The DNA Query Language DNAQL

Robert Brijder, Joris J.M. Gillis,[*] and Jan Van den Bussche
Hasselt University & transnational University of Limburg
jan.vandenbussche@uhasselt.be

ABSTRACT

This paper presents an exposition of the authors' past and present work on the query language DNAQL for querying databases in DNA. In DNA computing, data is represented and stored in DNA molecules. Accordingly, a logical data model is defined that models complexes of DNA molecules in a graph-oriented fashion. Next, a set of formal operations on DNA complexes is defined, much in the spirit of the operations of the relational algebra in the relational data model. These operations model laboratory operations on DNA in solution. Their combination leads to the query language DNAQL; but in order for programs to be well-defined on prescribed types of inputs, a type system is superimposed on the language. Finally a correspondence is shown between well-typed DNAQL programs and programs in a relational-algebra query language.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data models*; H.2.3 [**Database Management**]: Languages —*Query languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*

General Terms

Theory

Keywords

DNA computing

1. INTRODUCTION

DNA Computing is an interdisciplinary scientific field, composed of mainly of computer scientists, chemists and physicists. Various good introductions to the field are available in books [24, 2], in survey articles [16], and on the Web. The field is often placed within the larger field of Natural Computing [28, 20, 26] and is also increasingly becoming associated with the related field of nanotechnology. Recent research results in DNA Computing can be obtained from the proceedings of the annual International Conference on DNA Computing and Molecular Programming, as well as the Annual Conference on Foundations of Nanoscience, and the specialized journal Natural Computing. We also recommend the annotated list of publications [11] from Erik Winfree's group on DNA and Natural Algorithms at Caltech, which is a leading laboratory in the field since the beginning.

DNA Computing became popular with Adleman's experiment [1] which gave a molecular solution to a small instance of the Hamiltonian Path problem. The two important lessons from this experiment are, first, that synthetic single-stranded DNA molecules can be used to represent structured data, such as the edges of a directed graph, and, second, that hybridization of two complementary single strands into a double strand (the famous double helix) can serve as a powerful computational primitive. Since the Hamiltonian Path problem is NP-complete in general, Adleman's experiment created the impression of DNA Computing as an approach to solving NP-complete problems. That view was quickly abandoned, however, as well as the view of DNA Computing as a platform for general-purpose computation, the latter mainly due to the difficulty of controlling the error rates. The currently dominating view of DNA Computing is that of a platform for programmed self-assembly at the nanoscale.

Yet the two main lessons from Adleman's experiment remain valuable, and the robust (almost indestructible) storage capacity of DNA at the nanoscale [14, 18] remains very tempting from the databases perspective. The potential use of single-stranded DNA as an addressable or searchable memory is indeed well known [4, 25, 13]. Databases, however, are much more than searchable memories: they are structured according to a logical data model such as the relational model, and are queried and manipulated using global operations on data such as the operations of the relational algebra. In this paper, we present an exposition of our recent work [17, 8, 9] on developing a logical data model for DNA computing, along with a tractable query language called DNAQL. We will also briefly mention some recent results that show a quite satisfying correspondence between the crucial operation of hybridization in DNA computing, and the crucial operation of join (or cartesian product) in the relational algebra.

[*]Ph.D. fellow of the Research Foundation–Flanders (FWO).

Our exposition is largely on an intuitive level; the full formal details can be found in our 67-page report [6] and a forthcoming paper on the expressive power of DNAQL. This work will also be part of Joris Gillis's forthcoming PhD thesis at Hasselt University.

2. DNA, HYBRIDIZATION, AND REPRESENTATION OF RELATIONAL DATA

Single strands of DNA are modeled as strings over the alphabet $\{A, C, G, T\}$. The individual positions in such a string are referred to as the *bases*. A very important concept is *Watson-Crick complementarity,* whereby A and T are complementary, as well as C and G. We indicate complementarity by overlining, writing $\bar{A} = T$, $\bar{T} = A$, $\bar{C} = G$, and $\bar{G} = C$. Complementation is extended to entire strings by complementing each base, then reversing the string. Thus, $\overline{AACTG} = \bar{G}\bar{T}\bar{C}\bar{A}\bar{A} = CAGTT$. Note that for any string we have $\bar{\bar{s}} = s$.

Two complementary single strands that encounter each other in a watery solution under natural conditions will *hybridize* to form a double-stranded duplex: the famous double helix. Hybridization happens through the process of base pairing, in which the two strands will align with each other in opposite directions allowing complementary base pairs to form bonds. The same happens more generally when a substring of one strand is complementary to a substring of the other strand. Thus, AAAACTG and AACAGTT will hybridize to the following:

```
AAAACTG
   TTGACAA
```

Here the two strands stick to each other in five base pairs.

A minimal number of consecutive base pairs is needed for the sticking to be robust; this number depends on the temperature. Indeed, base pairing can be undone by raising the temperature, and accordingly, higher temperatures are needed to undo the sticking of longer consecutive substrings of base pairs. The temperature at which a sticker detaches is called the *melting temperature* of the sticker; it actually depends not only on the length but also on the actual sequence. The process of going from double-stranded to single-stranded DNA is known as *denaturation.*

Working with just a four-letter alphabet $\{A, C, G, T\}$ is a bit awkward for data representation. Hence in practice we work with 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 DNA codewords, being crucial for successful DNA computing, is a research topic in itself [5, 15, 27, 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.

Atomic data values can thus be represented by fixing a finite alphabet Λ of *value bits* and representing atomic data values as strings of value bits. We can also naturally represent tuples as strings over a finite alphabet. Thereto, we introduce two additional finite alphabets Ω of *attributes* and Θ of *tags.* Attributes are, as in the relational data model, chosen depending on the application. Tags are used as additional "punctuation marks" that will later be seen to facilitate the manipulation of strings. In our work, it will suffice to use a fixed set of nine tags $\Theta = \{\#_1, \#_2, \#_3, \#_4, \#_5, \#_6, \#_7, \#_8, \#_9\}$. We denote the combined finite alphabet $\Lambda \cup \Omega \cup \Theta$ by Σ.

Now let t be a tuple over a relation scheme $R \subseteq \Omega$, where each attribute value $t(A)$, for $A \in R$, is a string of value bits. Let us put the attributes of R in some agreed upon order $A, \dots, B$. Then we agree to represent t as a string over Σ, denoted by $\mathit{complex}(t)$, defined by

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

So, the tuple is represented as a concatenation of attribute-value pairs, where each attribute-value pair is initiated with $\#_2$, then comes the attribute, and then comes the attribute value flanked by $\#_3$ on the left and $\#_4$ on the right. We use the terminology of "complex" because a string is a very special kind of complex as we will define later. Naturally we can now represent a *relation* r over R, i.e., a set of tuples over R, by a set of strings over Σ, also denoted by $\mathit{complex}(r)$, and defined by $\mathit{complex}(r) = \{\mathit{complex}(t) \mid t \in r\}$.

Here, a set of strings is a model for the content of a test tube containing the corresponding DNA single strands (typically in surplus quantities) in solution.

3. DATA MANIPULATION BY DNA COMPUTING

We have seen how relational data may be represented in DNA in a straightforward manner, but this alone is not satisfactory. We want to operate on the data using natural operations on DNA, most notably hybridization, but also others, such as splitting DNA strands, or filtering out strands satisfying some condition.

Note that in a test tube containing just the tuples of a relation and no other material, no spontaneous hybridization is possible. Indeed, all strands are composed of codewords only, so there is no complementary material present. To make this more formal, let us refer to Σ as the *positive alphabet* and let us consider the complementary *negative alphabet* $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$. So, each $\bar{a}$ is the Watson-Crick complement of a. Any string of negative letters can now be used to function as a sticker. In our work, however, we found that we can make do with stickers of at most two letters, so we formally define a *sticker* as a string consisting of just one or two negative letters.

The simplest use of stickers is to perform an operation known as *affinity separation* in the biotechnological world; in database terms we can think of it as a selection, as in the relational algebra. Let $a \in \Lambda$ be a value bit, and consider the sticker $\bar{a}$. Let us add surplus quantities of $\bar{a}$ into a test tube x containing a relation instance r, i.e., $x = \mathit{complex}(r)$. The stickers will bind, by hybridization, to all value bits a in the instance. If we now would have a way to retrieve all strands that are bound to a sticker, and afterwards are able to detach the stickers, we obtain the result of the selection $\sigma_{\theta_a}(r)$, where θ_a is the condition that one of the attribute values contains the value bit a.

3.1 Probing, flushing, and cleaning

In the above we assumed that one can effectively retrieve the strands that bind to a sticker. This can be achieved by *immobilizing* the stickers, prior to bringing them in contact with the strands. Here, by immobilizing we mean affixing to a physical substrate that we, humans, can handle mechanically, as opposed to pure free-floating molecules which are too small to handle directly. For example, stickers can be affixed to a surface, over which the solution can flow; this is what happens in so-called DNA chips. Or they can be affixed to microscopic magnetic beads, which are immersed in the solution but which can be recovered by applying a magnetic force.

All that is important for us is that, once a strand binds to an immobilized sticker, which we call a *probe*, the entire complex formed by sticker and strand now becomes immobilized, so that we can separate them from the strands that did not bind and thus are not immobilized. The operation of keeping only the immobilized complexes, washing away the rest, is a primitive operation in our model, called `flush`. Probes are distinguished constants in our model. Adding a probe to a test tube is accomplished by the union operation $\cup$, another primitive operation of our model. Finally, also the crucial hybridization reaction by which the strands bind to the probes is an explicit operation in our model. We can thus summarize the procedure discussed so far by the following DNAQL expression:

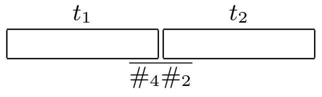
$$e_1 = \texttt{flush}(\texttt{hybridize}(x \cup \texttt{immob}(\bar{a}))).$$

It remains to detach the retrieved strands from the probes. This can be achieved by denaturing (raising the temperature), then again separating the free-floating complexes from the immobilized ones, but this time dismissing the latter and keeping the free-floating ones. In our model we call this operation `cleanup`. Thus the selection $\sigma_{\theta_a}(r)$ is expressed in DNAQL as $\texttt{cleanup}(e_1)$ with e_1 the expression from above. Actually, `cleanup` is more complicated in that it keeps from the free-floating strands only the longest ones. Biotechnicians can implement this using a procedure called *gel electrophoresis*.

3.2 Ligation and nontermination

In the previous section we have seen how length-one stickers can be used as probes, so that hybridization is used to compute selection-like queries. We next see how length-two stickers can be used to staple two strands together, so that hybridization is used to perform join-like operations.

Consider two relation instances r and s over disjoint relation schemes R and S respectively, and consider their representations in DNA $x = \mathit{complex}(r)$ and $y = \mathit{complex}(s)$. Let us take their union $x \cup y$ and add surplus quantities of the sticker $\overline{\#_4\#_2}$. Since each strand ends with #4 and begins with #2, the stickers will tie together pairs of strands. For example, let $R = \{A, B\}$ and $S = \{C, D\}$, then for any pair $t_1 \in r$ and $t_2 \in s$ the following complex will be formed by hybridization:



However, after denaturing (operation `cleanup`), the two tuples would be separate again, since they are not actually concatenated, but just held together by the sticker. To truly concatenate two strands that are already held together, one applies a natural enzyme called *ligase*. The application of ligase is another primitive operation of DNAQL, called `ligate`.

The above can be summarized by the DNAQL expression

$$e_2 = \texttt{cleanup}(\texttt{ligate}(\texttt{hybridize}(x \cup y \cup \{\overline{\#_4\#_2}\}))).$$

The composition of the three operations `hybridize`; `ligate`; and `cleanup` occurs often and we abbreviate it by `connect`, so e_2 would become $\texttt{connect}(x \cup y \cup \{\overline{\#_4\#_2}\})$. This program does not behave as desired, however. The attentive reader will note that the sticker might also stick to other occurrences of $\#_4$ and $\#_2$ than those at the end of one strand and the beginning of another. Moreover, nothing prevents the hybridization to go into a chain reaction, forming complexes not of two, but of three, or four, indeed any number of strands may be joined using multiple copies of the sticker. We say that this application of `hybridize` is *nonterminating.*

To avoid nontermination, we can give each strand from x a unique ending, and every strand from y a unique beginning, using the extra tags from the alphabet Θ that we have to our disposal. We can add a $\#_5$ at the end of each strand of x, and add a $\#_1$ in front of each strand of y, by the expressions $e_x = \texttt{connect}(x \cup \{\overline{\#_4\#_5}, \#_5\})$ and $e_y = \texttt{connect}(y \cup \{\overline{\#_1\#_2}, \#_1\})$. We can now safely concatenate pairs (t_1, t_2) of tuples t_1 from r and t_2 from s by the program

$$e_3 = \texttt{connect}(e_x \cup e_y \cup \{\overline{\#_5\#_1}\}).$$

3.3 Blocking and splitting

The program e_3 would correctly compute $\mathit{complex}(r \times s)$, were it not for the two tags $\#_5\#_1$ that sit in the middle of each strand. These tags were instrumental in avoiding nonterminating hybridization, but now we should get rid of them again. In biotechnology one employs *restriction enzymes* which can split complexes at designated points that occur before or after designated substrings. Since restriction enzymes are obtained from nature, there is only a limited repertoire of them available. Hence, a realistic abstract model of DNA computing should not simply assume that splitting can occur anywhere. In our work we have found we can make do with five split points, and perhaps by clever programming this number could even be reduced further.

To get rid of $\#_5\#_1$, we cannot simply split before $\#_2$ (which follows $\#_1$ in each strand in the result of e_3) and after $\#_4$ (which precedes $\#_5$), for then we would undo the concatenation of the two tuples. The solution is to circularize the strands before splitting them, so that the two tuples remain together after splitting. This is illustrated in Figure 1. Intuitively, we circularize by concatenating the end of t_2 to the beginning of t_1. However, we can not again avoid nonterminating hybridization as above, by introducing new tags in the middle, as our current job at hand is precisely to get rid of tags in the middle!

Instead, we now use the naive sticker $\overline{\#_4\#_2}$, but prevent it from sticking in undesirable places by *blocking* all bases except for the initial $\#_2$ of t_1 and the final $\#_4$ of t_2. By blocking a contiguous substrand, we mean to make it so that bases on a blocked substrand can no longer engage in further hybridization. A simply way to achieve this is by making the blocked substrand double-stranded. One applies

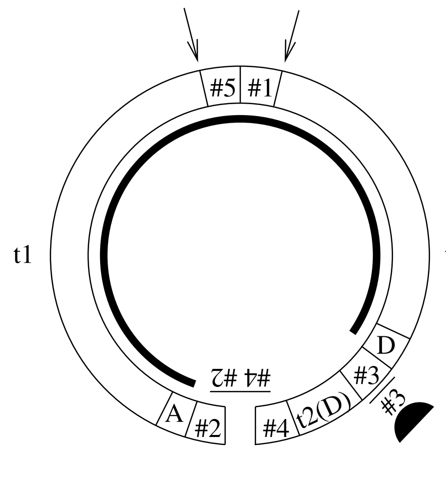


Figure 1: Getting rid of $\#_5\#_1$ by circularization. The figure shows many things at once that in reality happen consecutively. First, we block from A to D, indicated by the bold line. We then bind the strand to the probe `immob`$(\overline{\#_3})$, the immobilization indicated in the figure by a black half-disc. We add the sticker $\overline{\#_4\#_2}$ and let operation `connect` produce the circular strand. At that point we are safe in splitting at the two points indicated by the arrows.

the enzyme *polymerase* for this purpose.[1] In DNAQL we have two primitive operations for blocking: $\texttt{block}_a$ blocks each single occurrence of the letter a given as parameter, and $\texttt{blockfrom}_a$ blocks everything starting at every occurrence of a and stopping at any letter b that was already blocked earlier by an operation $\texttt{block}_b$ (or running until the end).

This blocking avoids the sticker to stick to unwanted occurrences of $\#_4$ and $\#_2$, but still does not avoid nonterminating hybridization. Our proposed solution is to block a little less than indicated above, from attribute A on t_1 until attribute D on t_2, and binding the only remaining free tag $\#_3$ (situated near the end of t_2) to a probe `immob`$(\overline{\#_3})$. Our model then makes the hypothesis, inspired by experimental results [3], that complexes bound to different probes cannot hybridize. As a result, there are only two possible effects for each strand: either the sticker circularizes the strand, or one copy of the sticker sticks to the beginning and another copy sticks to the end. The first effect is the desired one; the second effect is still undesired, but its resulting complexes can be removed later by splitting them up in little pieces which are then removed by a final `cleanup`. A basic assumption of our model is that every possible complex is present in surplus quantities, so we obtain the result of the first effect for each strand as well.

We omit the final detailed program for cartesian product; we also note that the procedure described above produces tuples with attributes in the order C, D, A, B. In order to do further operations, such as the difference operation, it is important to be able to shuffle the attributes in a different order. We have invented a procedure to do so, based on an original use of stickers. Since the procedure is rather intricate we omit it from the present paper [17, 6].

[1]To implement $\texttt{block}_a$ we hybridize with a dideoxy-variant of $\bar{a}$.

3.4 Difference

DNAQL includes a difference operation much like that of the relational algebra. In DNAQL we can only take the difference of two sets of DNA complexes on condition that each set consists exclusively of single strands, and all strands must have the same length. The implementation of this operator is based on a technique known as *subtractive hybridization.* This operation is expensive and error-prone, which draws an interesting parallel to the situation in the relational algebra, where queries that use the difference operator are typically harder to optimize.

3.5 Dimension and for-loops

With the DNA operations seen in the previous section, one can perform all the operations of the relational algebra, with the exception of equality selection. In Section 3.1 we have seen how the selection $\sigma_{\theta_a}(r)$ can be performed, but this is much weaker than a full equality selection $\sigma_{A=B}(r)$. To be able to do this we need to be more precise about how attribute values are represented. We have already agreed that they are represented as strings of value bits, but we have not been specific yet about the length of these strings. For any natural number ℓ, let us call *ℓ-instance* any database instance where all attribute values are words of length ℓ. We call ℓ the *dimension* of the instance; one may think of the dimension as the analogue of the word length in a digital computer. Of course, it will be important to allow instances of any dimension, since for any fixed dimension, and any fixed database schema, there are only a finite number of different instances of that schema.

By using ℓ additional marker codewords to indicate the positions of every value bit, we can then implement the special blocking operation $\texttt{blockexcept}_i$ as a composition of four `block` and `blockfrom` operations already seen above. This operation blocks, in any word of value bits of length ℓ, all value bits except the ith one. For all other purposes the marker codewords can be treated transparently, so we consider $\texttt{blockexcept}_i$ to be a primitive operation rather than an abbreviation like `connect` is.

Now crucially, DNAQL includes a *for-loop mechanism* that allows an expression to be iterated ℓ times while letting a counter variable i range from 1 to the dimension ℓ. The for-loop construct is crucial because it allows DNAQL programs to run on inputs of unknown, but well-defined, dimension.

Using a for-loop, a DNAQL program for $\sigma_{A=B}$ is now readily written. First, we make an obvious modification of the program for σ_{θ_a}, using `blockexcept`, to arrive at a program for $\sigma_{A=_ia}$, selecting all tuples t for which the ith value bit of $t(A)$ equals a. Then using that as a subroutine, the following program computes $complex(\sigma_{A=B}(r))$ on input $x = complex(r)$, where $r(A, B)$ is a relation instance and where for simplicity we assume a binary alphabet $\Lambda = \{0, 1\}$:

$$\texttt{for } y := x \texttt{ iter } i \texttt{ do}$$
$$\sigma_{B=_i0}(\sigma_{A=_i0}(y)) \cup \sigma_{B=_i1}(\sigma_{A=_i1}(y)).$$

The above for-loop should be read as follows: y is initialized to x; then y is repeatedly replaced by the value of the body expression, for $i = 1, \ldots, \ell$ where ℓ is the dimension of the input at hand.

We emphasize that our for-loops merely iterate a counter until the dimension of the instance. Thus they should not be confused with the much more powerful for-loop iteration mechanisms that iterate over the cardinality of the instance, as considered earlier in the theory of query languages [12, 21].

3.6 DNAQL

We have now introduced all the features of the language DNAQL. To summarize, the expressions of the language are built up as follows. The atomic expressions are variables representing the contents of test tubes; constants for single strands; constants for probes; and constants for length-two stickers. The following operations can be applied to build further expressions: union; difference; hybridize; ligate; flush; split; block; blockfrom; blockexcept; and cleanup. Finally the language has a let-construct to be able to store intermediate results in a new variable, and the for-construct described in the previous paragraph.

In this paper we omit the formal semantics of DNAQL [17, 6].

4. COMPLEXES

We have seen in the previous section how operations on DNA can perform data manipulation. During these manipulations intermediate structures are created. These structures, which we have referred to as *complexes*, consist of various strands and stickers, some connected up by base-pair bonding. Various fragments may be blocked, and some of the connected components of the complex may be immobilized by probes. In our model we assume that each connected component may be bound to at most one probe; this assumption is motivated by the physical distance that exists between two probes.

A good data model must be closed under its operations. Hence, it is not sufficient to think of a database instance as the complex representation of a relation instance as defined in Section 2. Instead we formally define the notion of complex and take this as our notion of database instance in our model. Complexes are finite directed graphs (V, L) where each node represents a position in a strand or sticker, and $L \subseteq V \times V$ represents the successor relations within the strands and stickers. In addition, each node v is labeled by a letter from $\Sigma \cup \bar{\Sigma}$, and there is also a partial matching on the nodes that indicates base pairing.[2] Finally, the complex includes two subsets β and ι of V that hold the blocked and the immobilized nodes, respectively.

For such a structure $(V, L, \lambda, \mu, \beta, \iota)$ to be a valid complex, however, additional restrictions must be satisfied:

1. (V, L, λ) must be a disjoint union of *positive strands* and *stickers*. Here, a positive strand is a linear or circular chain where all nodes are labeled with positive letters, and a sticker is a single node or a chain of length two where all nodes are labeled with negative letters.
2. Negative value bits cannot occur in length-two stickers.
3. If $\{u, v\} \in \mu$ then $\lambda(v) = \overline{\lambda(u)}$ (as expected from base pairing).
4. Nodes in β cannot occur in μ (since blocked nodes represent pieces of double strand).
5. The nodes in ι must be exactly the nodes of the single-node stickers (thus, probes are formalized as length-one stickers).
6. As already mentioned in the previous paragraph, each connected component (where connections can be made by μ as well as L) can contain at most one node in ι.

[2]A partial matching on V is a set μ of unordered pairs of elements of V such that each element of V occurs in at most one pair from μ.

These restrictions are satisfied by all the DNAQL data manipulations described in the previous section. Hence, our model of complexes provides a restricted, thus more predictable and tractable, setting for DNA computing, which still allows sufficient computation to take place so as to have at least the power of the relational algebra.

4.1 Weak types

The above-defined notion of complex serves as our notion of "database instance" in the DNA computing world. But then what is the corresponding notion of "database schema"? Recall that atomic data values are represented by strings of value bits. In a complex, such strings are visible as chains of consecutive nodes on a strand, with all nodes labeled by value bits. We consider nodes labeled by letters other than value bits, i.e., nodes labeled by attributes or tags, to serve more as markers rather than data carriers. Thus, such nodes serve as punctuation marks, as data structuring nodes, or as temporary markers during the computation. In other words, the structure of the nodes not labeled by value bits serves as the scaffolding of the complex.

The above observations suggest to consider as the "schema" of a complex C, the complex S obtained from C by abstracting away the nodes labeled by value bits, but keeping the other nodes intact. Formally, we define a *weak type* just like a complex, except that there is only one positive value bit symbol '$*$'. A single node labeled $*$ stands for a chain of consecutive nodes labeled by value bits. We refer to such chains as *data cores*. The definition of when a complex C *satisfies* a weak type S is then simply that every connected component of C can be obtained from a connected component of S by replacing each $*$-node by a data core.

For example, the complex representation $compex(r)$ of any relation instance $r(A, B)$ satisfies the weak type consisting of the single strand $\#_2 A \#_3 * \#_4 \#_2 B \#_3 * \#_4$.

5. TYPECHECKING DNAQL

Some of the operations of DNAQL are not always well-defined on all possible inputs. Indeed, the operations union, ligate, flush, split, and cleanup can be performed on any complex with a well-defined result. In contrast, the operations difference, hybridize, and the three blocking operations have only a well-defined result on inputs satisfying certain conditions, where the precise conditions depend on the operator. These conditions are so as to make the operation effectively implementable on real DNA using known techniques.

The condition for the difference of two complexes to be well defined is, roughly, that both complexes consist exclusively of positive strands all of the same length. The condition for the blocking operations to be well defined is that the input complex is *saturated* in the sense that no further hybridization is possible in the input complex, i.e., all base pairings that are possible in the input complex are realized.

The condition for hybridization to behave well is that it will not go into a chain reaction, what we have called nonterminating hybridization in Section 3.2. In our work [8, 7] we have characterized termination of hybridization applied to a complex C in terms of the absence of so-called *alternating cycles* in C. Formally, in any complex C, consider the following two kinds of moves that one may make to jump from one node to another. In what follows it must be understood that it is only allowed to move between nodes that are not blocked and not paired by μ; we call such nodes *free*. Now a *complementary move* is a move from a free node to another free node with a complementary label; a *component move* is a move from a free node to another free node belonging to the same connected component.

We have proved [8, 7]: *A complex C has nonterminating hybridization if and only if one can make a cycle in C by an alternating sequence of complementary moves and block moves, so that no node of an immobilized connected component is involved.*[3]

5.1 The weakness of weak types

That some operations are undefined on some inputs, as seen above, should not surprise us too much, as the same is true for conventional programming languages. For example, in Java, the operation `x.a` is only well defined if `x` points to an object that has an instance variable `a`. A similar situation is present in database query languages such as SQL or the relational algebra. For example, we can only take the union $r \cup s$ of two relations if r and s have the same relation scheme.

The standard solution to guarantee well-definedness is through type checking, and this is what we will do here as well. A DNAQL expression is said to be well defined on an input if, during the natural order of evaluation of all operations, starting with the given inputs, every operation is well defined on the inputs it receives from the results of its preceding operations. For a simple example, $\texttt{block}_a(\texttt{hybridize}(x))$ is well defined on all inputs, i.e., all possible assignments of a complex C to input variable x, because $\texttt{block}_a$ requires its input to be saturated, and the output of `hybridize` is trivially saturated.

In the relational algebra, if we know the relation schemes of the input relations, we can typecheck a relational algebra expression to see if it will be well defined on all input relations over the given schemes. For example, if x is a relation variable with scheme $\{A, B, C\}$, then $\sigma_{A=B}(\pi_{A,B}(x))$ is well typed but $\sigma_{A=C}(\pi_{A,B}(x))$ is not. We determine this by type inference [32]: inferring the scheme $\{A, B\}$ for the intermediate expression $\pi_{A,B}(x)$, then concluding that $\sigma_{A=B}$ is well defined on instances of that scheme but $\sigma_{A=C}$ is not. We would like to have a similar type inference mechanism for DNAQL.

Recall from Section 4.1 that weak types serve as schemes for complexes. They are a little bit too weak to guarantee well-definedness, however, as a weak type only gives the possible structures that may exist in a complex, without being tight in this respect. Consider, for example, the weak type S consisting of the single strand $\#_3 * \#_4$. Suppose x is a variable of type S, and consider the expression $e_4 = \texttt{hybridize}(x \cup \texttt{immob}(\bar{a}))$ for some value bit a. When x is assigned a concrete complex C and e_4 is evaluated, some strands in C may contain a data core containing a, and these will bind to the probe $\texttt{immob}(\bar{a})$. In this way connected components of weak type S' are formed, where S' is obtained by taking the disjoint union of S and a single-node complex representing the probe $\texttt{immob}(\bar{a})$, and pairing in μ the $*$-labeled node from S with the node from the probe. On the other hand, strands in C that do not contain a remain free in the result of e_4. These strands still have weak type S. Moreover, in case that no strand in C contain a, the probe will remain free as well. Hence, the best weak type we can infer for the result of e_0, given type S for x, is the weak type S_4 formed by the disjoint union of the three weak types S, S', and $\texttt{immob}(\bar{a})$. We conclude that complexes of weak type S_4 need not actually contain components of the three possible types. One may compare type S_4 with an untagged union type [10].

We now show that this aspect of uncertainty of weak types prevents typechecking of expressions that we want to allow. For a simple example, consider the constant expression $e_5 = \texttt{hybridize}(\#_2\#_4 \cup \overline{\#_4\#_5})$. The strand $\#_2\#_4$ obviously has itself as its weak type, and similarly for the sticker $\overline{\#_4\#_5}$. Hence the weak type for the union is simply the union of these weak types. This is the input type given to the type checker to check the application of `hybridize`. Since we have agreed that components part of weak types are not necessarily present in actual complexes of that type, the type checker must take into account the possibility that the sticker is actually not present, although in this particular constant expression we know it is present. The resulting weak type would thus still contain a free copy of the strand $\#_2\#_4$. This is bad because the larger expression $e_6 = \texttt{hybridize}(e_5 \cup \overline{\#_4\#_2})$ will now be rejected by the type checker, since a possible free strand ($\#_4\#_2$ in this case) and its complementary sticker are diagnosed as leading to nontermination hybridization. In fact, the evaluation of expression e_6 is well-defined.

5.2 Strong types

From the above it is clear that weak types have to be extended with a way of indicating that some of the components of the type are certain to occur in any complex of that type. In our work we have called these the *mandatory* components. Moreover, to allow satisfactory typechecking of the blocking operations, we have additionally extended weak types with a bit, called the *hybridization bit*, to indicate that complexes of that type are certain to be saturated (as that is the condition for the blocking operations to be well defined). Finally, to represent the result of blocking on the type level, we provide next to the symbol $*$, two further symbols $\underline{*}$ and $\hat{*}$ to represent a data core that is entirely blocked, or entirely blocked with the exception of one value bit (as resulting from an application of the `blockexcept` operation). It turns out that with these extensions (which we call *strong types*), all the operations of DNAQL can be evaluated on the type level, leading to a type inference algorithm for DNAQL.

Typechecking the cleanup operation.

To give some appreciation of what it means to evaluate DNAQL on the type level, we briefly discuss the `cleanup` operation. Recall that this operation first removes all base pairing and all blocking, and in a second step keeps only the longest strands. The second step is not entirely trivial to do on the type level. Consider, for example, the type S consisting of the two strand types $S_1 = A * A * A$ and $S_2 = AAAAA * AAAAA$ (for A an attribute). In any complex of

[3]In the cited papers, complementary moves are called edge moves and component moves are called block moves.

type S, of dimension ℓ, strands of type S_1 have length $3+2\ell$ whereas strands of type S_2 have length $10+\ell$. Hence, when $\ell = 7$, both types of strands are equally long; when $\ell < 7$, the strands of type S_2 win; otherwise, the strands of type S_1 win. We conclude that the result type of $e_7 = \texttt{cleanup}(x)$, when the type of x is S, equals S itself, with none of the strand types mandatory.

On the other hand, when we replace in the above discussion type S_2 by type $S_2' = AA * AA$, we see that strands of type S_1 will always be longest no matter what the dimension is. So, given the input type S'' which is like S but with S_2' instead of S_1, the result type of e_7 is S_1. Moreover, if in type S'' the component S_1 would be marked as mandatory, the type inference algorithm will propagate this to the output type.

Properties of the type inference algorithm.

Our type inference algorithm [9, 6] has the following desirable properties.

Soundness This is the expected property. We have proved that if our type system judges an expression $e(x_1, \dots, x_k)$ to well-typed for given input types $\tau_1, \dots, \tau_k$ for the input variables $x_1, \dots, x_k$, and infers a result type τ', then e will always have a well-defined evaluation on any input complexes $C_1, \dots, C_k$ of types $\tau_1, \dots, \tau_k$, and the result will be of type τ'. Importantly, this holds regardless of the dimension that the input complexes have, as long as they all have the same dimension.

Maximality This is a converse to soundness, but restricted to atomic expressions, i.e., expressions that consist of a single operation applied to variables. For atomic expressions we have proved that when the operator is well defined on all complexes of given input types, then the type system judges the expression well-typed under these input types.

Since well-definedness of general DNAQL expressions under given input types is undecidable, we cannot expect a full converse to soundness (for then the type-checking algorithm would be an algorithm for checking well-definedness). Well-definedness for DNAQL is undecidable because it is undecidable for the relational algebra [32], and we have seen in Section 3 that every relational algebra expression can be simulated by a well-defined DNAQL expression (actually, a well-typed DNAQL expression).

Tightness This is a property first considered by Papakonstantinou et al. in the context of type inference for XML transformations [22, 23]. We can prove the following. Let e again be an atomic expression, and assume that the type system infers an output type τ' for e, given input types τ. Assume furthermore that e, on inputs of type τ, always results in a complex of type τ''. Then τ' will be a subtype of τ''.

6. EXPRESSIVE POWER OF WELL-TYPED DNAQL

The design of DNAQL has been guided by the relational algebra: we wanted a language in which the data manipulations of the relational algebra can be expressed. And indeed, on complexes representing relational instances, we can implement every relational algebra operation by a well-typed DNAQL program. At the same time, however, we have been careful not to simply design a clone of the relational algebra. Indeed, DNAQL programs can work on much more general DNA complexes than just those that represent relational instances.

It remains to understand what the expressive power is of well-typed DNAQL programs in the general case. We can show a rather satisfying converse simulation of DNAQL by the relational algebra. Since this result still has to appear in a forthcoming paper, we only provide a brief sketch.

For the simulation to work, we need to replace equality selection $\sigma_{A=B}$ by value bit selection $\sigma_{A=_i a}$ as introduced in Section 3.5. As seen there, we can then express equality selection using value bit selection, if in addition we add DNAQL's for-loop construct also to the relational algebra, which is what we do. We thus obtain a relational algebra variant, which could be called the *string relational algebra*, suitable for relational databases where atomic data values are strings of value bits, of some uniform but arbitrary length (the dimension).

The simulation of typed DNAQL by the string relational algebra is based on a relational representation of complexes of a known type. Given a weak type T that is connected, we can form a relation scheme $\mathit{rel}(T)$ that uses as "attributes" the nodes of T labeled with $*$. Note that any complex C of type T is a disjoint union of connected components D, where each such D is of the form described by T. We can represent the data stored in D by a tuple t_D over $\mathit{rel}(T)$ that contains, for each $*$-labeled node of T, the data value stored in the corresponding data core in D. Then the relation consisting of all these tuples t_D represents the complex C. For a weak type S that has multiple connected components T, we use a relational database schema that has one relation scheme $\mathit{rel}(T)$ for each T and proceed similarly. This representation has to be further refined for strong types; we omit the details.

As a simple example, consider the operation $\texttt{hybridize}(x)$ where x is of type S consisting of two connected components: T_1 which is the strand $*\#_4$ and T_2 which is the following complex:

$$\begin{array}{r} \#_2 * \\ \overline{\#_4 \#_2} \end{array}$$

Here, the node from the upper strand labeled $\#_2$ is already matched by μ with the node labeled $\overline{\#_2}$ from the lower sticker. Then the result type contains the following connected component T:

$$\begin{array}{c} *\#_4 \#_2 * \\ \overline{\#_4 \#_2} \end{array}$$

It is now clear that for any complex C of type S, the relation for T in the relational representation of $\texttt{hybridize}(C)$ equals the cartesian product $r_1 \times r_2$, where r_i is the relation for T_i in the relational representation of C. We thus see that the main workhorse of DNA computing, namely hybridization, corresponds to the main workhorse of the relational algebra, namely cartesian product. When hybridizing with probes, the value bit selection operation is additionally used in the simulation.

A final caveat is that we cannot in general simulate every DNAQL expression by a single relational algebra expression that works across all possible dimensions for the input data. Since the result type of applications of `cleanup` can depend on the dimension, we must be satisfied with a finite case statement over the dimensions. A relational algebra expression is then given in each case, described by a simple linear condition on the dimension. For example, we have seen in Section 5.2 where the cases were $\ell < 7$, $\ell = 7$, and $\ell > 7$.

7. CONCLUSION

In this paper we have (informally) presented a formally defined data model and query language, in an attempt of defining the analogues of the relational model and the relational algebra in the world of DNA computing. Our model is a restriction, suitable for database manipulation, of what is known as Adleman's model of DNA computing. Meanwhile, various other models of DNA computing have come in vogue, such as the tile assembly model [16], and the model of chemical reaction networks, implemented in DNA by strand displacement [30]. It would be very interesting to repeat our effort for these other models.

Meanwhile, in further developing the model we have presented here, many challenges remain. An obvious one is its implementation and experimental validation. In this respect we should make clear that our descriptions of biotechnological protocols or operations on DNA have been very high-level, with the only purpose of giving the average reader a rough idea on how such operations may be achieved. Although we think it is plausible that DNAQL can in principle be implemented, actually doing so is a major topic for further research. We definitely anticipate technical obstacles to full implementation of DNAQL programs on arbitrary data, and our model will most likely have to be restricted or adjusted in the face of such implementation challenges.

One of the challenges of DNA computing is controlling the error rates. In this respect, it seems very interesting to us to develop a rigorous theory of database query languages that can make errors. We anticipate that similar techniques will play a role as have been developed for probabilistic querying [31].

Acknowledgment

Jan Van den Bussche would like to thank Anthony J. (Tony) Bonner for getting him interested in DNA, a long, long time ago. Tony suggested Jan to read *The Cartoon Guide to Genetics* [19] which Jan still highly recommends to all interested readers.

8. REFERENCES

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

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

[3] M. Arita, M. Hagiya, and A. Suyama. Joining and rotating data with molecules. In *Proceedings 1997 IEEE International Conference on Evolutionary Computation*, pages 243–248.

[4] E.B. Baum. Building an associative memory vastly larger than the brain. *Science*, 268:583–585, 1995.

[5] E.B. Baum. DNA sequences useful for computation. In L.F. Landweber and E.B. Baum, editors, *DNA Based Computers II: DIMACS Workshop, held June 10–12, 1996*, pages 235–242. American Mathematical Society, 1998.

[6] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. DNAQL: A query language for DNA sticker complexes. Available from `http://alpha.uhasselt.be/jan.vandenbussche/pubs.html`.

[7] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. Graph-theoretic formalization of hybridization in DNA sticker complexes. *Natural Computing.* In press.

[8] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. Graph-theoretic formalization of hybridization in DNA sticker complexes. In L. Cardelli and W. Shih, editors, *DNA Computing and Molecular Programming, 17th International Conference, DNA17*, volume 6937 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2011.

[9] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. A type system for DNAQL. In D. Stefanovic and A. Turberfield, editors, *DNA Computing and Molecular Programming*, volume 7433 of *Lecture Notes in Computer Science*, pages 12–24, 2012.

[10] P. Buneman and B. Pierce. Union types for semistructured data. In R.C.H. Connor and A.O. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming*, volume 1949 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 2000.

[11] Publications by the DNA and Natural Algorithms Group, California Institute of Technology. `http://www.dna.caltech.edu/DNAresearch_publications.html`.

[12] A.K. Chandra. Programming primitives for database languages. In *Conference Record, 8th ACM Symposium on Principles of Programming Languages*, pages 50–62, 1981.

[13] J. Chen, R.J. Deaton, and Y.-Z. Wang. A DNA-based memory with in vitro learning and associative recall. *Natural Computing*, 4(2):83–101, 2005.

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

[15] A.E. Condon, R.M. Corn, and A. Marathe. On combinatorial DNA word design. *Journal of Computational Biology*, 8(3):201–220, 2001.

[16] D. Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55(12):78–88, 2012.

[17] J.J.M. Gillis and J. Van den Bussche. A formal model of databases in DNA. In K. Horimoto, M. Nakatsui, and N. Popov, editors, *Algebraic and Numeric Biology, 4th International Conference, ANB 2010*, Lecture Notes in Computer Science, pages 18–37. Springer, 2012.

[18] N. Goldman et al. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature*, 2013. Published online, 23 January.

[19] L. Gonick. *The Cartoon Guide to Genetics.* HarperCollins, 1991.

[20] L. Kari and G. Rozenberg. The many facets of natural computing. *Communications of the ACM*, 51(10):72–83, 2008.

[21] F. Neven, M. Otto, J. Tyszkiewicz, and J. Van den Bussche. Adding for-loops to first-order logic. *Information and Computation*, 168(2):156–186, 2001.

[22] Y. Papakonstantinou and P. Velikhov. Enhancing semistructured data mediators with document type definitions. In *Proceedings 15th International Conference on Data Engineering*, pages 136–145. IEEE Computer Society, 1999.

[23] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings 19th ACM Symposium on Principles of Database Systems*, pages 35–46. ACM Press, 2000.

[24] G. Paun, G. Rozenberg, and A. Salomaa. *DNA Computing*. Springer, 1998.

[25] J.H. Reif et al. Experimental construction of very large scale DNA databases with associative search capability. In N. Jonoska and N.C. Seeman, editors, *Proceedings 7th International Meeting on DNA Computing*, volume 2340 of *Lecture Notes in Computer Science*, pages 231–247. Springer, 2002.

[26] G. Rozenberg, T. Bäck, and J.N. Kok, editors. *Handbook of Natural Computing*. Springer, 2012.

[27] J. Sager and D. Stefanovic. Designing nucleotide sequences for computation: A survey of constraints. In A. Carbone and N.A. Pierce, editors, *Proceedings 11th International Meeting on DNA Computing*, volume 3892 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2006.

[28] D. Shasha and C. Lazere. *Natural Computing: DNA, Quantum Bits, and the Future of Smart Machines*. Norton, 2010.

[29] M.R. Shortreed et al. A thermodynamic approach to designing structure-free combinatorial DNA word sets. *Nucleic Acids Research*, 33(15):4965–4977, 2005.

[30] D. Soloveichik, G. Seelig, and E. Winfree. DNA as a universal substrate for chemical kinetics. *PNAS*, 2010. Published online, 4 March.

[31] D. Suciu, D. Olteanu, Ch. Ré, and Ch. Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

[32] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. A crash course in database queries. In *Proceedings 26th ACM Symposium on Principles of Database Systems*, pages 143–154. ACM Press, 2007.