The Relational Completeness of the DNA Query Language DNAQL

ROBERT BRIJDER, M&CS Department, Eindhoven University of Technology, Eindhoven, Netherlands
JORIS GILLIS, Data Science Institute, Hasselt University, Hasselt, Belgium
JAN VAN DEN BUSSCHE, Data Science Institute, Hasselt University, Hasselt, Belgium

DNAQL is a query language for databases implemented as DNA molecules in solution, with well-known DNA wetlab procedures as operators. DNAQL focusses on faithful implementability in DNA, and, as a result, does not include powerful, but difficult to implement, features like nonterminating/recursive hybridization needed for general computation (i.e., Turing-completeness). In this article, we show that DNAQL is relationally complete, i.e., the full relational algebra can be performed by DNAQL programs.

CCS Concepts: • **Information systems** → **Relational database query languages**; • **Hardware** → **Biology-related information processing**; • **Theory of computation** → *Models of computation;*

Additional Key Words and Phrases: Query languages, molecular computing, relational algebra, DNA computing, relational completeness

ACM Reference Format:
Robert Brijder, Joris Gillis, and Jan Van den Bussche. 2026. The Relational Completeness of the DNA Query Language DNAQL. *ACM Trans. Datab. Syst.* 51, 4, Article 20 (April 2026), 30 pages. https://doi.org/10.1145/3779650

1 Introduction

As large artificial DNA-based storage is rapidly becoming a reality [5, 10, 11, 14, 16, 19, 20], there are significant efforts to equip this storage medium with efficient search capabilities [4, 15, 24, 30]. A natural next step is to increase the querying power from mere search (more precisely, selection based on equality or similarity) to allow typical database queries involving, for example, joins and projections. Of course, one way to accomplish this is by using one of the various Turing-complete models of computation studied in the rich literature on DNA computing [2, 22, 33]. However, database applications generally require much more modest computational power than Turing-completeness. Indeed, the relational algebra, the yardstick of database systems forming the core of the most popular database query language SQL, is equivalent to first-order definability and therefore significantly less powerful than Turing-complete models [1, 12, 17]. Large computational power comes with a cost, both with respect to feasibility of implementing the model and with respect to the ability to analyze its behavior (e.g., for model checking or query optimization). In particular, (unbounded) recursion, required for Turing-completeness, is very difficult to implement faithfully

Research was carried out in the position of Ph.D. fellow of the research foundation - flanders (FWO).
Authors' Contact Information: Robert Brijder (corresponding author), M&CS Department, Eindhoven University of Technology, Eindhoven, Netherlands; e-mail: r.brijder@tue.nl; Joris Gillis, Data Science Institute, Hasselt University, Hasselt, Belgium; e-mail: joris.gillis@gmail.com; Jan Van den Bussche, Data Science Institute, Hasselt University, Hasselt, Belgium; e-mail: jan.vandenbussche@uhasselt.be.

ACM 0362-5915/2026/04-ART20
https://doi.org/10.1145/3779650

in DNA. Therefore, it is desirable to devise computing models specifically targeted for database applications, in which the full relational algebra can be expressed (called relational completeness) while at the same time allowing for faithful implementability in DNA.

The DNA query language DNAQL [8, 9, 18] is specifically designed for database querying. DNAQL consists of a graph-based abstraction of the DNA species, called sticker complexes, that are present in a watery solution along with a number of operations on these sticker complexes that are inspired by well-known DNA wetlab procedures. Sticker complexes represent a finite number of species in a test tube, where we assume that DNA molecules of each of these species occur in arbitrary large multiplicity. As a consequence, DNA hybridization (a process in which DNA molecules can bind together) is not allowed to be recursive, as otherwise the result would violate the finiteness of the number of species. Fortunately, the property of hybridization being recursive for sticker complexes can be decided in polynomial time [7]. Apart from hybridization, DNAQL allows other well-known DNA wetlab procedures like mixing of test tubes, ligation, and gel electrophoresis.

With DNAQL in place, it is natural to compare the expressivity of DNAQL with the expressivity of the relational algebra. The main result of this article is to show that DNAQL is relationally complete. In fact, we show that this correspondence is computable by providing an explicit method translating a relational algebra expression into an equivalent DNAQL expression. As we do not allow recursive hybridization, the (proof of the) result carefully ensures that hybridization is kept in check. The most involved part of the construction, both theoretically and on the level of DNA, is expressing column reorderings in DNAQL, which is accomplished using a novel technique we call double bridging (cf. Section 5.8). While we are not aware of implementations of the double bridging technique in the wetlab, we provide several figures that support implementability in DNA. An attractive feature of the relational algebra is that its operators function independently of the specific data values in the relations. As a consequence of the main result, the DNAQL programs that simulate these relational algebra expressions are independent of the data values stored in the sticker complexes.

This article is organized as follows: In Section 2, we recall sticker complexes, its query language DNAQL, and sticker complex types. In Section 3, we recall the relational algebra, and in Section 4, we formulate the main result of this article: the simulation of the relational algebra by DNAQL. The proof of this result is given in Section 6. Following software engineering best practises, the proof uses helper functions with accompanying results, which are given in Section 5. Finally, a discussion is provided in Section 7.

2 DNAQL

In this section, we recall DNAQL [8, 18]. We first need to recall the notions of DNA complementarity and of a sticker complex and its operations.

2.1 DNA Complementarity and Hybridization

In this section, we recall DNA complementarity and related notions, see also [25].

Single strands of DNA are represented by strings over the alphabet $\{A, C, G, T\}$ of *(nucleotide) bases*. Bases A and T are *complementary*, as well as C and G, which means that they can form bonds. Due to complementarity, two single strands of DNA appearing in a watery solution can bond together in a process called *hybridization*. For example, hybridization of $TAAAGTC$ and $TAGACTT$ will obtain the following molecule (the bottom DNA strand should be read from right to left):

$$\begin{array}{c} TAAAGTC \\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots \\ TTCAGAT \end{array}$$

Here the two single DNA strands bond to each other in five base pairs (denoted by dotted lines). The bonding can be undone by raising the temperature to melting temperature in a process called *denaturation*.

We indicate complementarity by overlining, writing $\bar{A} = T$, $\bar{T} = A$, $\bar{C} = G$, and $\bar{G} = C$. Complementarity is extended to strings by complementing each base, then reversing the string (since the other should be read from right to left). Thus, $\overline{AAGTC} = \bar{C}\bar{T}\bar{G}\bar{A}\bar{A} = GACTT$. Note that for any string we have $\bar{\bar{s}} = s$. Hybridization of two complementrary DNA molecules obtains so-called double-stranded DNA.

The bases $\{A, C, G, T\}$ should be seen as the basic building block for data representation, just like bits (0 and 1) are in computer science. Since we often want to abstract from this low-level view, in practice we work with a larger alphabet, where every letter is encoded by a single DNA string called a *codeword*. The set of codewords should be well behaved, in that all codewords should have the same length, have similar melting temperatures, and should stick only to their own complement. Specifically, we do not want a codeword to stick to other codewords or their complements, or to concatenations of those. The design of "well behaved" DNA codewords is well studied [23, 28, 29]. From now on, we will understand that all finite alphabets are coded in DNA. So, a string over such a finite alphabet is to be understood as a concatenation of the DNA codewords corresponding to the sequence of letters in the string.

2.2 Sticker Complexes

In this section, we concisely recall the notion of sticker complex, a data structure that can be implemented in DNA [8, 9].

We assume a fixed finite alphabet Σ. The elements of Σ are called *positive symbols* and the elements of $\overline{\Sigma}$ are called *negative symbols*. For the purpose of data formatting and manipulation Σ is decomposed into three disjoint sets: the set Λ of *atomic value symbols*; the set Ω of *attribute names*; and the set $\Theta = \{\#_1, \ldots, \#_6, \#_{|a}, \#_{a|v}, \#_{v|}\}$ of *tags*. Tags represent DNA sequences reserved for special purposes, such as binding sites for enzymes.

The DNA structures that we want to consider are abstracted by the notion of a *sticker complex*. First we define the notion of a pre-complex. A pre-complex is a graph structure with nodes V denoting DNA codewords, one type of edges L encodes which DNA codewords corresponding to the nodes are consecutive in a DNA strand, and another type of edges μ encodes which DNA codewords are hybridized. Furthermore, nodes might be immobilized, meaning that they are affixed to a physical substrate (e.g., a DNA microarray) as opposed to pure free-floating molecules [21]. Due to this physical separation, two molecules that are immobilized cannot together engage in hybridization [3]. Finally, a substrand (that is, a contiguous subsequence of a strand) might be blocked, which means it is hybridized to a complementary strand that we otherwise don't want to explicitly model [27].

Formally, a *pre-complex* is a 6-tuple $C = (V, L, \lambda, \mu, \iota, \beta)$, where V is a finite set of nodes; $L \subseteq V \times V$ is a set of directed edges without self-loops; $\lambda : V \to \Sigma \cup \overline{\Sigma}$ is a function labeling each node with a positive or negative alphabet symbol; $\mu \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$ is a partial matching on the nodes, i.e., the sets of μ are mutually disjoint; $\iota \subseteq V$ is the set of *immobilized* nodes; and $\beta \subseteq V$ is the set of *blocked* nodes.

Let C be a pre-complex. A *strand* of C is a connected component of the directed graph (V, L) (so ignoring μ). A strand with only positive (negative, respectively) symbols is called a *positive (negative, respectively) strand*. The *length* of a strand is its number of nodes. Two strands s and s' are *bonded* if there is a node v in s and some node v' in s' with $\{v, v'\} \in \mu$. A maximal set of bonded strands, possibly indirectly bonded, is a *component* (i.e., a component is a connected component taking both L and μ into account).

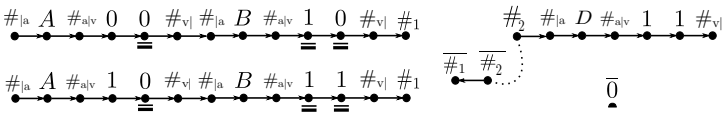

Fig. 1. A complex C.

Example 2.1. Let alphabet Σ contain the attribute names A, B, and D and the atomic value symbols 0 and 1. Figure 1 shows a pre-complex $C = (V, L, \lambda, \mu, \iota, \beta)$. Directed edges represent L, dashed edges represent μ, a double line under a node signifies that the node is blocked, and an immobilized node is depicted as half a dot (instead of a full dot). Note that C has four components: two positive strands, each containing two 2-vectors, one component containing two strands (one positive strand and one negative strand) connected by μ, and one component consisting of an immobilized node.

An *isomorphism* from pre-complex C_1 to pre-complex C_2 is a one-to-one correspondence α from the set of nodes of C_1 to the set of nodes of C_2 which respects the edges, labels of the nodes, the matching relation, immobilizations, and blockings. In this case, we say that C_1 and C_2 are *isomorphic*.

In molecular computing, and DNA computing in particular, it is usual to assume that if a molecule is present in a solution, then it is present in abundant quantity (indeed in chemistry molecule count is by default expressed in terms of Avogadro's number, which is about $6 \cdot 10^{23}$). Therefore, we consider pre-complexes equivalent when, roughly speaking, they are isomorphic up to isomorphic copies of components. In other words, we are interested in *DNA species* as opposed to individual DNA molecules of that species, and the DNA species that are present are represented by the components. We now give a precise definition of equivalence. We say that C_1 is *subsumed* by C_2 if for each component D_1 in C_1 there is an isomorphic component D_2 in C_2. Two pre-complexes are *equivalent* if they subsume each other. So two pre-complexes are equivalent if they are isomorphic after retaining for each of the two pre-complexes only one component for each equivalence class of components that are mutually isomorphic. Since we are only interested in pre-complexes up to equivalence, we allow definitions on pre-complexes to be well defined only up to equivalence (note that graph-theoretical notions are typically well defined up to the stricter notion of isomorphism).

We now define the notion of a sticker complex, which is a pre-complex with some additional constraints that either have an origin in biochemistry or form a convenient restriction of the type of DNA structures we want to allow as input.

Definition 2.2. A *sticker complex* (or *complex* for short) is a pre-complex satisfying the following:

(1) Each node has at most one incoming and one outgoing edge (of L). Thus each strand has the form of a chain or a cycle.
(2) Each strand is either positive or negative.
(3) Every negative strand has at most one edge.
(4) Matchings by μ only occur between nodes with complementary labels.
(5) Blocked nodes do not occur in μ.
(6) A node can be immobilized only if it is the sole node of a negative strand.
(7) Each component can contain at most one immobilized node.

Negative strands of a sticker complex are called *stickers*. Note that a sticker has length one or two, and if it has length two, then it cannot be a 2-cycle.

The requirements (1), (4), (5), and (7) in Definition 2.2 are biochemical restrictions and describe that DNA strands are (possibly cyclic) strands (1), hybridization only occurs between complementary codewords (4), a codeword cannot bond with two strands at the same time (5), and any

two immobilized nodes are physically separated far enough that no single molecule (i.e., component) can bridge (7). The other requirements are convenient restrictions on the input and aim at avoiding physically unrealistic states. Requirement (2) avoids hybridization of a strand with itself, Requirement (3) avoids a sticker to hybridize on three of more (potentially distant) positions. Requirement (6) models the fact that DNA microarrays consist of single strands of DNA, called probes, each separately attached to a surface. Therefore, we call a negative, immobilized node a *probe*. Note that the operations we will consider on sticker complexes (defined below) respect these restrictions in that their output will always be a sticker complex if their input is.

Notice that the pre-complex of Figure 1 is a sticker complex. A node u of C is called *free* if u occurs neither in β nor in μ, and is called *closed* if it is not free. Nodes u and v are called *mutually interacting* if they are complementary labeled and both free, and do not belong to distinct immobilized components (i.e., components containing an immobilized node).

Let $\ell \geq 2$ be an integer, and let $s = s_0 \dots s_{\ell+1}$ be a sequence of $\ell + 2$ consecutive nodes of a strand of a sticker complex C. Such a sequence is called an *ℓ-vector* of C if s_0 is labeled with tag $\#_{\mathrm{a|v}}$, $s_{\ell+1}$ is labeled with tag $\#_{\mathrm{v|}}$ and $s_1 \dots s_\ell$ are labeled with atomic value symbols. In this case, $s' = s_1 \dots s_\ell$ is called an *ℓ-core* of C. We say that C has *dimension* ℓ, if all nodes labeled with an atomic value symbol occur in an ℓ-vector. We then call C an *ℓ-complex*. Notice that the sticker complex of Figure 1 is, in fact, a 2-complex with two positive strands having two 2-vectors and one positive strand having one 2-vector.

2.3 Operations on Sticker Complexes

We now recall the sticker complex operations from [8], and we recall that each of these operations correspond to well-known wetlab operations.

Union. The union operation corresponds to mixing two watery solutions (in, for example, test tubes) together to obtain a single solution. Mathematically, union simply takes the disjoint union of the complexes. Let $C_1 = (V_1, L_1, \lambda_1, \mu_1, \iota_1, \beta_1)$ and $C_2 = (V_2, L_2, \lambda_2, \mu_2, \iota_2, \beta_2)$ be complexes. Without loss of generality, we assume that V_1 and V_2 are disjoint. Then the union $C_1 \cup C_2$ is the complex $(V_1 \cup V_2, L_1 \cup L_2, \lambda_1 \cup \lambda_2, \mu_1 \cup \mu_2, \iota_1 \cup \iota_2, \beta_1 \cup \beta_2)$, where $\lambda_1 \cup \lambda_2 : V_1 \cup V_2 \to \Sigma \cup \overline{\Sigma}$ is the function that restricts to λ_1 on V_1 and to λ_2 on V_2.

Hybridize. In Section 2.1, we recalled the process of DNA hybridization. After defining some auxiliary notions, we will formally define here hybridization as an operation on complexes. Let $C = (V, L, \lambda, \mu, \iota, \beta)$ be a complex. Hybridization extends μ. Therefore, a *hybridization extension* of C is a complex $C' = (V, L, \lambda, \mu', \iota, \beta)$ with $\mu \subseteq \mu'$. Note that the definition of a complex ensures all pairs in μ' are between complementary nodes.

Recall that the components of a complex represents the species of DNA molecules that occur in the watery solution. Since it may happen that several different molecules of the same species end up in a common output molecule, we need to consider copies of each component of a complex to faithfully represent hybridization. Let us call complex C' a *multiplying hybridization extension (MHE for short)* of C if C' is a hybridization extension of some complex C'' that is subsumed by C. Note that for each component D of C, complex C'' has zero, one, or more isomorphic copies of D.

While the number of DNA molecules in a test tube is typically very large, it is, of course, not infinite. Therefore, we do not want to allow infinite chain reactions that occur, for example, if the complex contains strands ab and $\bar{b}\bar{a}$. In other words, we only want to allow complexes that have a finite number of MHEs up to equivalence. If complex C has finitely many nonequivalent MHEs, then we say that C is *terminating*. It turns out that termination of complexes can be efficiently decided [7].

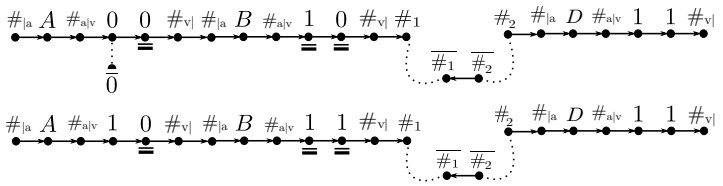

Fig. 2. The hybridization of the complex of Figure 1.

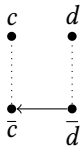

Fig. 3. A complex representing a double-stranded DNA molecule with a gap between the last nucleotide of c and the first nucleotide of d.

Hybridization will bond all complementary substrands that appear in the test tube. This is captured by the notion of finished we define now. A complex C' is called *unfinished* with respect to C if there exists a node u in C' and a node v in C such that u and v are mutually interacting; otherwise C' is called *finished* with respect to C.

We are now ready to formally define the notion of hybridize. If C is terminating, then $\texttt{hybridize}(C)$ is defined as a finished MHE of C that subsumes every finished MHE of C. In this case, $\texttt{hybridize}(C)$ is uniquely defined up to equivalence, which we recall is sufficient for our purposes. Of course, it is possible to alter the definition, avoiding isomorphic copies of components, to obtain a notion that is well defined up to isomorphism [8]. If C is not terminating, then $\texttt{hybridize}(C)$ is undefined.

Finally, we sometimes need the following specialization of the notion of finished, and so we give it a name. We say that C is said to be *saturated* if C is finished with respect to itself. In other words, the only hybridization extension of C is C itself.

Example 2.3. Consider again the complex of Figure 1. During hybridization, (a copy of) the component containing the sticker can match with (a copy of) one the two strands on the left in Figure 1. Also, (a copy of) the component consisting of one immobilized node can match with (a copy of) the strand on the top-left in Figure 1. The hybridization of this complex is shown in Figure 2.

Ligate. Consider the complex of Figure 3, and recall that each node represents a codeword and that the directed edge represents concatenation. Therefore, this complex forms a double-stranded DNA molecule representing cd up to a gap (also called nick) between the last nucleotide of c and the first nucleotide of d. The enzyme DNA ligase occurs in nature to repair such gaps in double-stranded DNA, and can be used in a wetlab setting as well. In general, the ligate operator concatenates strands that are held together by a sticker. Formally, define a *gap* g of a complex C as a set $\{n_1, n_2, n_3, n_4\}$ of four nodes of C such that, like in Figure 3, (1) $\{n_1, n_4\}, \{n_2, n_3\} \in \mu$, (2) n_1 and n_2 (in that order) are consecutive nodes on a negative strand, (3) n_3 is the last node on its strand, which must be positive, and (4) n_4 is the first node on its strand, which again must be positive. By *filling* the gap g we mean adding the directed edge (n_3, n_4) to the complex. We define $\texttt{ligate}(C)$ as the complex obtained from C by filling all gaps of C.

Flush. We consider the wetlab operation of washing away all DNA complexes that are not bonded to the DNA microarray, i.e., removing all free-flowing DNA complexes. So, for a complex C, we

Table 1. The Split Points

label	*free*	*place*
$\#_{\|a}$	*true*	before
$\#_{a\|v}$	*true*	before
$\#_{v\|}$	*true*	after
$\#_3$	*false*	after
$\#_5$	*false*	before

define `flush`(C) to be the complex obtained from C by removing all components that do not contain an immobilized node.

Difference. Let C_1 and C_2 be complexes such that (1) all components of C_1 and C_2 are positive, non-circular strands having the same length, and (2) each strand of C_2 ends with $\#_{v|}$ and does not contain $\#_2$. 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.

The wetlab technique called *subtractive hybridization* [13] is able to accomplish the difference operation. Using the DNA enzyme polymerase, complements of the strands in C_2 can be created, to which magnetic beads are attached (here we use the fact that the strands of C_2 have a common suffix and do not have a certain code word as a substrand). Hybridization will attach these strands to the complementary strands in C_1 when present. By applying a magnetic field, we can separate the strands of C_1 that did not engage in hybridization (without magnetic bead) from those that did engage in hybridization (with magnetic bead). We discard the strands with magnetic beads. This last part of the operation is similar to the flush operation in that both operations remove components from a complex. Partial hybridization, leading to false removals, can be avoided by controlling the temperate to just below the melting temperature of the strands (which we know have a common length, so similar melting temperature).

Split. A class of natural enzymes called *restriction enzymes* are used to split complexes at designated points that occur before or after specific substrands, called recognition sites. These natural enzymes have been successfully used in wetlab settings. Since the number of restriction enzymes is quite limited, we assume a limited number of recognition sites. In fact, we assume only five recognition sites.

The effect of restriction enzymes on complexes is captured by the following definition. Consider a node n of some complex C. By *splitting before* (resp. *after*) n, we mean the following.

— If n has a predecessor (resp. successor) m in its strand, then (m, n) (resp. (n, m)) is removed from L.
— Furthermore, if there exists a node n' such that $\{n, n'\} \in \mu$ and n' has a successor (resp. predecessor) m' in its strand, then (n', m') (resp. (m', n')) is removed from L.

Now, consider the set of triples shown in Table 1. Each such triple is called a *split point* and has the form (*label*, *free*, *place*). By splitting C at such a split point, we mean splitting C at all nodes labeled *label* (be it before or after, based on the value of *place*), on the condition that the node is free or closed, depending on the boolean value *free*. Here *label* represents a DNA codeword that contains the recognition site. Since the split points are uniquely determined by their label, we (may) denote the result by `split`(C, *label*).

Block. We now describe three operations that block a substrand of complex, that is, adds the nodes of the substrand to the set β of blocked nodes. Recall from above that a blocked substrand means that the substrand is hybridized to a complementary strand to form a double stand (for

didactical reasons, we have an explicit β instead of modelling this through adding substrands and expanding μ).

A substrand s can be made double stranded by hybridizing with a single stranded DNA molecule, called a primer, which is complementary to s. This translates to the following definition for complexes.

Let C be a saturated complex. The *block* operation on C with respect to $\sigma \in \Omega \cup \Theta$, denoted by $\texttt{block}(C, \sigma)$, is the complex obtained from C by turning all free nodes labeled with σ into blocked nodes (i.e., adding them to β).

Block-From. A primer (used in the block operation above) can form the starting point for an DNA enzyme called polymerase to initiate a chain reaction that makes all nodes after that point (with respect to L) double stranded. This process stops once a hybridized (not is, a closed) node is encountered or until the end of the strand is reached.

A substrand s can be made double stranded in the wetlab by adding a short single stranded DNA molecule, called a primer, to the solution which is complementary to the target prefix of s. The primer forms the starting point for the DNA enzyme polymerase for a chain reaction that makes the rest of the substrand double standed. This translates to the following definition, called `blockfrom`, for complexes.

Let C be a saturated complex. Let again $\sigma \in \Sigma$, and consider any contiguous substrand s of C. We call s a *σ-blocking range* if (1) all nodes of s are free, and (2) the last node of s is labeled with σ. We define $\texttt{blockfrom}(C, \sigma)$ to be the complex obtained from C by turning all nodes appearing in some σ-blocking range into blocked nodes (i.e., adding them to β). Note that this operation is called `blockfrom` instead of `blockto` because polymerase operates from σ in opposite direction [8].

Block-Except. Let $n \geq 1$ be a natural number and let C be a complex satisfying the following conditions:

(1) C is an ℓ-complex with $n \leq \ell$;
(2) in every ℓ-vector in C, either all nodes are free or all nodes are closed; and
(3) C is saturated.

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 nodes except e_n. If (C, n) does not satisfy the conditions above, then $\texttt{blockexcept}(C, n)$ is undefined.

The block-except operation can be accomplished in the wetlab by implementing ℓ-vectors $\#_{\mathrm{a|v}} v_1 \ldots v_\ell \#_{\mathrm{v|}}$ using ℓ additional codewords ϕ_i to obtain $\#_{\mathrm{a|v}} \phi_1 v_1 \ldots \phi_\ell v_\ell \#_{\mathrm{v|}}$. These ℓ additional codewords should be seen as part of the set of codewords. We can then implement $\texttt{blockexcept}(C, n)$ as $\texttt{blockfrom}(\texttt{block}(\texttt{blockfrom}(\texttt{block}(C, \#_{\mathrm{a|v}}), \phi_{n-1}), \phi_{n+1}), \#_{\mathrm{v|}})$. For all other purposes the ϕ_i codewords can be treated transparently, so for notational convenience we consider $\texttt{blockexcept}_i$ to be a primitive operation rather than an abbreviation.

Cleanup. The `cleanup` operator undoes matchings and blockings and retains only the longest positive strands. If the input complex does not contain positive strands, then `cleanup` obtains the empty complex.

This operation can be accomplished in the wetlab as follows: First by raising the temperature denaturation occurs in which hybridization bonds are undone. The biotechnology called gel electrophoresis is able to separate DNA strands according to their length. In this way, the largest strands can be separated from the other strands. Since negative strands consist of two nodes, they rarely are the longest. To cover this cornercase, we can immobilize all negative strands by using a DNA microarray that contains all positive alphabet symbols and then keep the free-flowing (necessarily positive) strands.

2.4 DNAQL

The ***DNA query language*** (**DNAQL**) [8] is an applicative programming language for expressing transformations of ℓ-complexes, independent of ℓ. The main features of DNAQL are the operations of Section 2.3, a (limited) for-loop construct to iterate over the elements of ℓ-cores, and an emptiness test (if-then-else).

The syntax of DNAQL expressions is given below.

$$
\begin{array}{rcl}
\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)) \\
& \mid & \texttt{hybridize}(\langle \mathit{expression} \rangle) \mid \texttt{ligate}(\langle \mathit{expression} \rangle) \\
& \mid & \texttt{flush}(\langle \mathit{expression} \rangle) \mid \texttt{split}(\langle \mathit{expression} \rangle, \langle \mathit{splitpoint} \rangle) \\
& \mid & \texttt{block}(\langle \mathit{expression} \rangle, \Sigma \setminus \Lambda) \mid \texttt{blockfrom}(\langle \mathit{expression} \rangle, \Sigma \setminus \Lambda) \\
& \mid & \texttt{blockexcept}(\langle \mathit{expression} \rangle, \langle \mathit{counter} \rangle) \mid \texttt{cleanup}(\langle \mathit{expression} \rangle) \\
\langle \mathit{constant} \rangle & ::= & \Sigma^+ \mid \left(\overline{\Sigma} \setminus \overline{\Lambda}\right)\left(\overline{\Sigma} \setminus \overline{\Lambda}\right) \mid \texttt{immob}(\overline{\Sigma}) \mid \texttt{empty} \\
\langle \mathit{splitpoint} \rangle & ::= & \#_{|\mathrm{a}} \mid \#_{\mathrm{a}|\mathrm{v}} \mid \#_{\mathrm{v}|} \mid \#_3 \mid \#_5
\end{array}
$$

The constant expressions represent specific complexes defined (up to isomorphism) as follows: A string $w \in \Sigma^+$ represents a linear (positive) strand s where $w = \lambda(v_1) \cdots \lambda(v_n)$ and $(v_1, \ldots, v_n)$ is the unique path of s containing all nodes of s. A two-letter string $\bar{a}\bar{b}$, for $a, b \in \Sigma \setminus \Lambda$, represents a sticker of the form $1 \to 2$ with $\lambda(1) = \bar{b}$ and $\lambda(2) = \bar{a}$. The expression $\texttt{immob}(\bar{a})$, for $a \in \Sigma$, represents a probe labeled $\bar{a}$. The expression `empty` represents the empty complex.

Note that expressions can contain two kinds of variables: complex variables ($\langle \mathit{complexvar} \rangle$), and counters ($\langle \mathit{counter} \rangle$). Complex variables can be bound by let-constructs ($\langle \mathit{let} \rangle$), and counters can be bound by for-constructs ($\langle \mathit{foreach} \rangle$). The set of free (unbound) complex variables of a DNAQL expression e is denoted by $FV(e)$. A DNAQL *program* is a DNAQL expression without free counters. So, in a program, all counters are bound by for-loops.

Let $\ell \geq 2$ be an integer. An *ℓ-complex assignment* ν is a mapping from a finite set of complex variables, $dom(\nu)$, to ℓ-complexes, and an *ℓ-counter assignment* γ is a mapping from a finite set of counters, $dom(\gamma)$, to $\{1, \ldots, \ell\}$. For a DNAQL expression e, we say that ν is an ℓ-complex assignment *on* e if $dom(\nu) \supseteq FV(e)$, and we say that γ is an ℓ-counter assignment *on* e if the free counters of e form a subset of $dom(\gamma)$.

For a given integer $\ell \geq 2$, a DNAQL expression e may evaluate to an ℓ-complex, with respect to an ℓ-complex assignment ν on e and an ℓ-counter assignment γ on e, denoted by $[\![e]\!]^\ell(\nu, \gamma)$. The semantic rules that define this evaluation are shown in Figure 4, except for the obvious semantics rules of the operations defined in Section 2.3. The superscript ℓ has been omitted in the figure to reduce clutter. The rules for `let` and `for` use the oft-used notation $f[x := u]$ to denote the function obtained from f by mapping x to u.

Because the operations on complexes are not always defined, the evaluation may fail, so $[\![e]\!]^\ell(\nu, \gamma)$ may be undefined. When e is a DNAQL program, we denote $[\![e]\!]^\ell(\nu, \emptyset)$ simply by $[\![e]\!]^\ell(\nu)$.

When ℓ is clear from the context, we omit the superscript ℓ and refer to an ℓ-complex assignment and an ℓ-counter assignment simply as a complex assignment and counter assignment, respectively.

2.5 Sticker Complex Types

We now recall from [6, 8] the notion of sticker complex types. Intuitively, a type is a complex where all data entries have been replaced by wildcards. In this way, the dimension and the actual values of the data entries are hidden.

$$\frac{x \text{ is a complex variable}}{[\![x]\!](\nu,\gamma) = \nu(x)} \qquad \frac{[\![e_1]\!](\nu,\gamma) = C_1 \qquad [\![e_2]\!](\nu[x := C_1],\gamma) = C_2}{[\![\texttt{let } x := e_1 \texttt{ in } e_2]\!](\nu,\gamma) = C_2}$$

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

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

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

Fig. 4. The semantic rules for DNAQL .

Let $N = \{\star, \underline{\star}, \hat{\star}\}$ be a set of three symbols disjoint from $\Sigma \cup \overline{\Sigma}$. We let $\Sigma_N = \Omega \cup \Theta \cup N$ represent the positive alphabet of types. Note that Σ_N does not contain atomic value symbols. We let $\overline{\Sigma_N} = \overline{\Omega} \cup \overline{\Theta} \cup \{?\}$, where ? is again a new symbol, represent the negative alphabet of types.

We extend the complementarity relation for types, by defining $\overline{\star} = \overline{\hat{\star}} = ?$ and having $\overline{?}$ undefined. Note that $\underline{\star}$ has no complementary symbol.

A *sticker complex type* (or *type* for short) is very similar to a sticker complex: it is a structure $S = (V, L, \lambda, \mu, \iota, \beta)$ that satisfies the same definition as that of a sticker complex, but with the following exceptions: (1) the range of the node labeling function λ is now $\Sigma_N \cup \overline{\Sigma_N}$ instead of $\Sigma \cup \overline{\Sigma}$, and (2) $\beta \subseteq V$ is not allowed to contain nodes labeled with a symbol from N. Of course, notation and terminology concerning complexes (such as component, isomorphism, subsumes, etc.) carry over to types.

Consider an ℓ-core r of ℓ-complex C. We say that r is of *type* (1) $\star$ if every node of r is free, (2) $\underline{\star}$ if every node of r is blocked, and (3) $\hat{\star}$ if all but one node of r is blocked. Now we call C *well typed* if every ℓ-core occurring in C is of type $\star$, $\underline{\star}$, or $\hat{\star}$. An immobilized node n labeled with the complement of an atomic value symbol is called an *atomic value probe*. We say that an atomic value probe is of *type* ?.

If complex C is well typed, then we define the *stype* (short for "smallest type") of C, denoted by $stype(C)$, as the type obtained from C by:

- contracting every ℓ-core r occurring in C to a node v labeled with the type of r (matchings are respected, i.e., v is matched to a node u if and only if there is a node in r matched to u), and
- replacing the label of a node labeled with an immobilized negative atomic value by ?.

Note that the stype of a complex is defined up to isomorphism.

A well-typed complex C is said to be of *type* S if S subsumes the stype of C. Equivalently, C is of type S if and only if for each component c of C there is a component of S that is an stype of c.

Note that if C is of type S, and S is subsumed by S', then C is also of type S'.

Example 2.4. In Figure 5 a sticker complex type S is depicted. The reader is invited to verify that S is the stype of the complex C of Figure 1.

Remark 2.5. The notion of type as defined here is called "weak type" in a companion article [8]. That article also considers a stronger version of types and develops type checking for DNAQL programs.

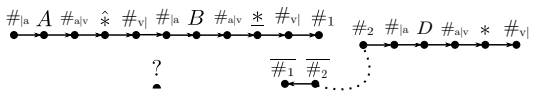

Fig. 5. A sticker complex type S. The complex C in Figure 1 is of type S.

3 Relational Algebra

In this section, we briefly recall the relational algebra, mainly to fix notation and terminology. See, e.g., the monographs [1, 17] for a detailed account.

We assume a set U of *data elements*, a set $\mathcal{A}$ of *attributes*, and a set $\mathcal{N}$ of *relation variables*. A *relation schema* R is a finite subset of $\mathcal{A}$. A *tuple* t over R is a function from R to U. A *relation instance* r over R is a set of tuples over R. A *database schema* $\mathcal{D}$ is a mapping from a finite subset X of $\mathcal{N}$ to relation schemas. We denote by $\mathcal{N}(\mathcal{D})$ the domain X of $\mathcal{D}$. A *database instance* I over $\mathcal{D}$, has domain $\mathcal{N}(\mathcal{D})$ and maps each relation variable $x \in \mathcal{N}(\mathcal{D})$ to a relation instance $I(x)$ over relation schema $\mathcal{D}(x)$.

We recall now the standard operations of the relational algebra: union, difference, cartesian product, projection, renaming, and selection. For convenience, the version of projection presented here is slightly nonstandard as it projects *away* some given attribute instead of *on* some given finite subset of attributes. Obviously, this difference is not essential as the expressive power of this version of the relation algebra is equal to the usual version of the relational algebra. Let r and s be relation instances over relation schemas R and S, respectively.

- If $R = S$, then the *union* and *difference* of r and s are defined and equal to the usual set-theoretical union and difference, respectively: $r \cup s = \{t \mid t \in r \vee t \in s\}$ and $r - s = \{t \mid t \in r \wedge t \notin s\}$.
- If $R \cap S = \emptyset$, then the *cartesian product* of r and s is defined as $r \times s = \{t_r \cup t_s \mid t_r \in r \text{ and } t_s \in s\}$, where $t_r \cup t_s$ denotes the tuple over $R \cup S$ with $(t_r \cup t_s)(A)$ equal to $t_r(A)$ if $A \in R$ and to $t_s(A)$ if $A \in S$.
- The *projection* of r outside $A \in R$ is defined as $\hat{\pi}_A(r) = \{t|_{R\setminus\{A\}} \mid t \in r\}$, where $t|_{R\setminus\{A\}}$ is the restriction of t to $R \setminus \{A\}$.
- The *renaming* of r from $A \in R$ to $B \in \mathcal{A} \setminus R$ is the relation instance over $R' = (R \setminus \{A\}) \cup \{B\}$ defined as $\rho_{A/B}(r) = \{t \circ \phi \mid t \in r\}$, where the function $\phi : R' \to R$ maps B to A and is the identity on $R \setminus \{A\}$.
- The *selection* of r on $A, B \in R$ is defined as $\sigma_{A=B}(r) = \{t \in r \mid t(A) = t(B)\}$.

The syntax of relational algebra expressions is generated by the following grammar:

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

where $x \in \mathcal{N}$, and $A, B \in \mathcal{A}$.

Let us recall the semantics of the relational algebra. Let e be a relational algebra expression, and let $FV(e)$ be the set of relation variables in e. Let I be a database instance over $\mathcal{D}$ with $\mathcal{N}(\mathcal{D}) \supseteq FV(e)$. Then the *evaluation* of e on I, denoted by $[\![e]\!](I)$, is defined inductively in the natural way: $[\![x]\!](I) = I(x)$ and $[\![e_1 \cup e_2]\!](I) = [\![e_1]\!](I) \cup [\![e_2]\!](I)$. The definition is similar for the other relational algebra operations.

The relational algebra operators are only defined under certain conditions on the schemes of their input relation instances. Thus, the evaluation of a relational algebra expression e on a database instance I may not always succeed. To guarantee well-definedness, we employ the following simple typing rules for the relational algebra [31, 32]. Let $\mathcal{D}$ be again a database schema such that $FV(e) \subseteq \mathcal{N}(\mathcal{D})$, and let R be a relation schema. The rules for when e has *type* R given $\mathcal{D}$, denoted

$\mathcal{D} \vdash e : R$, are as follows:

$$\frac{\mathcal{D}(x) = R}{\mathcal{D} \vdash x : R} \qquad \frac{\mathcal{D} \vdash e_1 : R \qquad \mathcal{D} \vdash e_2 : R}{\mathcal{D} \vdash e_1 \cup e_2 : R} \qquad \frac{\mathcal{D} \vdash e_1 : R \qquad \mathcal{D} \vdash e_2 : R}{\mathcal{D} \vdash e_1 - e_2 : R}$$

$$\frac{\mathcal{D} \vdash e_1 : R_1 \qquad \mathcal{D} \vdash e_2 : R_2 \qquad R_1 \cap R_2 = \emptyset}{\mathcal{D} \vdash e_1 \times e_2 : R_1 \cup R_2} \qquad \frac{\mathcal{D} \vdash e : R \qquad A, B \in R}{\mathcal{D} \vdash \sigma_{A=B}(e) : R}$$

$$\frac{\mathcal{D} \vdash e : R \qquad A \in R}{\mathcal{D} \vdash \hat{\pi}_A(e) : R \setminus \{A\}} \qquad \frac{\mathcal{D} \vdash e : R \qquad A \in R \qquad B \notin R}{\mathcal{D} \vdash \rho_{A/B}(e) : (R \setminus \{A\}) \cup \{B\}}.$$

The obtained type system has the property that if I is a database instance over $\mathcal{D}$ and $\mathcal{D} \vdash e : R$, then $[\![e]\!](I)$ is a relation instance over R. If $\mathcal{D} \vdash e : R$ for some R, then we say that e is *well-typed* over $\mathcal{D}$.

4 Relational Algebra Simulation

We represent now relations by complexes. We store data elements as ℓ-vectors of atomic value symbols. So formally, we use Λ^*, the set of string over Λ, as 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. Let $<$ be a linear order of the set $\mathcal{A}$ of attributes. Let $A_1, \ldots, A_k$ be the order of the elements of R under $<$.

We now represent t by an ℓ-complex that consists of a single strand, as follows:

$$\mathit{complex}(t)^< := \#_{|\mathrm{a}} A_1 \#_{\mathrm{a}|\mathrm{v}} t(A_1) \#_{\mathrm{v}|} \cdots \#_{|\mathrm{a}} A_k \#_{\mathrm{a}|\mathrm{v}} t(A_k) \#_{\mathrm{v}|}.$$

This notation carries over to relations r: we denote by $\mathit{complex}(r)^<$ the ℓ-complex $\bigcup\{\mathit{complex}(t)^< \mid t \in r\}$. Here $\bigcup$ denotes the union operation on complexes.

Example 4.1. Let $R = \{A, B, E\}$ be a relation schema with three attributes ordered as $A < B < E$. Let $\ell = 3$ and let $\Lambda = \{0, 1\}$. Consider the tuple t over R with $t(A) = 000$, $t(B) = 010$, and $t(E) = 111$. Then

$$\mathit{complex}(t)^< = \#_{|\mathrm{a}} A \#_{\mathrm{a}|\mathrm{v}} 000 \#_{\mathrm{v}|} \#_{|\mathrm{a}} B \#_{\mathrm{a}|\mathrm{v}} 010 \#_{\mathrm{v}|} \#_{|\mathrm{a}} E \#_{\mathrm{a}|\mathrm{v}} 111 \#_{\mathrm{v}|} \ .$$

Remark 4.2. We remark that the corner case where $R = \emptyset$ is not directly supported in the above defined representation of a relation by a complex. Indeed, the two possible relations over $R = \emptyset$ are both represented by the same complex, namely the empty complex. In order to faithfully cover this corner case, we can do a "preprocessing step" by extending R by a distinguished "dummy" attribute $D \notin R$ that is not used in any relational algebra expression e under consideration. Each tuple t over R is then represented by tuple t' over $R' = R \cup \{D\}$ where $t'|_R = t$ and $t'(D)$ is some fixed constant. For clarity of exposition we implicitly assume this preprocessing step was done, and so we assume that if $\mathcal{D} \vdash e : R$, then R is nonempty.

Let us call the type with string representation $\#_{|\mathrm{a}} A_1 \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} \cdots \#_{|\mathrm{a}} A_k \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|}$ the *relation-schema type* of R, denoted by $S_R^<$. Note that $\mathit{complex}(r)^<$ is of type $S_R^<$ when r is a relation over R. Also note that by Remark 4.2 we assume without loss of generality that $S_R^<$ is not empty. A substrand of the form $\#_{|\mathrm{a}} A \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|}$ consists of an attribute and a wildcard for a value, whence it is called an *attribute-value block*.

In a final step, we can represent database instances over some database schema $\mathcal{D}$ as complex assignments. Any database instance I over $\mathcal{D}$ can be represented by the complex assignment $\mathit{complex}(I)^<$ that assigns $\mathit{complex}(I(x))^<$ to each $x \in \mathcal{N}(\mathcal{D})$. Note that $\mathit{complex}(I(x))^<$ is of type $S_{\mathcal{D}(x)}^<$ for each $x \in \mathcal{N}(\mathcal{D})$.

We are now ready to state the main result of this article.

Theorem 4.3. *Let e be a well-typed relational algebra expression over a database schema $\mathcal{D}$. Then there exists a DNAQL program e^{DNA} that simulates e uniformly over all dimensions ℓ. In other words, for each integer $\ell \geq 2$ and for any ℓ-dimensional database instance I over $\mathcal{D}$ we have*

$$[\![e^{DNA}]\!](complex(I)^{<}) \equiv complex([\![e]\!](I))^{<}.$$

Moreover, such a DNAQL program e^{DNA} is effectively computable from e and $\mathcal{D}$. Finally, the length of e^{DNA} (i.e., the number of operations in e^{DNA}) can be chosen to be $O(|e| + kn \log n)$, where $|e|$ is the length of e, k is the number of cartesian product operations in e, and n is the maximum cardinality among the schemas of subexpressions e' of e in which the final operation is the cartesian product (i.e., $e' = e_1 \times e_2$ for some expressions e_1 and e_2). In particular, if e does not use cartesian product, then the length of e^{DNA} can be chosen to be linear in the length of e.

In Sections 5 and 6, we provide a proof of this result. If the order $<$ is clear from the context, then $<$ is often omitted from the notation.

5 Helper Functions

In order to prove Theorem 4.3 in Section 6, we define in this section some helper functions (that is, DNAQL expression fragments) and prove properties about them.

5.1 Some Auxiliary Definitions

Schemas and Types. A *pseudo-relation-schema type* of R, resembles the relation-schema type of R, except that a limited number of additional tags outside $\{\#_{|a}, \#_{a|v}, \#_{v|}\}$ may be present between attribute-value blocks. More precisely, a pseudo-relation-schema-type S is of the form

$$\#_{|a} A_1 \#_{a|v} * \#_{v|} I_1 \#_{|a} A_2 \#_{a|v} * \#_{v|} I_2 \cdots I_{k-1} \#_{|a} A_k \#_{a|v} * \#_{v|},$$

where $A_1, \ldots, A_k$ are attributes and $I_1, \ldots, I_{k-1}$ are (possibly empty) sequences of tags of length at most 3, not containing the tags $\#_{|a}$, $\#_{a|v}$, and $\#_{v|}$, called *intermediate tag sequences* of S. Hence, a relation-schema type is a special case of a pseudo-relation schema type, where the intermediate tag sequences are all empty.

Remark 5.1. We remark that the upper bound of 3 on the length of each intermediate tag sequence is sufficient for our purposes, but can be relaxed up to $2(\ell + 4) - 1 = 2\ell + 7$ without changing any results or proofs. We will see that we cannot go beyond this value (at least not in the current setup) because the proof of Lemma 5.4 uses the assumption that the length of each intermediate tag sequence is less than the length of two attribute-value blocks, which is $2(\ell + 4)$.

We introduce now a few useful abbreviations.

Blockfromto. For $a, b \in \Sigma$ and e a DNAQL expression, we use $blockFromTo(e, a, b)$ to abbreviate

$$\texttt{blockfrom}(\texttt{block}(e, b), a).$$

We remark that this operation is called $blockFromTo(e, a, b)$ instead of $blockFromTo(e, b, a)$ to be in line with the definition of `blockfrom` (which in turn is in line with how blocking is accomplished in DNA).

Connect. A frequently reoccurring pattern in DNAQL programs is adding several complexes together, by means of unions, and then applying hybridize, ligate and cleanup consecutively. We abbreviate the last part of this pattern

$$\texttt{cleanup}(\texttt{ligate}(\texttt{hybridize}(e)))$$

by $connect(e)$.

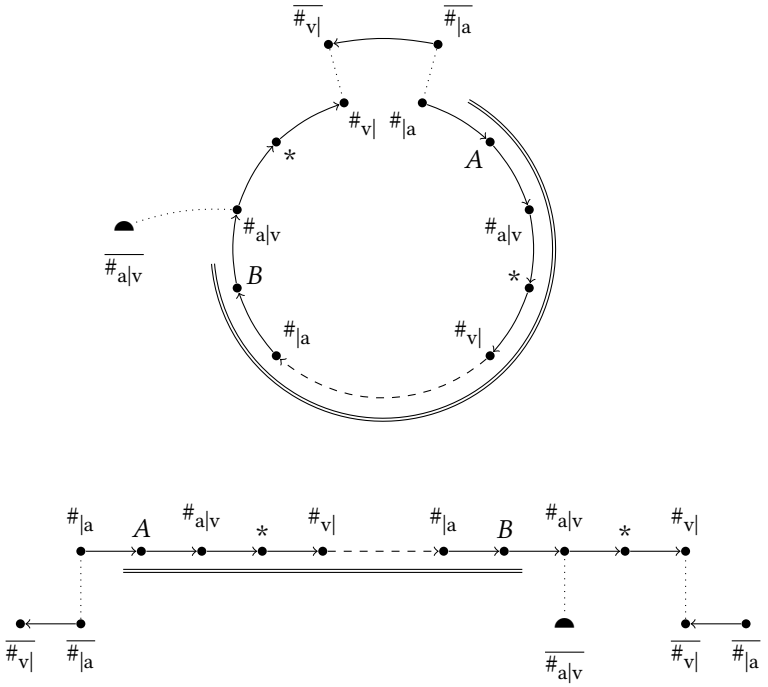

Fig. 6. The state after $\texttt{hybridize}(f_2 \cup \overline{\#_{|a}\#_{v|}})$ during circularization.

5.2 Circularization

Let e be a DNAQL expression and let A and B be attributes. We use $\mathit{circularize}(e, A, B)$ to abbreviate

$$\texttt{let } f_2 := \texttt{hybridize}(\mathit{blockFromTo}(e, B, A) \cup \texttt{immob}(\overline{\#_{a|v}})) \texttt{ in}$$
$$\texttt{let } f_1 := \mathit{connect}(f_2 \cup \overline{\#_{|a}\#_{v|}}) \texttt{ in}$$
$$\texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(f_1, A), \#_{a|v})).$$

A *circularization* [3, 26], or *circular version*, s' of a linear strand s is a complex isomorphic to the complex obtained from s by adding a directed edge from the last node of s to the first node of s.

Since DNAQL operations do not modify the ℓ-cores, we (may) for notational convenience often focus on types. In particular, in figures we will stick from now on to types.

Lemma 5.2. *Let e be a DNAQL expression, ν be a complex assignment, and γ be a counter assignment, and $C = [\![e]\!](\nu, \gamma)$ be of a pseudo-relation-schema type S. Let A and B be the first and last attributes of S, respectively. Then $[\![\mathit{circularize}(e, A, B)]\!](\nu, \gamma)$ consists of the circularizations of all strands of C.*

Proof. It is straightforward to verify that the type of the complex $[\![\texttt{hybridize}(f_2 \cup \overline{\#_{|a}\#_{v|}})]\!](\nu, \gamma)$ is as shown in Figure 6. Consequently, we observe that the type of the complex $[\![\texttt{blockfrom}(f_1, A)]\!](\nu, \gamma)$ is as shown in Figure 7. Indeed, the top (bottom, resp.) component of Figure 7 corresponds to the top (bottom, resp.) component of Figure 6. To keep only (the unblocked version of) the top component, we cut the bottom component into smaller pieces by applying split on $\#_{a|v}$. The positions where the bottom component is cut are indicated by scissor symbols. Note

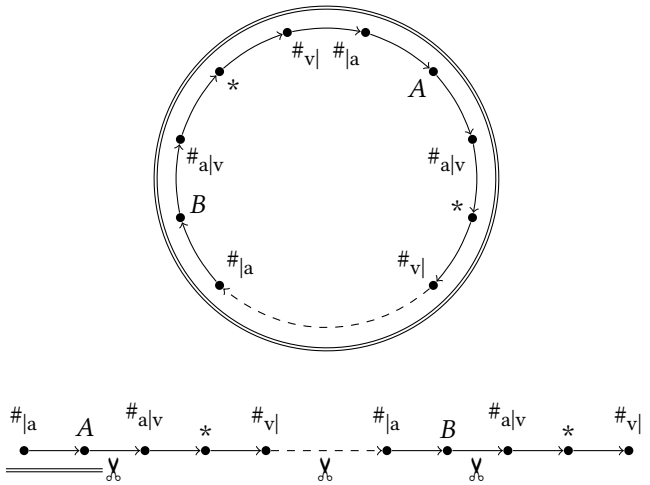

Fig. 7. The state after blockfrom(f_1, A) during circularization and the positions where split on $\#_{a|v}$ will remove edges are indicated.

that the top component is unaffected by split. Since cleanup retains the largest component and removes blocking, we obtain the desired result. Note that, as required, cleanup obtains the empty complex when C is empty, since *circularize* does not introduce positive strands. □

5.3 Inserting Into a Circle

Let e be a DNAQL expression and let A and B be attributes. We use $insertCirc(e, A, B, s)$ to abbreviate

$$\begin{aligned}&\texttt{let } y_1 := \texttt{split}(blockFromTo(e, A, B), \#_{v|}) \texttt{ in}\\ &\texttt{let } y_2 := \texttt{hybridize}(\texttt{hybridize}(y_1 \cup \texttt{immob}(\overline{\#_{a|v}})) \cup \overline{\#_{v|}\sigma_1} \cup s) \cup \overline{\sigma_2\#_{|a}} \texttt{ in}\\ &\texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(connect(y_2), B), \#_{a|v})).\end{aligned}$$

Lemma 5.3. *Let S be a pseudo-relation-schema type with an intermediate tag sequence I_i of length at least two such that the first and the last symbol of I_i are unique in S, and let S' be obtained from S by omitting I_i. Let S_c and S'_c be the circularizations of S and S', respectively.*

Let e be a DNAQL expression, ν be a complex assignment, and γ be a counter assignment, and $C = [\![e]\!](\nu, \gamma)$ be nonempty and of type S'_c. Let A_i and A_{i+1} be consecutive attributes of S'. Then $[\![insertCirc(e, A_i, A_{i+1}, I_i)]\!](\nu, \gamma)$ is of type S_c and obtained from C by inserting, in each component, the intermediate tag sequence I_i between the consecutive attributes A_i and A_{i+1}.

Proof. It is straightforward to verify that the complex $[\![blockFromTo(e, A_i, A_{i+1})]\!](\nu, \gamma)$ is of type as shown in Figure 8. By the scissor symbol the location is depicted where split on $\#_{v|}$ will remove an edge. We next observe that the type of the complex $[\![\texttt{hybridize}(y_2)]\!](\nu, \gamma)$ is as shown in Figure 9. Note that the only difference between the top and bottom component is whether one or two copies of $\overline{\sigma_2\#_{|a}}$ attach to the immobilized component. Notice that the subsequent ligate and cleanup operations will retain both a circular and linear strand since (1) they are of equal size and (2) strands of these forms appear in the input before applying cleanup because C is nonempty. Similar as for the circularization operation, we apply a blockfrom followed by a split, followed by a cleanup to get rid of the linear component, and consequently obtain the desired result. □

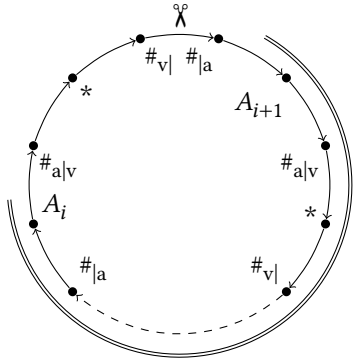

Fig. 8. The state after *blockFromTo*(e, A_i, A_{i+1}) during insertion into a circle and the position where `split` on $\#_{\text{v}|}$ will remove an edge.

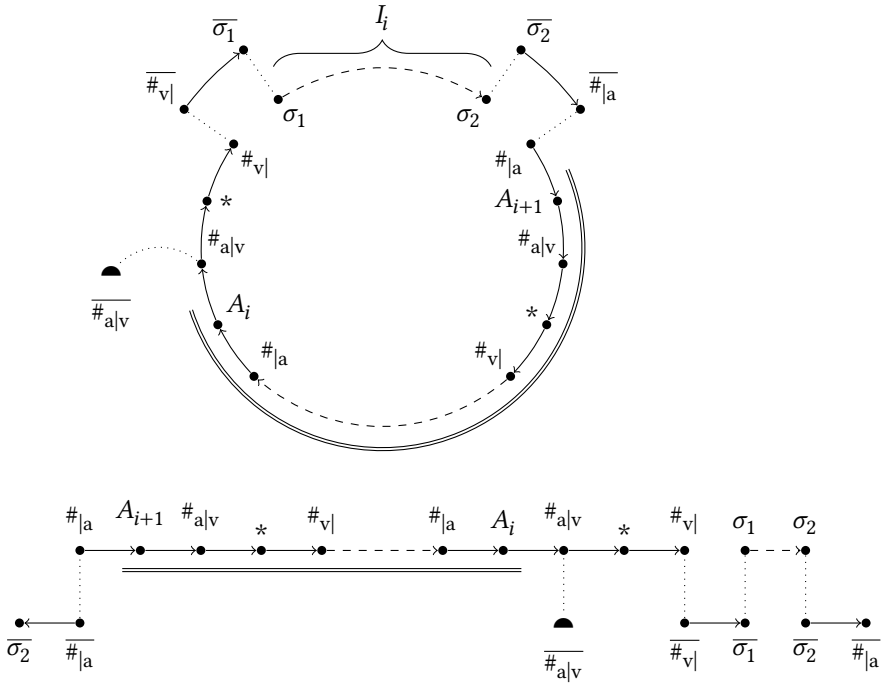

Fig. 9. The state after `hybridize`(y_2) during insertion into a circle.

5.4 Removing Substrands

Let S be the circularization of a pseudo-relation-schema type. Given a complex of type S, we would like to be able to remove substrands, for example to remove an intermediate tag sequence between two particular consecutive attribute-value blocks (and thereby also linearizing the strands). We now define the operation *removeBetweenCirc* to do exactly that.

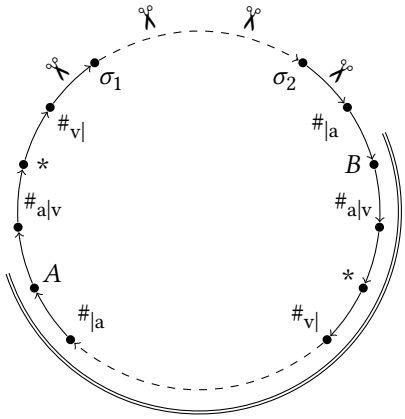

Fig. 10. The state after *blockFromTo*(e, B, A) during the computation of *removeBetweenCirc*(e, A, B) that removes the string $\sigma_1 \cdots \sigma_2$. The positions where `split` on $\#_{|a}$ and $\#_{v|}$ will removes edges are indicated.

Let e be a DNAQL expression and A and B attributes. We abbreviate

$$\texttt{cleanup}(\texttt{split}(\texttt{split}(\mathit{blockFromTo}(e, A, B), \#_{v|}), \#_{|a}))$$

by *removeBetweenCirc*(e, A, B).

We now show that *removeBetweenCirc* can remove substrands between attribute-value blocks.

Lemma 5.4. *Let e be a DNAQL expression, ν be a complex assignment, and γ be a counter assignment, and S be the circularization of a pseudo-relation-schema type having distinct attributes A and B. If $C = [\![e]\!](\nu, \gamma)$ is of type S, then $[\![\mathit{removeBetweenCirc}(e, A, B)]\!](\nu, \gamma)$ is of pseudo-relation-schema type and is obtained from C by removing the substrands from directly after the attribute-value block of A until (and excluding) the attribute-value block of B. In other words, the substrands from the attribute-value block of B until (and including) the attribute-value block of A are retained.*

Proof. Figure 10 depicts the type of $[\![\mathit{blockFromTo}(e, B, A)]\!](\nu, \gamma)$ and the positions where `split` removes edges when splitting on $\#_{|a}$ and $\#_{v|}$. Since $A \neq B$ and by the length restriction on intermediate tag sequences, the blocked component is largest, and so only components of this form are retained (in unblocked fashion) by cleanup. In the corner case where nothing is removed (and so σ_1 and σ_2 in Figure 10 do not exist), merely the edge is removed between the free nodes labelled $\#_{v|}$ and $\#_{|a}$. □

By Lemma 5.4, *removeBetweenCirc* is able to remove intermediate tag sequences. Indeed, if I_i is an intermediate tag sequence right after the attribute-value block of A and right before the attribute-value block of B, then *removeBetweenCirc*(e, A, B) removes I_i.

Let e be a DNAQL expression and let A, Z, D, and E be attributes. We abbreviate

$$\mathit{removeBetweenCirc}(\mathit{circularize}(e, A, Z), D, E)$$

by *removeBetweenShift*$(e, A, Z; (D, E))$.

We observe that if A and Z are the first and last attribute-value blocks of a pseudo-relation-schema type S, respectively, and the attribute-value block of D precedes that of E in S, then each strand of *removeBetweenShift*$(e, A, Z; (D, E))$ is obtained from C (of type S) by concatening two substrands of a strand s of C: the substrand of s from the attribute-value block of E until the end of

s with the substrand of s from the start of s until (and including) the attribute-value block of D. This explains the name *removeBetweenShift*: it removes a substrand and concatenates the remaining two substrands in a way that it is a cyclic shift compared to the original ordering of the attributes.

Let e be a DNAQL expression, let A, A', Z, and Z' be attributes, and let $\sigma = ((D_1, E_1), \cdots, (D_n, E_n))$ be a sequence of pairs of attributes. We abbreviate

$$\begin{aligned}
&\texttt{let } f_1 := \mathit{removeBetweenShift}(e, A, Z; (D_1, E_1))) \texttt{ in}\\
&\texttt{let } f_2 := \mathit{removeBetweenShift}(f_1, E_1, D_1; (D_2, E_2)) \texttt{ in}\\
&\cdots\\
&\texttt{let } f_n := \mathit{removeBetweenShift}(f_{n-1}, E_{n-1}, D_{n-1}; (D_n, E_n)) \texttt{ in}\\
&\mathit{removeBetweenShift}(f_n, E_n, D_n; (Z', A')),
\end{aligned}$$

by $\mathit{removeBetweenIntervals}(e, A, Z, A', Z', \sigma)$.

To state the next result, we first define the notion of intervals. Let us define for distinct attributes A, B, and D, the predicate $\text{between}_<(A, D, B)$ that is true if and only if $A < D < B$ or ($B < A$ and not $B < D < A$). So, roughly speaking, D is between A and B after turning the linear ordering $<$ into a directed circle. We now define the *interval* of a pair of distinct attributes (A, B) as the set of attributes $\{D \mid \text{between}_<(A, D, B)\}$. If the interval of (A, B) is empty, then we say that A is the *cyclic predecessor* of B and that B is the *cyclic successor* of A. The *closed interval* of (A, B) is $I \cup \{A\}$, where I is the interval of (A, B).

The following corollary says that *removeBetweenIntervals* removes all substrands between the attribute-value blocks of each pair in σ.

Corollary 5.5. *Let e be a DNAQL expression, ν be a complex assignment, and γ be a counter assignment, and S be of pseudo-relation-schema type. Let A and Z be the first and last attribute of S, respectively. Let α be a set of pairs of attributes appearing in S such that their closed intervals are mutually disjoint, and let σ be any linear ordering of α. Let A' and Z' be the first and last attribute of S, respectively, that is not in any interval of a pair in α.*

If $C = [\![e]\!](\nu, \gamma)$ is of type S, then $\mathit{removeBetweenIntervals}(e, A, Z, A', Z', \sigma)$ is of pseudo-relation-schema type and is obtained from C by removing all substrands between (in the cyclic sense) the attribute-value blocks of D and E for all $(D, E) \in \alpha$ and gluing the remaining strands back together in the same order as for S.

For example, let S be of the form

$$\#_{|\mathrm{a}} A \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_1 \#_{|\mathrm{a}} D \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_2 \#_{|\mathrm{a}} E \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_3 \#_{|\mathrm{a}} F \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_4 \#_{|\mathrm{a}} G \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_5 \#_{|\mathrm{a}} Z \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|}.$$

If $\sigma = ((A, E), (E, Z))$, then $\mathit{removeBetweenIntervals}(e, A, Z, A, Z, \sigma)$ is of type

$$\#_{|\mathrm{a}} A \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_1 \#_{|\mathrm{a}} E \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_3 \#_{|\mathrm{a}} Z \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|},$$

and if $\sigma = ((G, D), (E, F))$, then $\mathit{removeBetweenIntervals}(e, A, Z, D, G, \sigma)$ is of type

$$\#_{|\mathrm{a}} D \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_2 \#_{|\mathrm{a}} E \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_3 \#_{|\mathrm{a}} F \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_4 \#_{|\mathrm{a}} G \#_{\mathrm{a}|\mathrm{v}} * \#_{\mathrm{v}|} I_5.$$

Proof. We first verify that the attribute-value blocks of D and E always exist for all $(D, E) \in \alpha$ during *removeBetweenIntervals* (for any linear ordering σ of α). Since the closed intervals are mutually disjoint, it suffices to show that E is not removed by any *removeBetweenShift* in *removeBetweenIntervals*. Assume to the contrary that the attribute-value blocks of E is removed. Then E is in the interval of some $(D', E') \in \alpha$. Let F be the cyclic predecessor of E. Then F is in the closed interval of (D', E'). Also, since $D \neq E$, F is either equal to D or in the interval of (D, E). So, F is also in the closed interval of (D, E), which is a contradiction of the mutual disjointness of the closed intervals.

The result now follows from Lemma 5.2 and Lemma 5.4. Notice that the last line in the definition of *removeBetweenIntervals* is only there to retain the original ordering of the attributes (as opposed to some cyclic shift of this ordering) and does not remove anything. □

We remark that since $\mathit{removeBetweenShift}(f_n, E_n, D_n; (Z', A'))$ in the definition of $\mathit{removeBetweenIntervals}(e, A, B, A', B', \sigma)$ does not remove anything, in this situation splitting on $\#_{|\mathrm{a}}$ is a no-op and can be removed. Since the goal is about expressivity of DNAQL, we do not introduce additional terminology to avoid the no-op.

5.5 Concatenation

Let e_1 and e_2 be DNAQL expressions and let A_1, Z_1, A_2, and Z_2 be attributes. We use $\mathit{concatenate}(e_1, e_2, (A_1, Z_1), (A_2, Z_2))$ to abbreviate

$$
\begin{aligned}
&\texttt{if empty}(e_1)\ \texttt{then empty else}\\
&\ \texttt{if empty}(e_2)\ \texttt{then empty else}\\
&\ \texttt{let}\ l := \mathit{connect}(e_2 \cup \overline{\#_{\mathrm{v}}|\#_2} \cup \#_2)\ \texttt{in}\\
&\ \texttt{let}\ r := \mathit{connect}(e_1 \cup \overline{\#_1\#_{|\mathrm{a}}} \cup \#_1)\ \texttt{in}\\
&\ \texttt{let}\ f := \mathit{connect}(l \cup r \cup \overline{\#_2\#_1})\ \texttt{in}\\
&\ \mathit{removeBetweenShift}(f, A_2, Z_1; (Z_2, A_1)).
\end{aligned}
$$

For strands s_1 and s_2 with disjoint sets of nodes, we denote by $s_1 \cdot s_2$ the strand obtained from s_1 and s_2 by adding an edge from the last node of s_1 to the first node of s_2. We extend the $\cdot$ operator in a natural way to complexes C_1 and C_2 where each component is a strand. Then $C_1 \cdot C_2$ denotes the complex (defined up to equivalence) consisting of $s_1 \cdot s_2$ for (disjoint) copies s_1 and s_2 of strands of C_1 and C_2, respectively.

The next result shows that *concatenate* expresses the $\cdot$ operator if the inputs are of relation-schema type.

Lemma 5.6. *Let e_1 and e_2 be DNAQL expressions, ν be a complex assignment, and γ be a counter assignment, and S_1 and S_2 both be of relation-schema type with mutually distinct attributes. Let A_1 and Z_1 be the first and last attribute of S_1, respectively, and A_2 and Z_2 be the first and last attribute of S_2, respectively.*

If $C_1 = [\![e_1]\!](\nu, \gamma)$ is of type S_1 and $C_2 = [\![e_2]\!](\nu, \gamma)$ is of type S_2, then complex $C = [\![\mathit{concatenate}(e_1, e_2, (A_1, Z_1), (A_2, Z_2))]\!](\nu, \gamma) \equiv C_1 \cdot C_2$. Consequently, C is of the relation-schema type that has as string respresentation the concatenation of the string representations of S_1 and S_2 (in this order).

Proof. Note that if C_1 or C_2 are empty, then C is the empty complex, as required. We now assume both C_1 and C_2 are nonempty.

Since C_1 and C_2 are nonempty, $C_l = [\![l]\!](\nu, \gamma)$ is obtained from C_2 by appending a node labelled $\#_2$ to each strand, and $C_r = [\![r]\!](\nu, \gamma)$ is obtained from C_1 by prepending a node labelled $\#_1$ to each strand. Consequently, $C_f = [\![f]\!](\nu, \gamma)$ concatenates the strands of C_l with the strands of C_r. Note that the first attribute of C_f is A_2, the last attribute of C_f is Z_1, and there is a "junk" sequence $\#_2\#_1$ between the attribute-value blocks of Z_2 and A_1. From Section 5.4 we observe that $\mathit{removeBetweenShift}(f, A_2, Z_1; (Z_2, A_1))$ removes the junk and does a cyclic shift of the attribute-value blocks such that each strand of C consists of two substrands, the first being a strand of C_1 and the other a strand of C_2. □

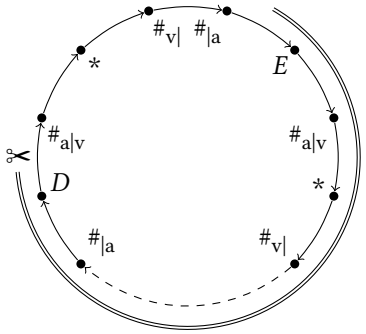

Fig. 11. The state after expression *blockFromTo*(f_1, D, E) during attribute replacement *replaceAttribute* and the position where `split` on $\#_{a|v}$ will remove an edge.

5.6 Replacing an Attribute

Let e be a DNAQL expression and let A, B, D, D', E, and Z be attributes. We abbreviate

$$\begin{aligned}
&\texttt{if empty}(e)\ \texttt{then empty else}\\
&\texttt{let } f_1 := \mathit{circularize}(e, A, Z)\ \texttt{in}\\
&\texttt{let } f_2 := \texttt{split}(\mathit{blockFromTo}(f_1, D, E), \#_{a|v})\ \texttt{in}\\
&\texttt{let } f_3 := \mathit{connect}(f_2 \cup \#_{|a} D' \cup \overline{D'\#_{a|v}})\ \texttt{in}\\
&\texttt{let } f_4 := \texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(f_3, B), \#_{|a}))\ \texttt{in}\\
&\mathit{removeBetweenShift}(f_4, D', B; (Z, A)),
\end{aligned}$$

by *replaceAttribute*$(e, A, Z; (B, D, E), D')$.

Lemma 5.7. *Let e be a DNAQL expression, v be a complex assignment, and γ be a counter assignment, and S be of relation-schema type with at least two attributes. Let A and Z be the first and last attribute of S, respectively. Let D be an attribute of S, and D' be an attribute not in S. Let B be the cyclic predecessor of D and E be the cyclic successor of D with respect to the linear ordering of the attributes of S.*

If $C = [\![e]\!](v, \gamma)$ is of type S, then $[\![\mathit{replaceAttribute}(e, A, Z; (B, D, E), D')]\!](v, \gamma)$ is obtained from C by replacing every occurrence of attribute D by attribute D' in every component of C.

Proof. For similar reasons as in the proof of Lemma 5.6, we need to consider the case where C is the empty complex separately. If C is empty, then *replaceAttribute* obtains the empty complex, as required. We now assume C is nonempty.

The computation of expression f_2 after circularization is almost identical to the computation of expression y_1 in the definition of *insertCirc*. Indeed, the only difference is the tag on which `split` takes place. Figure 11 depicts the type of the complex. We next observe that the type of the complex of $\texttt{hybridize}(f_2 \cup \#_{|a} D' \cup \overline{D'\#_{a|v}})$ is as given in Figure 12. Ligate adds an edge from the node labelled D' to the unblocked node labelled $\#_{a|v}$, and `cleanup` removes the negative strands of the form $\overline{D'\#_{a|v}}$. Therefore, the type of $\texttt{blockfrom}(f_3, B)$ is as given in Figure 13. In the figure the position where `split` removes an edge is indicated. After splitting and `cleanup` where the strands of the form $\#_{|a} D$ are removed, we are done modulo a cyclic shift of the attribute-value blocks. This cyclic shift is accomplished by *removeBetweenShift*$(f_4, D', B; (Z, A))$. □

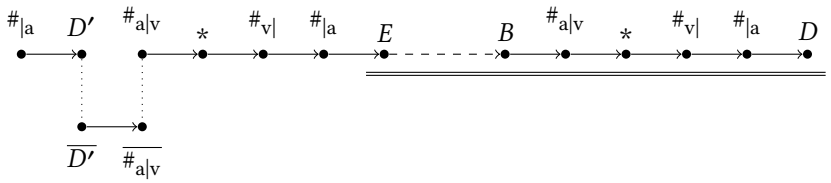

Fig. 12. The state after hybridize($f_2 \cup \#_{|a} D' \cup \overline{D'\#_{a|v}}$) during *replaceAttribute*.

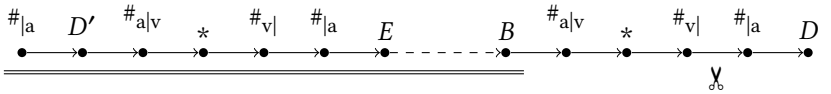

Fig. 13. The state after blockfrom(f_3, B) during *replaceAttribute*, with the position where splitting on $\#_{|a}$ takes place.

Note that Lemma 5.7 does not cover the case where S has one attribute. To this end, let us, for a DNAQL expression e and attribute D', abbreviate

$$\begin{aligned}&\texttt{if empty}(e)\ \texttt{then empty else}\\ &\mathit{connect}(\texttt{split}(e, \#_{a|v}) \cup \#_{|a} D' \cup \overline{D'\#_{a|v}}),\end{aligned}$$

by *replaceAttributeUnary*(e, D'). We observe that if S is a relation-schema type with exactly one attribute and $C = [\![e]\!](\nu, \gamma)$, then $[\![\mathit{replaceAttributeUnary}(e, D')]\!](\nu, \gamma)$ is obtained from C by replacing every occurrence of attribute D by attribute D' in every component of C. Indeed, $\texttt{split}(e, \#_{a|v})$ obtains two types of strands: $\#_{|a} D$ and $\#_{a|v} * \#_{v|}$. Similar as in the proof of Lemma 5.7, the rest of the *replaceAttributeUnary* operation prepends $\#_{|a} D'$ to the strands of type $\#_{a|v} * \#_{v|}$ and cleanup only retains strand of type $\#_{|a} D' \#_{a|v} * \#_{v|}$, as required.

5.7 Block Selecting

Let e be a DNAQL expression and let A and B be attributes. Let a be an atomic value symbol and let i be a counter variable. We use *blockSelect*(e, A, B, a, i) to abbreviate

$$\begin{aligned}&\texttt{let } y := \texttt{blockexcept}(\mathit{blockFromTo}(e, A, B), i) \cup \texttt{immob}(\overline{a})\ \texttt{in}\\ &\texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(y))).\end{aligned}$$

The next result shows that *blockSelect* functions as a filter for specific atomic value symbols in ℓ-cores, when the input complex is a circularization of a complex of pseudo-relation-schema type.

Lemma 5.8. *Let e be a DNAQL expression, ν be a complex assignment, and γ be a counter assignment, and S be a circularization of a pseudo-relation-schema type. Let A and B be two consecutive attributes of S (i.e., the attribute-value block of B directly follows the attribute-value block of A), let a be an atomic value symbol, and let i be a counter variable.*

If $C = [\![e]\!](\nu, \gamma)$ is of type S, then $C' = [\![\mathit{blockSelect}(e, A, B, a, i)]\!](\nu, \gamma)$ is also of type S and consists of those components of C such that the label of the $\gamma(i)^{th}$ node of the ℓ-core of the attribute-value block of attribute A is equal to a. In particular, C subsumes C'.

Proof. Figure 14 depicts the form of each component of the complex obtained just before cleanup is executed. Notice that the unblocked and unmatched nodes in the top-right of the circle are tags (they form an intermediate tag sequence together with a $\#_{|a}$ symbol), and therefore

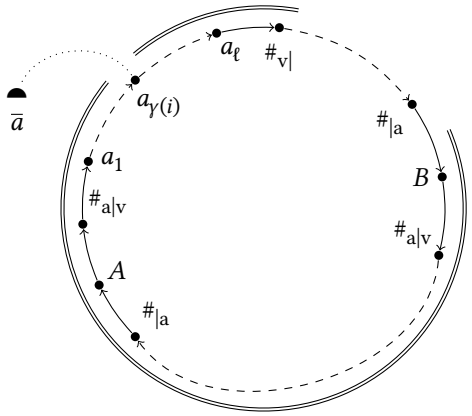

Fig. 14. The form of each component of the obtained complex just before `cleanup` is executed during the computation of *blockSelect*(x, A, B, a, i).

could not have been matched by $\overline{a}$ when `hybridize` was executed. Consequently, exactly those components of C are selected where the label of the $\gamma(i)^{th}$ node of the ℓ-core of the attribute-value block of attribute A is equal to a. In the final step `cleanup` retains only the circular strand without blocking, as desired. □

We can use *blockSelect* to filter out components for which two specific ℓ-cores are equal. First we define notation for expressions containing variables a for atomic value symbols. More precisely, consider expressions e_a that are DNAQL expressions except for the fact that it contains an indeterminate a in the position where a constant from Λ can reside. Then we let $\bigcup_{a\in\Lambda} e_a$ denote $e_{a_1} \cup e_{a_2} \cup \cdots \cup e_{a_n}$, where $\Lambda = \{a_1, \ldots, a_n\}$. Since Λ is fixed and union is commutative and associative, $\bigcup_{a\in\Lambda} e_a$ is a well-defined DNAQL expression.

Let e be a DNAQL expression and let A, B, D, E, F, and Z be attributes. Let i be a counter variable. We abbreviate

$$
\begin{array}{l}
\texttt{let } f := \\
\quad \texttt{for } x_c := \mathit{circularize}(e, A, Z) \texttt{ iter } i \texttt{ do} \\
\qquad \bigcup_{a\in\Lambda} \mathit{blockSelect}(\mathit{blockSelect}(x_c, B, D, a, i), E, F, a, i) \\
\texttt{in} \\
\quad \mathit{removeBetweenCirc}(f, Z, A)
\end{array}
$$

by *coreEquality*$(e, A, Z; (B, D), (E, F), i)$.

Lemma 5.9. *Let e be a DNAQL expression, ν be a complex assignment, and γ be a counter assignment, and S be of relation-schema type having at least two attributes. Let, with respect to the linear order of the attributes of S, D be the cyclic successor of B and F be the cyclic successor of E. Let A and Z be the first and last attribute of S, respectively. Let i be a counter variable.*

If $C = [\![e]\!](\nu, \gamma)$ is of type S, then $C' = [\![\mathit{coreEquality}(e, A, Z; (B, D), (E, F), i)]\!](\nu, \gamma)$ is also of type S and consists of those components of C such that the ℓ-cores of the attribute-value blocks of attributes B and E are equal. In particular, C subsumes C'.

Proof. Note that by Lemma 5.8, for $a \in \Lambda$, $e_a = \mathit{blockSelect}(\mathit{blockSelect}(x_c, B, D, a, i), E, F, a, i)$ obtains those components where the $\gamma(i)^{th}$ nodes of the ℓ-cores of B and E are both equal to a. Therefore, the union of the e_a's with $a \in \Lambda$ obtains exactly the components where the $\gamma(i)^{th}$ nodes of the ℓ-cores of B and E are equal. The expression f iterates over all $\gamma(i)$ over all nodes of the ℓ-cores and so f obtains those components where the ℓ-cores of B and E are equal. The final step $\mathit{removeBetweenCirc}(f, Z, A)$ is only there to make the strand linear again (nothing is removed). □

5.8 Shuffling Attribute-value Blocks

For the simulation of some relational algebra operations, we will need to reorder attribute-value blocks of complexes of relation-schema type. Any reordering can be performed by repeated moving of a specific attribute-value block to the end. Indeed, one first moves the first attribute-value block in the intended ordering to the end, then the second in the intended ordering, and so on. We now define *moveToEnd* which does this movement to the end.

Let e be a DNAQL expression. We use $\mathit{moveToEnd}(e, A, Z; (B, D, E))$ to denote

$$
\begin{aligned}
&\texttt{if empty}(e)\ \texttt{then empty else}\\
&\texttt{let } f_0 := \mathit{circularize}(e, A, Z)\ \texttt{in}\\
&\texttt{let } f_1 := \mathit{insertCirc}(f_0, B, D, \#_3\#_4)\ \texttt{in}\\
&\texttt{let } f_2 := \mathit{insertCirc}(f_1, D, E, \#_5\#_6)\ \texttt{in}\\
&\texttt{let } f_3 := \texttt{split}(\mathit{blockFromTo}(f_2, Z, A), \#_{\mathrm{v}|})\ \texttt{in}\\
&\texttt{let } f_4 := \mathit{connect}(f_3 \cup \#_6\#_1 \cup \overline{\#_{\mathrm{v}|}\#_6})\ \texttt{in}\\
&\texttt{let } f_5 := \texttt{hybridize}(f_4 \cup \texttt{immob}(\overline{A}))\ \texttt{in}\\
&\texttt{let } f_6 := \texttt{hybridize}(f_5 \cup \overline{\#_3\#_5} \cup \overline{\#_1\#_4})\ \texttt{in}\\
&\texttt{let } f_7 := \texttt{cleanup}(\texttt{ligate}(\texttt{split}(\texttt{split}(f_6, \#_3), \#_5)))\ \texttt{in}\\
&\mathit{removeBetweenIntervals}(f_7, A, D, A, D, ((B, E), (Z, D))).
\end{aligned}
$$

Lemma 5.10. *Let e be a DNAQL expression, v be a complex assignment, and γ be a counter assignment, and S be a relation-schema type. Let A be the first attribute of S, let Z be the last attribute of S, and let B be the cyclic predecessor of D and E be the cyclic successor of D with respect to the linear order of the attributes of S. We assume $D \neq Z$.*

If $C = [\![e]\!](v, \gamma)$ is of type S, then $C' = [\![\mathit{moveToEnd}(e, A, Z; (B, D, E))]\!](v, \gamma)$ consists of the strands obtained from C by moving the attribute-value block of D to the end. In particular, C' is of relation-schema type.

The proof of this result serves two purposes: in addition to proving the result, we also illustrate how certain non-trivial steps within the *moveToEnd* operation would occur on the level of DNA. In particular, the *moveToEnd* operation implements a novel technique we call *double bridging*. Instead of using a single sticky bridge, two sticky bridges are hybridized onto one strand. A careful placement of the bridges allows cutting the strand twice whilst keeping it connected. Moreover, the two bridges guide the strand into its new conformation.

Proof. If C is the empty complex, then C' is the empty complex, as required. We now assume C is nonempty.

We observe that f_3 in the definition of *moveToEnd* inserts $\#_3\#_4$ between attributes B and D and inserts $\#_5\#_6$ between attributes D and E. Next, f_4 inserts $\#_6\#_1$ at the end of each strand (since C is nonempty, no "junk" component $\#_6\#_1$ remains). Then f_5 immobilizes the strand by hybridizing on attribute A (the choice of A is not essential). The type of the resulting complex $[\![f_5]\!](v, \gamma)$ is

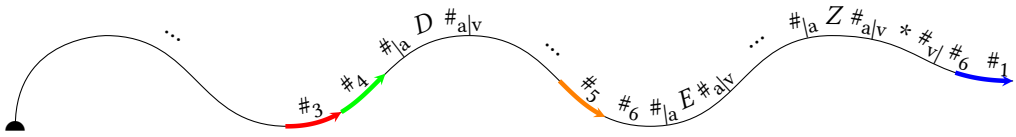

Fig. 15. Linear structure.

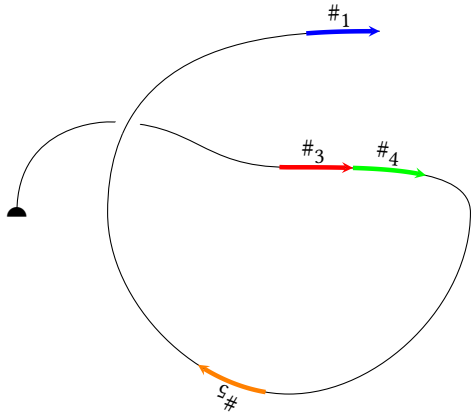

Fig. 16. Some configuration of the linear structure of Figure 15.

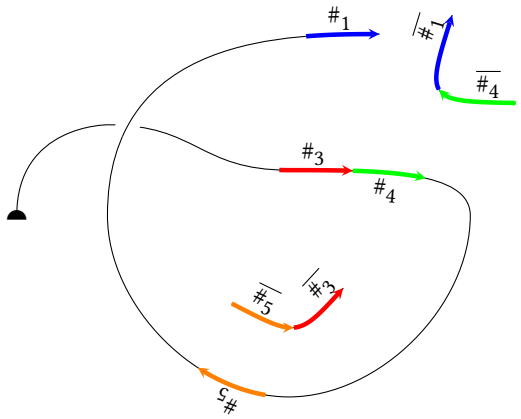

Fig. 17. Stickers added to Figure 16.

depicted in Figure 15. Since the locations of the markers $\#_1$, $\#_3$, $\#_4$, and $\#_5$ are most important, the figure indicates them with colors. While this figure, and the other figures referred to in this proof, depict the case where $A \neq D$ and $E \neq Z$, it is straightforward to see that nothing essential changes for the corner cases where $A = D$ or $E = Z$. Figure 16 is a more high-level version of Figure 15 where only the locations of the markers $\#_1$, $\#_3$, $\#_4$, $\#_5$ in the linear strand are shown.

Next, $f_5 \cup \overline{\#_3\#_5} \cup \overline{\#_1\#_4}$ is obtained from f_5 by adding the stickers/bridges $\overline{\#_3\#_5}$ and $\overline{\#_1\#_4}$, see Figure 17. Then f_6 is obtained through hybridization, which is shown in Figure 18. Notice that the immobilization of the positive strands is essential to avoid them hybridizing with themselves

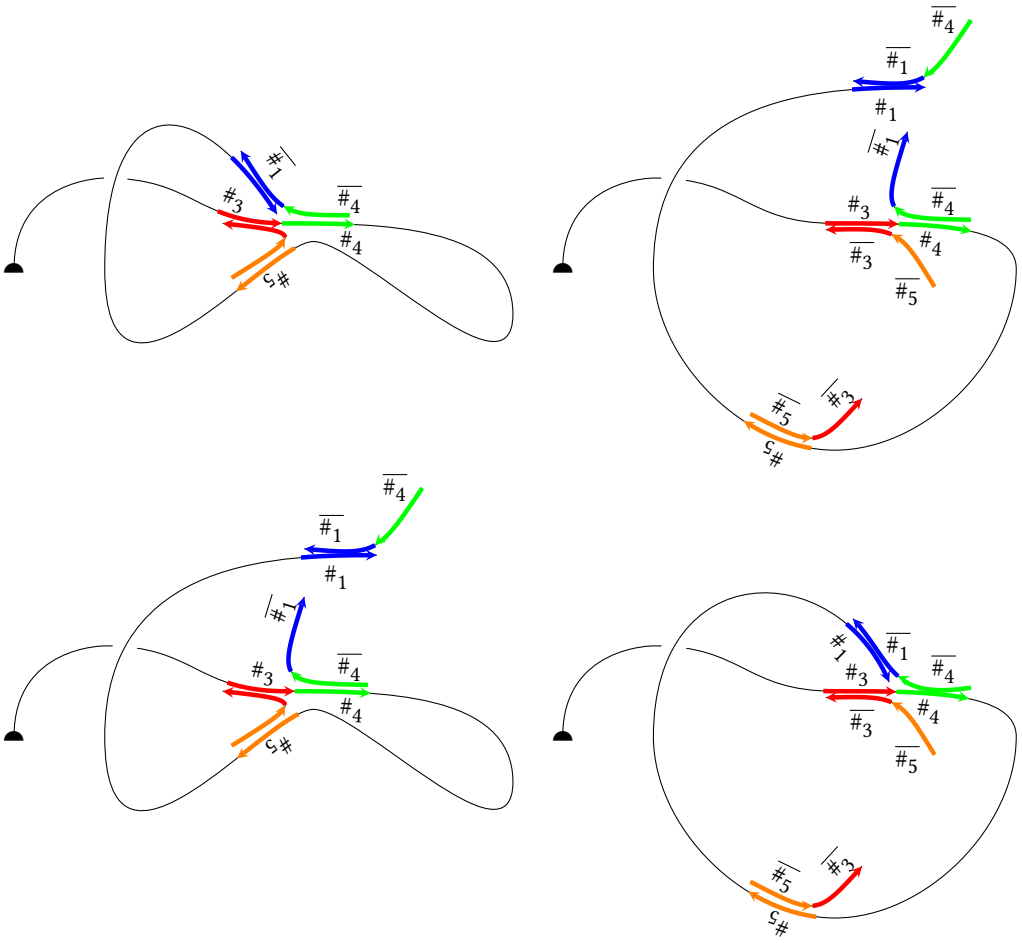

Fig. 18. The state after hybridization of Figure 17.

indirectly via these stickers. In fact, without immobilization, this hybridization would be not be terminating.

We will see that only components of the form depicted on the top-left of Figure 18 will influence the output complex; the other forms are undesirable side-effects and will be removed as a consequence of splitting and `cleanup`.

Next, the types of the largest components of $\texttt{split}(f_6, \#_3)$ are depicted in Figure 19. Notice that according to Table 1 `split` splits after tag $\#_3$. These two types result from splitting the two components on the left-hand side of Figure 18. Split after $\#_3$ for each component on the right-hand side of Figure 18 results in smaller pieces. The form of the largest components of $\texttt{split}(\texttt{split}(f_6, \#_3), \#_5))$ is depicted in Figure 20 and originates from the type of component on the left-hand side of Figure 19. Indeed, the type of component on the right-hand side is split in two. Also notice that according to Table 1 `split` indeed splits before tag $\#_5$.

Figure 21 then corresponds to the type of the largest components of $\texttt{ligate}(\texttt{split}(\texttt{split}(f_6, \#_3), \#_5))$. Finally, `cleanup` retains only the large linear strand (more precisely, the linear strands with this type), and we are done modulo the intermediate tag sequences $\#_3\#_5\#_6$ and $\#_6\#_1\#_4$

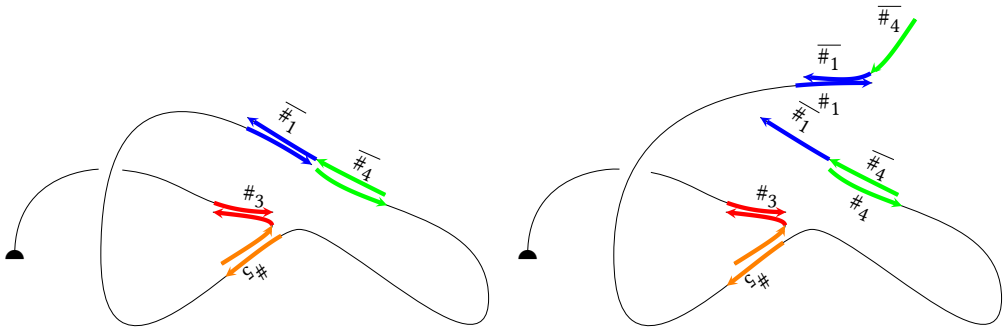

Fig. 19. The state of the largest type of components after applying splitting after $\#_3$ to Figure 18.

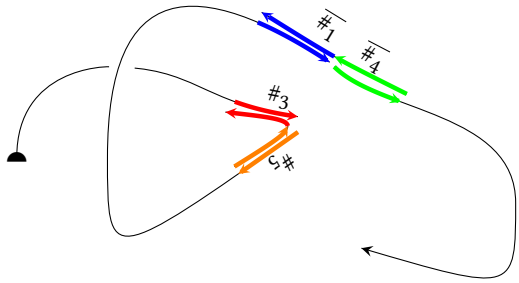

Fig. 20. The state of the largest type of components after applying splitting before $\#_5$ to Figure 19.

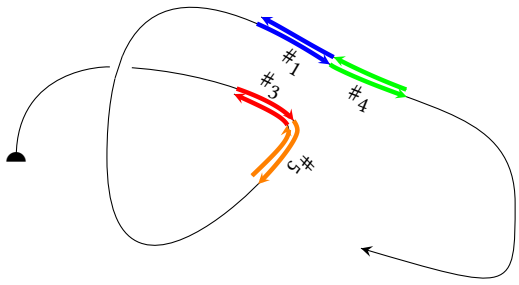

Fig. 21. The state of the largest type of components after applying ligation to Figure 20.

between the attribute-value blocks of B and E and between the attribute-value blocks of Z and D, respectively. According to Corollary 5.5 these can be removed by *removeBetweenIntervals*. □

By iteration of Lemma 5.10, we obtain the following result.

Theorem 5.11. *Any reordering of attribute-value blocks of all strands of a complex of relation-schema type is expressible in DNAQL.*

6 Proof of Theorem 4.3

The proof of Theorem 4.3 is by induction on the structure of expression e.

Variable. Suppose e is a relation variable x and $\mathcal{D} : x \vdash R$. In other words, $\mathcal{D}(x) = R$. We have $complex([\![x]\!](I))^< = complex(I(x))^< = [\![x]\!](complex(I)^<)$. Therefore, $e^{DNA} := x$ simulates e.

Union. Suppose $e = e_1 \cup e_2$ with $\mathcal{D} : e \vdash R$. By the induction hypothesis, there are DNAQL programs e_1^{DNA} and e_2^{DNA} that simulate e_1 and e_2, respectively, uniformly over all dimensions ℓ. Since $\mathcal{D} : e \vdash R$, the expressions e_1 and e_2 are also over the relation schema R. We have $complex([\![e_1 \cup e_2]\!](I))^< = complex([\![e_1]\!](I) \cup [\![e_2]\!](I))^<$. Recall that the definition of *complex*() is defined as the union of complexes of its tuples, and thus $complex([\![e_1]\!](I) \cup [\![e_2]\!](I))^< \equiv complex([\![e_1]\!](I))^< \cup complex([\![e_2]\!](I))^<$. By the induction hypothesis, this is in turn equivalent to $[\![e_1^{DNA}]\!](complex(I)^<) \cup [\![e_2^{DNA}]\!](complex(I)^<) = [\![e_1^{DNA} \cup e_2^{DNA}]\!](complex(I)^<)$. Therefore, $e^{DNA} := e_1^{DNA} \cup e_2^{DNA}$ simulates e.

Difference. The case of difference is completely analogous to that of union. Indeed, if $e = e_1 - e_2$ with $\mathcal{D} : e \vdash R$, then e_1 and e_2 are also over the relation schema R. We observe that $complex([\![e_1]\!](I) - [\![e_2]\!](I))^< \equiv complex([\![e_1]\!](I))^< - complex([\![e_2]\!](I))^<$ and similarly as for union we conclude that $e^{DNA} := e_1^{DNA} - e_2^{DNA}$ simulates e.

Cartesian product. Suppose $e = e_1 \times e_2$ with $\mathcal{D} : e \vdash R$. By the induction hypothesis, there are DNAQL programs e_1^{DNA} and e_2^{DNA} that simulate e_1 and e_2, respectively, uniformly over all dimensions ℓ. Since $\mathcal{D} : e \vdash R$, the expressions e_1 and e_2 are over disjoint relation schemas R_1 and R_2, respectively. We have $complex([\![e_1 \times e_2]\!](I))^< = complex([\![e_1]\!](I) \times [\![e_2]\!](I))^<$. By the definition of *complex*() and cartesian product, we observe that $complex([\![e_1]\!](I) \times [\![e_2]\!](I))^<$ is equivalent to $complex([\![e_1]\!](I))^< \cdot complex([\![e_2]\!](I))^<$ up to reordering of the attribute-value blocks (recall the definition of the $\cdot$ operator in Section 5.5). By the induction hypothesis, this is in turn equivalent to $[\![e_1^{DNA}]\!](complex(I)^<) \cdot [\![e_2^{DNA}]\!](complex(I)^<)$. By Lemma 5.6 this is equivalent to $[\![concatenate(e_1, e_2, (A_1, Z_1), (A_2, Z_2))]\!](complex(I)^<)$ for some attributes A_1, Z_1, A_2, and Z_2. By Theorem 5.11 any reordering of attribute-value blocks is expressible in DNAQL and so there is a DNAQL program e^{DNA} that simulates e.

Projection. Suppose $e = \hat{\pi}_D(e_1)$ with $\mathcal{D} : e \vdash R$. By the induction hypothesis, there is a DNAQL program e_1^{DNA} that simulates e_1 uniformly over all dimensions ℓ. In other words, $complex([\![e_1]\!](I))^<$ is equivalent to $[\![e_1^{DNA}]\!](complex(I)^<)$ for all ℓ-dimensional database instances I over $\mathcal{D}$. Observe that $complex([\![\hat{\pi}_D(e_1)]\!](I))^<$ is obtained from $complex([\![e_1]\!](I))^< \equiv [\![e_1^{DNA}]\!](complex(I)^<)$ by removing, for each component, the attribute-value block of D.

Let A and B be the first and last attributes of $R \cup \{D\}$, respectively, A' and B' be the first and last attributes of R, respectively, with respect to $<$, and, with respect to the linear order on $R \cup \{D\}$, X be the cyclic predecessor of D and Y be the cyclic successor of D. Note that by Remark 4.2, $|R| \geq 1$ and so $|R \cup \{D\}| \geq 2$. Therefore, these attributes are all well defined.

If $|R \cup \{D\}| \geq 3$, then X and Y are distinct and therefore, by Corollary 5.5, we obtain that $complex([\![\hat{\pi}_D(e_1)]\!](I))^<$ is equivalent to

$$[\![removeBetweenIntervals(e_1^{DNA}, A, B, A', B', ((X, Y)))]\!](complex(I)^<).$$

Finally, if $|R \cup \{D\}| = 2$, then it is straightforward to verify that

$$[\![\texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(\texttt{split}(e_1^{DNA}, \#_{\mathrm{v}|}) \cup \texttt{immob}(\overline{A}))))]\!](complex(I)^<),$$

with $R = \{A\}$ retains exactly the attribute-value blocks of A, as required.

So, in both cases there is a DNAQL program e^{DNA} that simulates e.

Renaming. Suppose $e = \rho_{D/D'}(e_1)$ with $\mathcal{D} : e \vdash R$. By the induction hypothesis, there is a DNAQL program e_1^{DNA} that simulates e_1 uniformly over all dimensions ℓ. Observe that $complex([\![\rho_{D/D'}(e_1)]\!](I))^<$ is obtained from $complex([\![e_1]\!](I))^< \equiv [\![e_1^{DNA}]\!](complex(I)^<)$ by replacing attribute D by attribute D' and then moving the attribute-value blocks of D' in the proper position according to the linear ordering $<$. By Remark 4.2, we have again $|R| \geq 1$.

If $|R| = 1$, then by the paragraph below the proof of Lemma 5.7, $complex([\![\rho_{D/D'}(e_1)]\!](I))^<$ is equivalent to $replaceAttributeUnary(e_1^{DNA}, D')$.

If $|R| \geq 2$, then, by Lemma 5.7, $complex([\![\rho_{D/D'}(e_1)]\!](I))^<$ is equivalent to

$$[\![replaceAttribute(e_1^{DNA}, A, Z; (B, D, E), D')]\!](complex(I)^<).$$

up to moving the attribute-value blocks of D' (the attributes A, B, E, and Z are as in Lemma 5.7). By Theorem 5.11, the movement of the attribute-value blocks of D' is expressible in DNAQL and so in both cases there is an program e^{DNA} that simulates e.

Selection. Suppose $e = \sigma_{B=E}(e_1)$ with $\mathcal{D} : e \vdash R$. By the induction hypothesis, there is a DNAQL program e_1^{DNA} that simulates e_1 uniformly over all dimensions ℓ. If $B = E$, then $e^{DNA} := e_1^{DNA}$ simulates e and we are done. Assume now that $B \neq E$. In particular, $|R| = 2$. Note that $complex([\![\sigma_{B=E}(e_1)]\!](I))^<$ is obtained from $complex([\![e_1]\!](I))^< \equiv [\![e_1^{DNA}]\!](complex(I)^<)$ by retaining exactly those components where the ℓ-cores corresponding to B and E are equal. By Lemma 5.9, $complex([\![\sigma_{B=E}(e_1)]\!](I))^<$ is equivalent to

$$[\![coreEquality(e_1^{DNA}, A, Z; (B, D), (E, F), i)]\!](complex(I)^<).$$

Therefore, there is a DNAQL program e^{DNA} that simulates e.

Note that e^{DNA} is effectively computable from e and $\mathcal{D}$ since in each of the above cases, we have either given e^{DNA} explicitly or described an effective method to obtain e^{DNA}.

We now establish the given length upperbound for e^{DNA}. Without cartesian product, e^{DNA} can be chosen to be linear in the length of e. Indeed, (1) we can avoid the duplication of expression e in operators like *replaceAttribute* by introducing a `let` construct, and (2) the use of $\bigcup_{a \in \Lambda} e_a$ in *coreEquality* merely increases the expression length by a constant since Λ is assumed fixed. For the general case, let k be the number of cartesian product operations in e. Each cartesian product involves a final step in which the attributes are sorted. Since sorting is of complexity $O(n \log n)$, we obtain that the length of e^{DNA} is $O(|e| + kn \log n)$, where n is the maximum cardinality among the schemas of subexpressions e' of e in which the final operation is the cartesian product.

7 Discussion

We have shown the relational completeness of DNAQL. We remark that a straightforward but rather long-winded verification shows that all results in this article can in fact be strengthened to well-typed DNAQL expressions/programs with respect to the type system for DNAQL as defined in [8]. Therefore, relational completeness also holds for the subclass of well-typed DNAQL programs.

By showing relational completeness we obtain a natural lower bound on the expressivity of DNAQL. We conjecture that well-typed DNAQL and the relational algebra have, in fact, equal expressivity.

Since there are well-known results connecting the relational algebra with other query languages (like Datalog, see, e.g., [1]), the result of this article, by transitivity, establishes connections between DNAQL and those query languages. Nevertheless, a establishing direct connection between DNAQL and, say, graph query languages is a worthwhile direction for future research as it might expose more, possibly unexpected, connections (e.g., regarding computational complexity).

While DNAQL focusses on faithful implementability in DNA, without relying on hard to implement notions such as nonterminating hybridization, verification of the feasibility of implementation in the wetlab remains an open problem. An interesting partial result would be the verification of the feasibility of the double bridging construction outlined in Section 5.8. Indeed, double bridging might be useful in other contexts where shuffling of DNA segments within a single DNA strand is required. A related research direction is to try to further simplify or restrict DNAQL to improve implementability while still maintaining relational completeness. For example, perhaps by clever programming we can make do with less than five recognition sites for restriction enzymes.

Another natural direction for further study is query optimization. Indeed, some operations in DNAQL are cheaper and faster to execute then others, and so it would be interesting to obtain methods that construct a DNAQL program with minimal cost among the DNAQL programs that are equivalent to some input DNAQL program. Related to this, it is interesting to see whether the obtained length upperbound of the program that simulates the relational algebra expression can be further minimized. In particular, is there an alternative construction of e^{DNA} such that its length is always linear in the length of e (that is, even if e contains cartesian product)?

Acknowledgments

We thank the anonymous reviewers for their valuable comments on an earlier version of this article.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. 1995. *Foundations of Databases*. Addison-Wesley Publishing Company Inc.

[2] M. Amos. 2005. *Theoretical and Experimental DNA Computation*. Springer.

[3] M. Arita, M. Hagiya, and A. Suyama. 1997. Joining and rotating data with molecules. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*. 243–248. DOI: https://doi.org/10.1109/ICEC.1997.592303

[4] James L. Banal et al. 2021. Random access DNA memory using boolean search in an archival file storage system. *Nature Materials* 20 (2021), 1272–1280. DOI: https://doi.org/10.1038/s41563-021-01021-3

[5] J. Bornholt, R. Lopez, D. Carmean, L. Ceze, G. Seelig, and K. Strauss. 2016. A DNA-based archival storage system. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, T. Conte and Y. Zhou (Eds.). ACM, 637–649. DOI: https://doi.org/10.1145/2872362.2872397

[6] R. Brijder, J. J. M. Gillis, and J. Van den Bussche. 2012. A type system for DNAQL. In *Proceedings of the 18th International Conference on DNA Computing and Molecular Programming*, D. Stefanovic and A. Turberfield (Eds.). Vol. 7433, Springer, 12–24.

[7] R. Brijder, J. J. M. Gillis, and J. Van den Bussche. 2013. Graph-theoretic formalization of hybridization in DNA sticker complexes. *Natural Computing* 12 (2013), 223–234.

[8] R. Brijder, J. J. M. Gillis, and J. Van den Bussche. 2021. DNAQL: A query language for DNA sticker complexes. *Natural Computing* 20 (2021), 161–189.

[9] Robert Brijder, Joris J. M. Gillis, and Jan Van den Bussche. 2013. The DNA query language DNAQL. In *Proceedings of the 16th International Conference on Database Theory*. Association for Computing Machinery, 1–9. DOI: https://doi.org/10.1145/2448496.2448497

[10] Luis Ceze, Jeff Nivala, and Karin Strauss. 2019. Molecular digital data storage using DNA. *Nature Reviews Genetics* 20 (2019), 456–466. DOI: https://doi.org/10.1038/s41576-019-0125-3

[11] G. M. Church, Y. Gao, and S. Kosuri. 2012. Next-generation digital information storage in DNA. *Science* 337, 6102 (2012), 1628.

[12] C. J. Date. 2004. *An Introduction to Database Systems*. Addison-Wesley.

[13] L. Diatchenko, Y. F. Lau, A. P. Campbell, A. Chenchik, F. Moqadam, B. Huang, S. Lukyanov, K. Lukyanov, N. Gurskaya, E. D. Sverdlov, and P. D. Siebert. 1996. 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 (1996), 6025–6030.
[14] Andrea Doricchi et al. 2022. Emerging approaches to DNA data storage: Challenges and prospects. *ACS Nano* 16 (2022), 17552–17571. DOI: https://doi.org/10.1021/acsnano.2c06748
[15] Alex El-Shaikh and Bernhard Seeger. 2023. Content-based filter queries on DNA data storage systems. *Scientific Reports* 13 (2023), 7053. DOI: https://doi.org/10.1038/s41598-023-34160-5
[16] Yaniv Erlich and Dina Zielinski. 2017. DNA fountain enables a robust and efficient storage architecture. *Science* 355, 6328 (2017), 950–954. DOI: https://doi.org/10.1126/science.aaj2038
[17] H. Garcia-Molina, J. D. Ullman, and J. Widom. 2009. *Database Systems: The Complete Book*. Prentice Hall.
[18] J. Gillis and J. Van den Bussche. 2010. A formal model for databases in DNA. In *Proceedings of the Algebraic and Numeric Biology*, K. Horimoto, M. Nakatsui, and N. Popov (Eds.). Lecture Notes in Computer Science, Vol. 6479, Springer, 18–37.
[19] Nick Goldman, Paul Bertone, Siyuan Chen, Christophe Dessimoz, Emily M. LeProust, Botond Sipos, and Ewan Birney. 2013. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Science* 494 (2013), 77–80.
[20] Kevin N. Lin, Kevin Volkel, James M. Tuck, and Albert J. Keung. 2020. Dynamic and scalable DNA-based information storage. *Nature Communications* 11 (2020), 2981. DOI: https://doi.org/10.1038/s41467-020-16797-2
[21] Q. Liu, L. Wang, A. G. Frutos, A. E. Condon, R. M. Corn, and L. M. Smith. 2000. DNA computing on surfaces. *Nature* 403 (2000), 175–179.
[22] Qian Ma, Chao Zhang, Mingzhi Zhang, Da Han, and Weihong Tan. 2021. DNA computing: Principle, construction, and applications in intelligent diagnostics. *Small Structures* 2, 11 (2021), 2100051. DOI: https://doi.org/10.1002/sstr.202100051
[23] A. Marathe, A. E. Condon, and R. M. Corn. 2001. On combinatorial DNA word design. *Journal of Computational Biology* 8, 3 (2001), 201–220.
[24] Lee Organick et al. 2018. Random access in large-scale DNA data storage. *Nature Biotechnology* 36 (2018), 242–248. DOI: https://doi.org/10.1038/nbt.4079
[25] G. Paun, G. Rozenberg, and A. Salomaa. 1998. *DNA Computing*. Springer.
[26] J. H. Reif. 1999. Parallel biomolecular computation: Models and simulations. *Algorithmica* 25 (1999), 142–175.
[27] G. Rozenberg and H. Spaink. 2003. DNA computing by blocking. *Theoretical Computer Science* 292, 3 (2003), 653–665.
[28] J. Sager and D. Stefanovic. 2006. Designing nucleotide sequences for computation: A survey of constraints. In *Proceedings of the DNA Computing*, A. Carbone and N. Pierce (Eds.). Lecture Notes in Computer Science, Vol. 3892, Springer Berlin, 275–289.
[29] M. R. Shortreed, S. B. Chang, D. Hong, M. Phillips, B. Campion, D. Tulpan, M. Andronescu, A. E. Condon, H. H. Hoos, and L. M. Smith. 2005. A thermodynamic approach to designing structure-free combinatorial DNA word sets. *Nucleic Acids Research* 33, 15 (2005), 4965–4977.
[30] S. M. H. Tabatabaei Yazdi, Y. Yuan, J. Ma, H. Zhao, and O. Milenkovic. 2015. A rewritable, random-access DNA-based storage system. *Scientific Reports* 5, 14138 (2015). DOI: https://doi.org/10.1038/srep14138
[31] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. 2007. A crash course in database queries. In *Proceedings of the 26th ACM Symposium on Principles of Database Systems*. ACM, 143–154.
[32] J. Van den Bussche and E. Waller. 2002. Polymorphic type inference of the relational algebra. *Journal of Computer and System Sciences* 64 (2002), 694–718.
[33] Karl-Heinz Zimmermann, Israel Martínez-Pérez, and Zoya Ignatova. 2008. *DNA Computing Models*. Springer. DOI: https://doi.org/10.1007/978-0-387-73637-2

Received 23 December 2024; revised 1 August 2025; accepted 16 November 2025