mid ((\langle \mathit{expression} \rangle) - (\langle \mathit{expression} \rangle)) \\
&\quad \mid \texttt{hybridize}(\langle \mathit{expression} \rangle) \mid \texttt{ligate}(\langle \mathit{expression} \rangle) \\
&\quad \mid \texttt{flush}(\langle \mathit{expression} \rangle) \\
&\quad \mid \texttt{split}(\langle \mathit{expression} \rangle, \langle \mathit{splitpoint} \rangle) \\
&\quad \mid \texttt{block}(\langle \mathit{expression} \rangle, \Sigma) \\
&\quad \mid \texttt{blockfrom}(\langle \mathit{expression} \rangle, \Sigma) \\
&\quad \mid \texttt{blockexcept}(\langle \mathit{expression} \rangle, \langle \mathit{counter} \rangle) \\
&\quad \mid \texttt{cleanup}(\langle \mathit{expression} \rangle) \\
\langle \mathit{constant} \rangle &::= \Sigma^+ \mid \left(\overline{\Sigma} - \overline{\Lambda}\right)\left(\overline{\Sigma} - \overline{\Lambda}\right) \mid \texttt{immob}(\overline{\Sigma}) \\
&\quad \mid \texttt{leftboot} \mid \texttt{rightboot} \mid \texttt{empty} \\
\langle \mathit{splitpoint} \rangle &::= \#_2 \mid \#_3 \mid \#_4 \mid \#_6 \mid \#_8
\end{aligned}
$$

Fig. 4. Syntax of DNAQL

The constants have the following meaning as particular complexes:

- A word $w \in \Sigma^+$ stands for a single, linear, positive strand that spells the word w.

- A two-letter word $\bar{a}\bar{b}$, for $a, b \in \Sigma - \Lambda$, stands for a single, linear, negative strand of length two of the form $1 \xrightarrow{\bar{b}} 2 \xrightarrow{\bar{a}} 3$.
- `immob`$(\bar{a})$, for $a \in \Sigma$, stands for a single, negative, immobilized edge labeled $\bar{a}$.
- `leftboot` and `rightboot` are illustrated in Figure 5.
- `empty` stands for the empty complex, i.e., the complex with the empty set of nodes.

Fig. 5. Left- and right-boot-shaped complexes

The semantics of a DNAQL expression e is defined relative to a context consisting of a dimension ℓ, an *ℓ-complex assignment* β, and an *ℓ-counter assignment* γ. An ℓ-complex assignment is a mapping from complex variables to ℓ-complexes; an ℓ-counter assignment is a mapping from counters to $\{1, \ldots, \ell\}$. Naturally, β must be defined on all free variables of e, and γ must be defined on all free counters of e. Within such a context, the expression can evaluate to an ℓ-complex, denoted by $[\![e]\!]^{\ell}(\beta, \gamma)$. The semantic rules that define this evaluation are shown in Figure 6. The superscript ℓ has been omitted to reduce clutter. The rules for `let` and `for` use the oft-used notation $f[x := u]$ to denote the mapping f updated so that x is mapped to u. Because the operations on complexes are not always defined, the evaluation may fail, so $[\![e]\!]^{\ell}(\beta, \gamma)$ may be undefined. When e is a program, we denote $[\![e]\!](\beta, \emptyset)$ simply by $[\![e]\!](\beta)$.

8 Simulation of the Relational Algebra

Let us first recall some basic definitions concerning the relational data model. Basically we assume a universe U of data elements. A *relation schema* R is a finite set of attributes. A *tuple* over R is a mapping from R to U. A *relation* over R is a finite set of tuples over R. A *database schema* is a mapping D on some finite set of relation variables that assigns a relation schema to each relation variable. An *instance* of D is a mapping I on the same set of relation variables that assigns to each relation variable x a relation over $D(x)$.

The syntax of the relational algebra [1] is generated by the following grammar:

$$e ::= x \mid (e \cup e) \mid (e - e) \mid (e \times e) \mid \sigma_{A=B}(e) \mid \widehat{\pi}_A(e) \mid \rho_{A/B}(e) \ .$$

Here, x stands for a relation variable, and A and B stand for attributes. Our version of the relational algebra is slightly nonstandard in that our version of

$$\frac{x \text{ is a complex variable}}{[\![x]\!](\beta,\gamma) = \beta(x)} \qquad \frac{[\![e_1]\!](\beta,\gamma) = C_1 \qquad [\![e_2]\!](\beta,\gamma) = C_2}{[\![e_1 \cup e_2]\!](\beta,\gamma) = C_1 \cup C_2}$$

$$\frac{[\![e_1]\!](\beta,\gamma) = C_1 \qquad [\![e_2]\!](\beta,\gamma) = C_2 \qquad C_1 - C_2 \text{ is well-defined}}{[\![e_1 - e_2]\!](\beta,\gamma) = C_1 - C_2}$$

$$\frac{[\![e']\!](\beta,\gamma) = C'}{[\![\texttt{hybridize}(e')]\!](\beta,\gamma) = \texttt{hybridize}(C')} \qquad \frac{[\![e']\!](\beta,\gamma) = C'}{[\![\texttt{ligate}(e')]\!](\beta,\gamma) = \texttt{ligate}(C')}$$

$$\frac{[\![e']\!](\beta,\gamma) = C'}{[\![\texttt{flush}(e')]\!](\beta,\gamma) = \texttt{flush}(C')} \qquad \frac{[\![e']\!](\beta,\gamma) = C' \qquad \sigma \in \{\#_2, \#_3, \#_4, \#_6, \#_8\}}{[\![\texttt{split}(e', \sigma)]\!](\beta,\gamma) = \texttt{split}(C', \sigma)}$$

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad \texttt{block}(C', \sigma) \text{ is well-defined}}{[\![\texttt{block}(e', \sigma)]\!](\beta,\gamma) = \texttt{block}(C', \sigma)}$$

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad \texttt{blockfrom}(C', \sigma) \text{ is well-defined}}{[\![\texttt{blockfrom}(e', \sigma)]\!](\beta,\gamma) = \texttt{blockfrom}(C', \sigma)}$$

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad i \text{ is a counter} \qquad \texttt{blockexcept}(C', \gamma(i)) \text{ is well-defined}}{[\![\texttt{blockexcept}(e', i)]\!](\beta,\gamma) = \texttt{blockexcept}(C', \gamma(i))}$$

$$\frac{[\![e']\!](\beta,\gamma) = C' \qquad \texttt{cleanup}(C') \text{ is well-defined}}{[\![\texttt{cleanup}(e')]\!](\beta,\gamma) = \texttt{cleanup}(C')}$$

$$\frac{[\![e_1]\!](\beta,\gamma) = C_1 \qquad [\![e_2]\!](\beta[x := C_1],\gamma) = C_2}{[\![\texttt{let } x := e_1 \texttt{ in } e_2]\!](\beta,\gamma) = C_2}$$

$$\frac{[\![e_1]\!](\beta,\gamma) = C_1 \qquad \beta(x) \text{ is the empty complex}}{[\![\texttt{if empty}(x) \texttt{ then } e_1 \texttt{ else } e_2]\!](\beta,\gamma) = C_1}$$

$$\frac{[\![e_2]\!](\beta,\gamma) = C_2 \qquad \beta(x) \text{ is } \textit{not} \text{ the empty complex}}{[\![\texttt{if empty}(x) \texttt{ then } e_1 \texttt{ else } e_2]\!](\beta,\gamma) = C_2}$$

$$\frac{[\![e_1]\!](\beta,\gamma) = C_0 \qquad [\![e_2]\!](\beta[x := C_{n-1}], \gamma[i := n]) = C_n \text{ for } n = 1, \ldots, \ell}{[\![\texttt{for } x := e_1 \texttt{ iter } i \texttt{ do } e_2]\!](\beta,\gamma) = C_\ell}$$

Fig. 6. Semantics of DNAQL

projection ($\widehat{\pi}$) projects away some given attribute, as opposed to the standard projection which projects on some given subset of the attributes.

The semantics of the relational algebra is well known and we omit a formal definition. A relational algebra expression e can be evaluated in the context of some database instance I that is defined on at least the relation variables occurring in e. When the evaluation succeeds, e evaluates to a relation denoted by $[\![e]\!](I)$.

(The evaluation of a relational algebra operator may fail due to mismatches between the attributes present in the argument relations and the attributes expected by the operator [25].)

We want now to represent relations by complexes. We will store data elements as vectors of atomic value symbols. So formally, we use Λ^* as our universe U. Then a tuple t (relation r, instance I) is said to be of dimension ℓ if all data elements appearing in t (r, I) are strings of length ℓ. Let t be a tuple of dimension ℓ over relation schema R. We may assume a fixed order on the attributes of R, say, $A, \ldots, B$. We then represent t by the following ℓ-complex: (using the constant notation of DNAQL)

$$\mathit{complex}(t) = \#_2 A \#_3 t(A) \#_4 \ldots \#_2 B \#_3 t(B) \#_4 \ .$$

A relation r of dimension ℓ is then represented by the ℓ-complex $\bigcup\{\mathit{complex}(t) \mid t \in r\}$ which we denote by $\mathit{complex}(r)$. Moreover, a database instance I of dimension ℓ can be represented by the ℓ-complex assignment $\mathit{complex}(I)$ that maps each relation variable x (used as a complex variable) to $\mathit{complex}(I(x))$.

We are now in a position to state our main theorem.

Theorem 1. *Let some database schema D be fixed. Every relational algebra expression e can be translated into a DNAQL program e^{DNA}, such that for each natural number ℓ and for each ℓ-dimensional database instance I over D, if $[\![e]\!](I)$ is defined, then so is $[\![e^{DNA}]\!]^\ell(\mathit{complex}(I))$, and*

$$\mathit{complex}\left([\![e]\!](I)\right) = [\![e^{DNA}]\!]^\ell(\mathit{complex}(I))$$

(up to isomorphism).

For the proof we introduce a few useful abbreviations. For $a, b \in \Sigma$, we use $\texttt{blockfromto}(x, a, b)$ to abbreviate $\texttt{blockfrom}(\texttt{block}(x, b), a)$. For attributes A and B, we use $\mathit{circularize}(x, A, B)$ to abbreviate

$$\begin{aligned}&\texttt{cleanup}(\texttt{ligate}(\texttt{hybridize}(\\ &\qquad \texttt{hybridize}(\texttt{blockfromto}(x_6, B, A) \cup \mathit{immob}(\overline{\#_3})) \\ &\qquad\qquad \cup \overline{\#_4\#_2}))) \ .\end{aligned}$$

If x holds a complex of the form $\mathit{complex}(r)$ for some relation r over a schema with first attribute A and last attribute B, then $\mathit{circularize}(x, A, B)$ will equal the complex obtained from x by circularizing every strand [18,4].

The proof now goes by induction on the structure of e.

Union, difference. If e is $e_1 \cup e_2$, then $e^{DNA} = e_1^{DNA} \cup e_2^{DNA}$. If e is $e_1 - e_2$, then $e^{DNA} = e_1^{DNA} - e_2^{DNA}$.

Cartesian product. Let e be of the form $e_1 \times e_2$ with e_1 over relation schema R and e_2 over a disjoint relation schema S. Let A be the first and B be the last

attribute of R and let C be the first and D be the last attribute of S. Consider the following DNAQL program e':

$$\texttt{let } x := e_1^{DNA} \texttt{ in let } y := e_2^{DNA} \texttt{ in}$$
$$\texttt{if empty}(x) \texttt{ then empty else if empty}(y) \texttt{ then empty else } e_4$$

where e_4 is given by the following:

$$e_4 := \texttt{cleanup}(\texttt{split}(\texttt{split}(\texttt{blockfromto}(e_5, B, C), \#_2), \#_4))$$
$$e_5 := \textit{circularize}(e_6, A, D)$$
$$e_6 := \texttt{cleanup}(\texttt{ligate}(\texttt{hybridize}[x_6^a \cup x_6^b \cup \overline{\#_5\#_1}]))$$
$$x_6^a := \texttt{cleanup}(\texttt{ligate}(\texttt{hybridize}(x \cup \texttt{rightboot})))$$
$$x_6^b := \texttt{cleanup}(\texttt{ligate}(\texttt{hybridize}(y \cup \texttt{leftboot})))$$

Parts e_6^a and e_6^b attach a unique ending (beginning) to the tuples in r (s). The new tuples are added together, in x_6, along with a *sticky bridge* ($\overline{\#_5\#_1}$), resulting in all possible joins of tuples of e_1^{DNA} and e_2^{DNA}. The rest of the expression is concerned with cutting out the $\#_5\#_1$ piece in the middle of the new chains and getting the "old" e_1^{DNA}-tuples back in front of the "new" tuples.

The program e' is not yet quite correct, however, since we assume that the attributes in complex representations of tuples are ordered in lexicographical order. This order may be disrupted by joining tuples from e_1^{DNA} and e_2^{DNA}. Therefore it is necessary to reorder the attribute-value pairs within each tuple resulting from e^{DNA}. Shuffling attribute-value pairs around in a tuple is done using a new technique we call *double bridging*. Instead of using a single sticky bridge, two sticky bridges are hybridized onto one chain. A careful placement of the bridges allows us to cut twice in the chain without separating parts from the chain. Moreover, the two bridges guide the chain into its new conformation.

Next we describe (in outline) a DNAQL program for shuffling some attribute C to the end of a chain. Assume that A is the first attribute, attribute B occurs just in front of C, C is the attribute that we want to move, D occurs exactly after C and E is the last attribute of the chain. The general outline of the program is:

1. Insert the first marker ($\#_6\#_7$) between attributes B and C.
2. Insert the second marker ($\#_8\#_9$) between attributes C and D.
3. Insert the third marker ($\#_9\#_1$) at the end of the chain.
4. Add the two bridges to the mix: $\overline{\#_6\#_8}$ and $\overline{\#_1\#_7}$.
5. Cut at $\#_6$ and $\#_8$ and ligate the resulting complex.
6. Remove the markers from the chains.

An illustration is in Figure 7. A detailed DNAQL program to do these steps will have a similar structure to program e'.

Projection. Let e be of the form $\widehat{\pi}_C(e_1)$, where the relation schema of e_1 is R. Assume that B is the attribute just in front of C and D is the attribute just

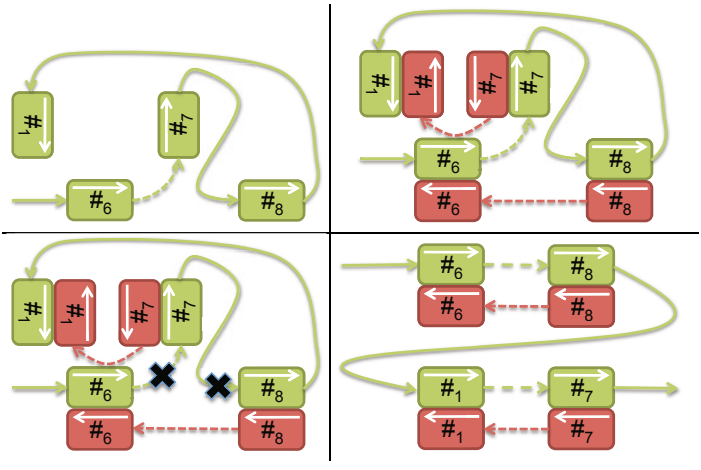

Fig. 7. Illustration of steps 1–3 (top left); step 4 (top right); and step 5 (bottom left, which simplifies to bottom right) described in the proof of simulation of Cartesian product

after attribute C. In the case that attribute C is the first attribute of the relation schema R, B is the last attribute of R. Likewise in the case that attribute C is the last attribute of R, then D is the first attribute of R. We thus perceive R to be circular. Assume that A and E are the first resp. last attribute of R.

We define e^{DNA} as the following program:

$$\texttt{let } x := e_1^{DNA} \texttt{ in if empty}(x) \texttt{ then empty else } f_1$$

where

$$f_1 := \texttt{cleanup}(\texttt{split}(\texttt{blockfromto}(\texttt{cleanup}(\texttt{ligate}(f_2)), E, A), \#_4))$$
$$f_2 := \mathit{circularize}(f_3, D, B)$$
$$f_3 := \texttt{cleanup}(\texttt{split}(\texttt{blockfromto}(\texttt{cleanup}(\texttt{ligate}(f_4)), B, D), \#_4))$$
$$f_4 := \mathit{circularize}(x, A, E)$$

Renaming. Let e be of the form $\rho_{C/F}(e_1)$, where R is the relation schema of e_1. Simulating renaming involves the following steps:

1. Rotate the chains to get attribute C at the start of each chain.
2. Cut the attribute from the chain, leaving the values of C on the chain.
3. Add the F attribute using *stickers*.
4. Rotate the chains again to get the first attribute at the start of each chain.

Assume that attribute B occurs just in front of C, D just after C, A is the first attribute of R and E is the last attribute. Then e^{DNA} is the following program:

$$\texttt{let } x := e_1^{DNA} \texttt{ in if empty}(x) \texttt{ then empty else } f_1$$

where

$$
\begin{aligned}
f_1 &:= \texttt{cleanup}(\texttt{split}(\texttt{blockfromto}(f_2, E, A), \#_4)) \\
f_2 &:= \texttt{cleanup}(\texttt{ligate}(\texttt{hybridize}[f_3 \cup \#_2 F \cup \overline{\#_4 \#_2} \cup \overline{F \#_3}])) \\
f_3 &:= \texttt{hybridize}(\texttt{split}(\texttt{blockfromto}(f_4, B, D), \#_3) \cup \texttt{immob}(\overline{\#_3})) \\
f_4 &:= \texttt{cleanup}(\texttt{split}(\texttt{blockfromto}(\texttt{cleanup}(\texttt{ligate}(f_5)), B, D), \#_2)) \\
x_5 &:= \mathit{circularize}(x, A, E)
\end{aligned}
$$

This program is not yet fully correct as attribute F may need to be shuffled into the right place. This can be done by repeatedly applying the shuffle procedure described in the case of cartesian product.

Selection. Let e be of the form $\sigma_{B=D}(e_1)$, where R is the relation schema of e_1. Translating the selection operator requires the most complicated expressions thus far. Assume that relation schema R has A as its first attribute, C following directly behind B, E following directly after D and F the last attribute of the schema. The Λ is fixed. The number of atomic value symbols is thus a constant; we denote them by v_1 to v_n. Note $A = B$, or $C = D$ or $D = E = F$ is possible; the program will still function correctly.

We define e^{DNA} as follows:

$$\texttt{let}\ x := e_1^{DNA}\ \texttt{in if empty}(x)\ \texttt{then empty else for}\ x_s := x\ \texttt{iter i do}\ e'$$

where

$$
\begin{aligned}
e' &:= \texttt{cleanup}(\texttt{split}(\texttt{blockfromto}(\\
&\qquad \texttt{let}\ x_c := \mathit{circularize}(x_s, A, F)\ \texttt{in}\ e'', F, A), \#_4)) \\
e'' &:= \mathit{select}^D_{v_1}(\mathit{select}^B_{v_1}(x_c)) \cup \cdots \cup \mathit{select}^D_{v_n}(\mathit{select}^B_{v_n}(x_c)) \\
\mathit{select}^B_a(x') &:= \texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(e_1^a(x')))) \\
e_1^a(x') &:= \texttt{blockexcept}(\texttt{blockfromto}(x', B, C), \texttt{i}) \cup \mathit{immob}(\overline{a}) \\
\mathit{select}^F_a(x') &:= \texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(e_2^a(x')))) \\
e_2^a(x') &:= \texttt{blockexcept}(\texttt{blockfromto}(x', D, E), \texttt{i}) \cup \mathit{immob}(\overline{a})
\end{aligned}
$$

9 Concluding Remarks

Many interesting questions remain open. A first issue is that an arbitrary DNAQL program may not evaluate on all possible inputs. We would like to have a type system by which programs can be statically typechecked to be safe on inputs of given types.

We would also like to better understand the expressive power of DNAQL. The relational algebra provides a lower bound on this expressive power. What is an upper bound? Can the semantics of DNAQL be defined in first-order logic? What is the computational complexity of DNAQL? Also, are all operations and

constructs of DNAQL really primitive in the language, or can some of them be simulated using the others?

Another interesting issue is the relationship between DNAQL and graph grammars. Furthermore, we could consider extensions, or restrictions, of DNAQL, just this has been done for the relational algebra. Extensions can lead to greater expressive power, while restrictions may lead to decidable static verification problems, such as testing the equivalence of DNAQL programs.

Finally, while we have gone to great efforts to design an abstraction that is as plausible as possible, of course, it would be great if it could be experimentally verified if DNAQL is workeable for practical DNA computing.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science 226, 1021–1024 (1994)
3. Amos, M.: Theoretical and Experimental DNA Computation. Springer, Heidelberg (2005)
4. Arita, M., Hagiya, M., Suyama, A.: Joining and rotating data with molecules. In: Proceedings 1997 IEEE International Conference on Evolutionary Computation, pp. 243–248 (1997)
5. Boneh, D., Dunworth, C., Lipton, R.J., Sgall, J.: On the computational power of DNA. Discrete Applied Mathematics 71, 79–94 (1996)
6. Cardelli, L.: Strand algebras for DNA computing. In: Deaton and Suyama [9], pp. 12–24
7. Chen, J., Deaton, R.J., Wang, Y.-Z.: A DNA-based memory with in vitro learning and associative recall. Natural Computing 4(2), 83–101 (2005)
8. Condon, A.E., Corn, R.M., Marathe, A.: On combinatorial DNA word design. Journal of Computational Biology 8(3), 201–220 (2001)
9. Deaton, R., Suyama, A. (eds.): DNA 15. LNCS, vol. 5877. Springer, Heidelberg (2009)
10. Diatchenko, L., Lau, Y.F., et al.: Suppression subtractive hybridization: a method for generating differentially regulated or tissue-specific cDNA probes and libraries. Proceedings of the National Academy of Sciences 93(12), 6025–6030 (1996)
11. Dirks, R.M., Pierce, N.A.: Triggered amplification by hybridization chain reaction. Proceedings of the National Academy of Sciences 101(43), 15275–15278 (2004)
12. Immerman, N.: Descriptive Complexity. Springer, Heidelberg (1999)
13. Liu, Q., Wang, L., et al.: DNA computing on surfaces. Nature 403, 175–179 (1999)
14. Lyngsø, R.B.: Complexity of Pseudoknot Prediction in Simple Models. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 919–931. Springer, Heidelberg (2004)
15. Majumder, U., Reif, J.H.: Design of a biomolecular device that executes process algebra. In: Deaton and Suyama [9], pp. 97–105
16. Paun, G., Rozenberg, G., Salomaa, A.: DNA Computing. Springer, Heidelberg (1998)
17. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill (2002)

18. Reif, J.H.: Parallel biomolecular computation: models and simulations. Algorithmica 25(2-3), 142–175 (1999)
19. Reif, J.H., LaBean, T.H., Pirrung, M., Rana, V.S., Guo, B., Kingsford, C., Wickham, G.S.: Experimental Construction of Very Large Scale DNA Databases with Associative Search Capability. In: Jonoska, N., Seeman, N.C. (eds.) DNA 2001. LNCS, vol. 2340, pp. 231–247. Springer, Heidelberg (2002)
20. Rozenberg, G., Spaink, H.: DNA computing by blocking. Theoretical Computer Science 292, 653–665 (2003)
21. Sager, J., Stefanovic, D.: Designing Nucleotide Sequences for Computation: A Survey of Constraints. In: Carbone, A., Pierce, N.A. (eds.) DNA 2005. LNCS, vol. 3892, pp. 275–289. Springer, Heidelberg (2006)
22. Sakamoto, K., et al.: State transitions by molecules. Biosystems 52, 81–91 (1999)
23. Seelig, G., Soloveichik, D., Zhang, D.Y., Winfree, E.: Enzyme-free nucleic acid logic circuits. Science 315(5805), 1585–1588 (2006)
24. Shortreed, M.R., et al.: A thermodynamic approach to designing structure-free combinatorial DNA word sets. Nucleic Acids Research 33(15), 4965–4977 (2005)
25. Van den Bussche, J., Van Gucht, D., Vansummeren, S.: A crash course in database queries. In: Proceedings 26th ACM Symposium on Principles of Database Systems, pp. 143–154. ACM Press (2007)
26. Yamamoto, M., Kita, Y., Kashiwamura, S., Kameda, A., Ohuchi, A.: Development of DNA Relational Database and Data Manipulation Experiments. In: Mao, C., Yokomori, T. (eds.) DNA12. LNCS, vol. 4287, pp. 418–427. Springer, Heidelberg (2006)