A Type System for DNAQL

A Type System for DNAQL

Robert Brijder, Joris J.M. Gillis*, Jan Van den Bussche

Hasselt University and transnational University of Limburg
Belgium

Abstract. Recently we have introduced a formal graph-based data model for DNA complexes geared towards database applications. The model is accompanied by the programming language DNAQL for querying databases in DNA. Due to natural restrictions on the implementability and termination of operations on DNA, programs in DNAQL are not always well defined on all possible inputs. Indeed, a problem left open by our previous work has been to devise a type system for DNAQL, with a soundness property to the effect that well-typed programs are well defined on all inputs adhering to given input types. The contribution of the present paper is to propose such a type system and to establish soundness. Moreover, we show that the type system is flexible enough so that any database manipulation expressible in the relational algebra is also expressible in DNAQL in a well-typed manner.

1 Introduction

Since Adleman's experiment [2], many different models for DNA computing have been invented and investigated, as can be learned from the books [3,16] and more recent developments [11,24,18]. At the same time, DNA computing has also high potential for database applications [4,7,25,20]. In this spirit, in recent work [13,5], we have defined the programming language DNAQL: a programming language specifically designed for the querying of databases in DNA. The goal of the present paper is to provide DNAQL with a sound type system.

DNAQL is a query language rather than a general-purpose programming language. It includes basic operators on DNA complexes in solution. Apart from the application of these operators, programs are formed using a let-construct and an if-then-else construct based on the detection of DNA in a test tube. Last but not least, the language includes a for-loop construct for iterating over the bits of a data entry, encoded as a vector of DNA codewords. Indeed, the number of operations performed during the execution of a DNAQL program, on any input, is bounded by a polynomial that depends solely on the dimension of the data, i.e., the number of bits needed to represent a single data entry. This makes that the execution time of programs scales well with the size of the input database.

A difficulty with DNAQL, and with DNA computing in general, however, is that various manipulations of DNA must make certain assumptions on their

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

input so as to be effectively implementable and produce a well-defined output. Even when these assumptions are well understood for each operation in isolation, the problem is exacerbated in an applicative programming language like DNAQL, where the output of one operation serves as input for another. Indeed the problem of deciding whether a given program will have well-defined behavior on all possible intended inputs is typically undecidable. While this undecidability is well known for Turing-complete programming languages, it remains so for database languages that are typically not Turing-complete [6].

The standard solution to ensure well-definedness of programs is to use a type system and check programs syntactically so as to allow only well-typed programs. Well-devised type systems have a soundness property to the effect that, once a program has been checked to be well-typed for a given input type, the behavior of the program is then guaranteed to be well defined on all inputs of the given type [17,14]. In the present paper, we propose a type system for DNAQL and establish a soundness theorem. Moreover, we show that the type system is flexible enough so that arbitrary relational databases can be represented as typed DNA complexes, and so that arbitrary relational algebra expressions on these data can be expressed by well-typed DNAQL programs. The relational algebra is the applicative language at the core of standard database query languages such as SQL [9,12,1].

We would like to make clear in what sense the present paper enhances previous work. That the relational algebra can be simulated in DNAQL has already been shown [13], but only insofar as the dynamic behavior at run-time is concerned. Here we show that the simulation can be syntactically guaranteed to be possible with well-typed DNAQL programs only. Also in recent work [5] we formulated a syntactic test on the well-definedness of hybridization, similar to weak satisfiability [15]. This syntactic test is but one component of the type system presented here, and here it is also extended to account for components of DNA complexes that are immobilized on separation surfaces such as magnetic beads.

Most importantly, a crucial feature of the type system presented here is a wildcard mechanism to account for the fact that the length (in bits), as well as the actual values, of data entries are unknown at compile time. This mechanism is integrated in a type-checking system that keeps track of mandatory components in DNA complexes, as well as their hybridization status. The result is a type system that allows a natural and flexible representation of structured data in DNA, in a way so that a significant class of data manipulations can be typed as programs in DNAQL.

2 Sticker Complexes

We recall [13,5] the data model of DNA sticker complexes, a graph-theoretically defined formalization of DNA complexes of a limited format geared towards data representation. Due to space limitations, we must be brief.

From the outset we assume a finite alphabet Σ. As customary in formal models of DNA computing [16], each letter represents a *string* over the DNA

alphabet $\{A, C, G, T\}$, such that the resulting set of sequences forms a set of DNA codewords [8,22,23]. This should always be kept in mind. The alphabet Σ is matched with its negative version $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$, disjoint from Σ. Thus there is a bijection between Σ and $\bar{\Sigma}$, which is called *complementarity* and is denoted by overlining; we also set $\bar{\bar{a}} = a$ so complementarity is symmetric. Obviously, $\bar{a}$ stands for the Watson-Crick complement of the DNA sequence represented by a. The elements of Σ are called *positive symbols* and the elements of $\bar{\Sigma}$ are called *negative symbols*.

For the purpose of data formatting we further assume that $\Sigma = \Lambda \cup \Omega \cup \Theta$ is composed of three disjoint parts: the set Λ of *atomic value symbols*; the set Ω of *attribute names*; and the set $\Theta = \{\#_1, \#_2, \#_3, \#_4, \#_5, \#_6, \#_7, \#_8, \#_9\}$ of *tags*.

The overall structure of a DNA complex is abstracted in the notion of *pre-complex*. Formally, a pre-complex is a 6-tuple $(V, L, \lambda, \mu, \iota, \beta)$, where

1. V is a finite set of nodes;
2. $L \subseteq V \times V$ is a set of directed edges without self-loops;
3. $\lambda : V \to \Sigma \cup \overline{\Sigma}$ is a total function labeling the nodes with positive and negative alphabet symbols;
4. $\mu \subseteq [V]^2 = \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$ is a partial matching on the nodes, i.e., each node occurs in at most one pair μ. Note that the pairs in μ are unordered.
5. $\iota \subseteq V$ is the set of *immobilized* nodes; and
6. $\beta \subseteq V$ is the set of *blocked* nodes.

Let C be a pre-complex as above. A *strand* of C is simply a connected component of the directed graph (V, L), so ignoring μ. The *length* of a strand is its number of nodes. A *sticker complex* (or *complex* for short) now is a pre-complex satisfying the following restrictions:

1. Each node has at most one incoming and at most one outgoing edge. Thus, each strand has the form of a chain or a cycle.
2. Strands are homogeneously labeled, in the sense that either all nodes are labeled with positive symbols, or all with negative symbols. Naturally, a strand with positive (negative) symbols is called a positive (negative) strand.
3. Every negative strand has length one or two; if it has length two, then it must have a single edge (i.e., it cannot be a 2-cycle). Negative strands are also referred to as "stickers".
4. Matchings by μ only occur between complementarily labeled nodes: formally, if $\{x, y\} \in \mu$ then $\lambda(y) = \overline{\lambda(x)}$.
5. A node can be immobilized only if it is the sole node of a negative strand.
6. Each component can contain at most one immobilized node.
7. Nodes in β do not occur in μ.

We see that the edges of a sticker complex indicate the sequence order within strands, and the matching μ makes explicit where stickers have annealed to positive strands. The predicate β represents longer stretches of double strands and

is used to restrict the places where hybridization can still occur [21]. Immobilized nodes represent probes attached to magnetic beads or surfaces that can be separated from the rest of the solution.

Components and redundancy. Two strands s and s' are *bonded* if there is a node v in s and some node v' in s' with $\{v, v'\} \in \mu$. When two strands are connected (possibly indirectly) by this bonding relation, we say they belong to the same component. Thus a *component* of a pre-complex is a substructure formed by a maximal set of strands connected by the bonding relation. Note that a component of a pre-complex is in itself a pre-complex. We use $comp(C)$ to denote the set of components of pre-complex C. Conversely, we can view a set of sticker complex components as a single sticker complex, basically by taking the union.

The intention of our model is that a complex defines the structural content of a test tube, which, however, will hold copies in surplus quantity of each component. Thus, each component of a complex stands for multiple occurrences. We formalize this using the notions of subsumption, equivalence, and minimality.

A pre-complex C_1 *is subsumed* by pre-complex 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. A component D in pre-complex C is *redundant* if there exists a component D' in C such that D and D' are isomorphic. Note that removing D from C yields an equivalent sticker complex. A pre-complex is *minimal* if there are no redundant components.

Saturated complexes. We call a complex C *saturated* if there do not exist any two nodes v and w such that adding the pair $\{v, w\}$ still results in a legal sticker complex. Intuitively, when a complex is saturated, hybridization is finished in the complex.

Representation of data entries. Dimension of a complex. The three disjoint parts of the alphabet $\Sigma = \Lambda \cup \Omega \cup \Theta$ serve distinct roles. Nodes labeled with tags from Θ indicate regions in the complex that have a function for data manipulation, as cleavage sites, or sites where stickers can anneal so as to circularize or concatenate strands. Nodes labeled with attribute names from Ω are used as annotations to data entries. Finally, the data entries themselves are represented using nodes labeled by atomic value symbols from Λ.

Atomic value symbols fulfill the same function as bits in a digital computer. A sequence of atomic value symbols represent a value, much like `100` is the binary representation of the number 8. Similar to the word size (number of bits) used in a digital computer to represent single data elements (such as integers), we will use sequences of atomic value symbols of a fixed length ℓ, called the dimension. Let $s = s_1 \dots s_\ell$ be a sequence of ℓ consecutive nodes of a strand of a sticker complex. If all nodes are labeled with atomic value symbols, s is called an *ℓ-core*. Let $s = s_0 \dots s_{\ell+1}$ be a sequence of $\ell + 2$ consecutive nodes of a strand of a sticker complex. Such a sequence is called an *ℓ-vector* if s_0 is labeled with $\#_3$, $s_{\ell+1}$ is labeled with $\#_4$ and $s_1 \dots s_\ell$ is an ℓ-core.

The *dimension* is now defined as follows. For a fixed value of ℓ, we say that sticker complex 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*.

3 DNAQL

DNAQL [13] is an applicative programming language for expressing functions from ℓ-complexes to ℓ-complexes. A crucial feature of DNAQL is that the same program can be applied uniformly to complexes of any dimension ℓ. DNAQL is not computationally complete, as it is meant as a query language and not a general-purpose programming language. The language is based on a basic set of operations on complexes, some distinguished constants, an emptiness test (if-then-else), let-variable binding, counters that can count up to the dimension of the complex, and a limited for-loop for iterating over a counter. The syntax of DNAQL is given in Figure 1. Note that expressions can contain two kinds of variables: variables standing for complexes, and counters, ranging from 1 to the dimension. Complex variables can be bound by let-constructs, and counters can be bound by for-constructs. The free (unbound) complex variables of a DNAQL expression stand for its inputs. A DNAQL *program* is a DNAQL expression without free counters. So, in a program, all counters are introduced by for-loops.

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

Fig. 1. Syntax of DNAQL.

The constant expressions provide particular complexes as constants. A word $w \in \Sigma^+$ stands for a single, linear, positive strand that spells the word w. A two-letter word $\bar{a}\bar{b}$, for $a, b \in \Sigma - \Lambda$, stands for a single, linear, negative strand of length two of the $1 \to 2$ with $\lambda(1) = \bar{b}$ and $\lambda(2) = \bar{a}$. The expression $\texttt{immob}(\bar{a})$, for $a \in \Sigma$, stands for a single, negative, immobilized node labeled $\bar{a}$: we call such a node a *probe*. The expression `empty` stands for the empty complex. The split operation is implemented by restriction enzymes. As the number of restriction enzymes is limited, and to ensure biological feasability of DNAQL, we allow only a limited number of split points.

The operation $\cup$ takes the disjoint union of two complexes. The difference $C - D$ of complexes C and D, which may be implemented using a subtractive hybridization technique [10], keeps only the strands of C that do not appear in D, and is only well defined when C and D consist solely of positive, equal-length strands. The operation `hybridize` performs hybridization as formalized [5] and extended here to take immobilized components into account, and may be undefined due to nonterminating behavior. Moreover, `ligate` behaves as ligase; `flush` removes supernatant (keeps only immobilized components), and `split` cleaves complexes. The blocking operations block a single node (`block`) or block a range starting from a primer (`blockfrom`); `blockexcept`(C, i) blocks, in each ℓ-vector $s_0, s_1, \ldots, s_\ell, s_{\ell+1}$ in the ℓ-complex C, all nodes except s_i. For the blocking operations to be well defined, the complex must be saturated. Finally, `cleanup` undoes matchings and blockings and removes all strands except the longest positive ones.

The for-loop iterates its body with the counter running from 1 to ℓ, thus allowing access to specific bits in data entries with the aid of the `blockexcept` construct.

Example 1. We give an example of a DNAQL program, over the input variables x_1 and x_2, with a behavior similar to the selection operator and the cartesian product operator from the relational algebra. Below, a and b are assumed to be atomic value symbols.

```
let y1 := cleanup(flush(hybridize(x1 ∪ immob(ā)))) in
let y2 := cleanup(flush(hybridize(x2 ∪ immob(b̄)))) in
if empty(y1) then empty else
if empty(y2) then empty else
cleanup(ligate(hybridize(y1 ∪ y2 ∪ #̄5#̄1)))
```

Assume complex C_1 holds a set of strands of the form $\#_3 * \#_4 \#_5$, where $*$ stands for a data entry in the form of an ℓ-core, and C_2 similarly holds a set of strands of the form $\#_1 \#_3 * \#_4$. Then the program applied to C_1 and C_2 filters from C_1 (C_2) the strands whose data entry contains the letter a (b); if both intermediate results are nonempty, the program then uses the stickers $\overline{\#_5 \#_1}$ to concatenate each remaining strand from C_1 with each remaining strand from C_2.

4 Sticker Complex Types

Intuitively, a sticker complex type is an ℓ-complex where all data entries have been replaced by wildcards. What remains is a structural description of the components that may appear in the complex, with attribute names and tags explicit, but the dimension and actual values of data entries hidden. In order to obtain a powerful type system for DNAQL, these "weak" types S are augmented to "strong" types that have an indication $\odot$ of the mandatory components, which must occur, and a bit $\mathfrak{h}$ indicating that the type is saturated. The former is needed to type common DNAQL programs that use hybridization, and the latter is needed to type blocking operators in a DNAQL program.

Formally, we begin by introducing four symbols assumed not present in $\Sigma \cup \bar{\Sigma}$:

1. $*$ (*free*) represents an ℓ-core with none of the nodes matched or blocked;
2. $\underline{*}$ (*blocked*) represents an ℓ-core with all nodes blocked;
3. $\hat{*}$ (*open*) is the result of a block-except operator on an ℓ-core;

Let N denote the set $\{*, \underline{*}, \hat{*}\}$. The positive alphabet without atomic value symbols, but with the above new symbols is denoted $\Sigma_N = \Omega \cup \Theta \cup N$.

The fourth new symbol, denoted by '?' will be used to represent a probe, i.e., a single negative atomic value symbol that has been immobilized. The negative alphabet without the negative atomic value symbols, but with ? is denoted $\overline{\Sigma_N} = \overline{\Omega} \cup \overline{\Theta} \cup \{?\}$. Note that ? is considered to be a negative symbol. The complementarity relation is extended by $\overline{*} =?$ and $\overline{\hat{*}} =?$. Complementarity is thus no longer a bijection, but a relation.

A *sticker complex type* 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 with the following exceptions:

- the range of the node labeling function λ is now $\Sigma_N \cup \overline{\Sigma_N}$ instead of $\Sigma \cup \overline{\Sigma}$;
- $\beta \subseteq V$ is not allowed to contain nodes labeled with a symbol from N;
- a node can be labeled '?' only if it is immobilized;
- there are no redundant components.

Next, we define the important notion of when a sticker complex $C = (V, L, \lambda, \mu, \iota, \beta)$ of some dimension ℓ is said to be well typed. Thereto, recall the intuitive meaning of the new symbols $\{*, \underline{*}, \hat{*}, ?\}$. Formally, consider an ℓ-core r occurring in C. We say that r is of type $*$ if no node of r is involved in μ nor in β; r is of type $\underline{*}$ if all nodes of r belong to β; and r is of type $\hat{*}$ if all but one node of r belong to β. Now we call C *well typed* if every ℓ-core occurring in C is of type $*$, $\underline{*}$ or $\hat{*}$. Moreover, if C is well typed, we define *stype*(C) as the sticker complex type obtained by replacing every ℓ-core occurring in C by a single node labeled by the type of the ℓ-core ($*$, $\underline{*}$ or $\hat{*}$), and replacing the label of any probe by ?.

The subsumption relation among sticker complexes (Section 2) can be adapted naturally to sticker complex types. We finally say that a well-typed sticker complex C *is of* some sticker complex type S, denoted by $C : S$, if *stype*(C) is subsumed by S. For sticker complex C, *stype*(C) is the "smallest" type, in the sense that there is no sticker complex type S' such that $C : S'$ and S' is strictly subsumed by S.

A sticker complex type is weak, in the sense that any well-typed sticker complex having as *stype* a subset of the components of a sticker complex type is of that type. In particular, the empty sticker complex is of every sticker complex type. This is too weak to type common DNAQL programs involving hybridization, where we need to know about components that are sure to be present. Thereto, we define a *strong sticker complex type* as a triple $\tau = (S, \odot, \mathfrak{h})$, where S is a sticker complex type, $\odot$ is a sticker complex type subsumed in S, $\mathfrak{h}$ is a boolean, and moreover if $\mathfrak{h} = \mathit{true}$, then $C \cup \odot$ is saturated for all every component C of S. Sticker complex type S is called the *weak type* of τ, the components of $\odot$ are called *mandatory* in τ, and $\mathfrak{h}$ is called the $\mathfrak{h}$-bit of τ.

A type τ is called *saturated* if all complexes having type τ are saturated.

For a well-typed complex C and a strong sticker complex type $\tau = (S, \odot, \mathfrak{h})$, we now say that C has type τ if C is of type S; the complex $\odot$ is subsumed by $stype(C)$; and C is saturated if $\mathfrak{h} = true$.

From now on, we will refer to sticker complex types as *weak types* and to strong sticker complex types as *types*.

5 A Type System for DNAQL

Given a DNAQL program $e(x_1, \ldots, x_k)$ with free complex variables $x_1, \ldots, x_k$, and given types $\tau_1, \ldots, \tau_k$ for the respective input variables, we would like to determine whether e is *safe* under these input types, meaning that for any dimension ℓ and for any input complexes $C_1, \ldots, C_k$ of dimension ℓ and of the given types $\tau_1, \ldots, \tau_k$, the result $e(C_1, \ldots, C_k)$ on these inputs is well defined. Since types do not restrict the dimension of complexes, if a type involves wildcards, there are infinitely many complexes of that type. Hence safety is not easy to guarantee, indeed safety is undecidable: this will follow from our later Theorem 2 and an easy reduction from satisfiability of well-typed relational algebra expressions, which is undecidable [1].

The best we can do is to come up with a type system that tries to infer the output types from given input types. We have developed a type system that, given e and $\Gamma = \tau_1, \ldots, \tau_k$ as above, determines whether e is *well-typed* under Γ, and, if so, infers an output type τ, this is denoted by $\Gamma \vdash e : \tau$. The DNAQL type system enjoys the following soundness property:

Theorem 1. *If $\Gamma \vdash e : \tau$ then e is safe under Γ, and the resulting complex of e applied to any inputs of type Γ will be of type τ.*

The full type-checking system and soundness proof are omitted from this conference paper. Here we give some intuitions and examples.

Obviously devising a sound type-checking system in itself is no challenge, as it suffices to judge every program ill typed so that soundness becomes trivial! The challenge is to have a sound type-checking system that still judges most useful DNAQL programs to be typed. Our type system checks DNAQL expressions bottom-up by applying the DNAQL operations on complexes symbolically, on the type level. The operations may fail on the type level, in case we cannot deduce from the type that the operation will be well-defined on all inputs of the given type. If we can deduce well-definedness, we output a tight result type and the type-checking continues. Furthermore, the typing inference made for if-then-else constructs, shown in Figure 2, are designed so as to maximally benefit from knowledge that complexes are nonempty. These rules maximally infer the presence of mandatory components, which allows later hybridization operations to be typed.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x := \tau_1] \vdash e_2 : \tau_1}{\Gamma \vdash \texttt{for}\ x := e_1\ \texttt{iter}\ i\ \texttt{do}\ e_2 : \tau_1}$$

$$\frac{\Gamma \vdash x : (S_x, \emptyset, \mathfrak{h}_x) \qquad S_x = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \texttt{if empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1}$$

$$\frac{\Gamma \vdash x : (S_x, \odot_x, \mathfrak{h}_x) \qquad \odot_x \neq \emptyset \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{if empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : (S_x, \odot_x, \mathfrak{h}_x) \qquad \odot_x = \emptyset \qquad |comp(S_x)| = 1 \\ \Gamma \vdash e_1 : (S_1, \odot_1, \mathfrak{h}_1) \qquad \Gamma[x := (S_x, comp(S_x), \mathfrak{h}_x)] \vdash e_2 : (S_2, \odot_2, \mathfrak{h}_2)\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{if empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : (S_1 \cup S_2, \odot_1 \cap \odot_2, \mathfrak{h}) \\ \mathfrak{h} = (S_1 \cup S_2 \text{ is saturated})\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : (S_x, \odot_x, \mathfrak{h}_x) \\ \odot_x = \emptyset \qquad |comp(S_x)| > 1 \qquad \Gamma \vdash e_1 : (S_1, \odot_1, \mathfrak{h}_1) \qquad \Gamma \vdash e_2 : (S_2, \odot_2, \mathfrak{h}_2)\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{if empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : (S_1 \cup S_2, \odot_1 \cap \odot_2, \mathfrak{h}) \\ \mathfrak{h} = (S_1 \cup S_2 \text{ is saturated})\end{array}}$$

Fig. 2. Typing relation for the control flow of DNAQL.

Example 2. Recall the program from Example 1 in Section 3. Consider the weak types $S_1 = \#_3{*}\#_4\#_5$ and $S_2 = \#_1\#_3{*}\#_4$. The program is well-typed under the types $\tau_1 = (S_1, S_1, \mathit{false})$ for x_1 and $\tau_2 = (S_2, \emptyset, \mathit{false})$ for x_2. Since S_1 is mandatory in τ_1, we know that input x_1 will be nonempty. Note also that the $\mathfrak{h}$-bit in τ_1 is false, although complexes of type S_1 are necessarily saturated. The subexpression $e_1 = \texttt{hybridize}(x_1 \cup \texttt{immob}(\bar{a}))$ is typed as $(S_1^?, \emptyset, \mathit{true})$, where $S_1^?$ consists of the following components: (i) S_1 itself; (ii) $\texttt{immob}(?)$; and the complex formed by the union of (i) and (ii) and matching the node $*$ with the node ?. Note that there are no mandatory components, since on inputs without an a, only (i) and (ii) will occur, whereas on inputs where all strands have an a, only (iii) will occur. The $\mathfrak{h}$-bit is now true since a complex resulting from hybridization is always saturated. Applying `flush` to e_1 yields output type $(S_1^{?\prime}, \emptyset, \mathit{true})$, where $S_1^{?\prime}$ consists of components (ii) and (iii) above. Finally the variable y_1 in the `let`-construct is assigned the type $(S_1, \emptyset, \mathit{true})$. Similarly, y_2 gets the type $(S_2, \emptyset, \mathit{true})$. Yet, by the design of the if-then-else typing rules, the subexpression on the last line of the program will be typed under the strong types $(S_1, S_1, \mathit{true})$ for y_1 and $(S_2, S_2, \mathit{true})$ for y_2. Because all components are now mandatory, the type inferred for subexpression $\texttt{hybridize}(y_1 \cup y_2 \cup \overline{\#_5\#_1})$ will be $(S_{12}, S_{12}, \mathit{true})$, where S_{12} is the weak type obtained from the union of S_1, S_2 and $\overline{\#_5\#_1}$ by matching the $\#_5$ and $\overline{\#}_5$ and the $\#_1$ and $\overline{\#}_1$ nodes, respectively. After ligate and cleanup the output type is (S, S, true) where S consists of the single strand $\#_3{*}\#_4\#_5\#_1\#_3{*}\#_4$. The final output type of the entire program, combining the then- and else-branches, is $(S, \emptyset, \mathit{true})$.

For another example, consider the program

$$\texttt{hybridize}(\texttt{hybridize}(x \cup \bigcup_{a \in \Lambda} \texttt{immob}(\overline{a})) \cup \overline{\#_3\#_4}).$$

This program is ill typed under the type $\tau = (S, S, \mathit{true})$ for x with $S = \#_3{*}\#_4$. Indeed, the nested hybridize subexpression is still well-typed, yielding the output type $(S^?, \emptyset, \mathit{true})$ without any mandatory components. Adding the component $\overline{\#_3\#_4}$ to $S^?$, however, yields a complex with nonterminating hybridization [5], so the type checker will reject the top-level hybridize.

Yet, this program will have a well-defined output on every input C of type τ. Indeed, every strand in C contains some $a \in \Lambda$, so the minimal type of the result of the nested hybridize will actually have a single complex component formed by the union of S and $\texttt{immob}(?)$ with $*$ and ? matched. Then the top-level hybridize will terminate since each sticker complex can have at most immobilized node.

This example shows that well-defined programs may be ill typed; this is unavoidable in general since safety is undecidable.

6 Relational Algebra Simulation

In this section we strengthen an earlier result [13] to the effect that relational algebra expressions can be simulated by DNAQL programs: we show that the simulation is already possible by *well-typed* programs. This illustrates the power of our type system (cf. the comment made after Theorem 1).

Basically we assume a universe $\mathbb{U}$ of data elements. A *relation schema* R is a finite set of attribute names. We can use the same alphabet Ω for these attribute names. A *tuple* over R is a mapping from R to U. A *relation instance* over R is a finite set of tuples over R.

The syntax of the relational algebra [9,12,1] is generated by the following grammar:

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

Here, x stands for a relation variable, and A and B stand for attributes. Our version of the relational algebra is slightly nonstandard in that our version of projection ($\widehat{\pi}$) projects away some given attribute, as opposed to the standard projection which projects on some given subset of the attributes.

The relational algebra obeys a simple type system where expressions are typed by relation schemes [6]. Given a relational algebra expression $e(x_1, \ldots, x_k)$ over the input relation variables $x_1, \ldots, x_k$, and given input relation schemas $\Gamma = R_1, \ldots, R_k$, we can determine whether e is well typed under Γ, and, if so, infer a result relation schema R, denoted by $\Gamma \vdash e : R$ or just $e : R$ if Γ is understood. The typing rules are simple. If $e_1 : R$ and $e_2 : R$ then $(e_1 \cup e_2) : R$ and $(e_1 - e_2) : R$; if $e_1 : R_1$ and $e_2 : R_2$ for disjoint R_1 and R_2, then $(e_1 \times e_2) : R_1 \cup R_2$; if $e : R$ and $A, B \in R$ then $\sigma_{A=B}(e) : R$; if $e : R$ and $A \in R$ then $\widehat{\pi}_A(e) : R \setminus \{A\}$; if $e : R$ and $A \in R$ and $B \notin R$ then $\rho_{A/B}(e) : (R \setminus \{A\}) \cup \{B\}$.

The semantics of the relational algebra is well known and we omit a formal definition. Provided $\Gamma \vdash e : R$, on any input relation instances $I_1, \ldots, I_k$ over $R_1, \ldots, R_k$, the result $e(I_1, \ldots, I_k)$ is well defined and is a relation instance over R.

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

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

A relation instance I of dimension ℓ is then represented by the ℓ-complex

$$\mathit{complex}(I) = \bigcup \{\mathit{complex}(t) \mid t \in I\}.$$

This $\mathit{complex}(I)$ is of strong type $\tau_R = (\mathit{complex}(R), \emptyset, \mathit{true})$, where $\mathit{complex}(R)$ is $\#_2 A \#_3 * \#_4 \ldots \#_2 B \#_3 * \#_4$.

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

Theorem 2. *Let $e(x_1, \ldots, x_k)$ be an arbitrary well-typed relational algebra expression, let $\Gamma = R_1, \ldots, R_k$ be input relation schemas, and let R be an output relation schema such that $\Gamma : e \vdash R$. Then e can be translated into a DNAQL program $e^{DNA}(x_1, \ldots, x_k)$, such that the following holds:*

1. *e^{DNA} is well-typed, i.e., $\tau_{R_1}, \ldots, \tau_{R_k} \vdash e^{DNA} : \tau_R$.*
2. *e^{DNA} simulates e uniformly over all dimensions ℓ, i.e., for each natural number ℓ and for any ℓ-dimensional input relation instances $I_1, \ldots, I_k$ over $R_1, \ldots, R_k$ respectively,*

$$e^{DNA}(\mathit{complex}(I_1), \ldots, \mathit{complex}(I_k)) = \mathit{complex}(e(I_1, \ldots, I_k))$$

(up to isomorphism).

The proof of the first statement is omitted in this conference paper. The second statement has been proven in previous work [13].

7 Conclusion

An interesting problem is to understand the precise expressive power of well-typed DNAQL programs. Theorem 2 provides a lower bound; a corresponding upper bound, to the effect that every well-typed DNAQL program can be simulated in the relational algebra (on relational structures representing the typed input complexes) would establish DNAQL as the DNA-computing equivalent of

the relational algebra. We note that untyped operations, e.g., the difference operator applied to arbitrary complexes of unknown type, are strictly more powerful than the relational algebra.

On the practical level, the obvious research direction is to verify some non-trivial DNAQL programs experimentally, or simulate them in silico. Indeed, we have gone to great efforts to design an abstraction that is as plausible as possible. A static analysis of the error rates of DNAQL programs on the type level is another necessary topic for further research.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Adleman, L.: Molecular computation of solutions to combinatorial problems. Science 226, 1021–1024 (1994)
3. Amos, M.: Theoretical and Experimental DNA Computation. Springer (2005)
4. Arita, M., Hagiya, M., Suyama, A.: Joining and rotating data with molecules. ICEC 1997.
5. Brijder, R., Gillis, J., Van den Bussche, J.: Graph-theoretic formalization of hybridization in DNA sticker complexes. DNA17.
6. Van den Bussche, J., Van Gucht, D., Vansummeren, S.: A crash course in database queries. PODS 2007.
7. Chen, J., Deaton, R., Wang, Y.Z.: A DNA-based memory with in vitro learning and associative recall. Natural Computing 4(2), 83–101 (2005)
8. Condon, A., Corn, R., Marathe, A.: On combinatorial DNA word design. Journal of Computational Biology 8(3), 201–220 (2001)
9. Date, C.: An Introduction to Database Systems. Addison-Wesley (2004)
10. Diatchenko, L., Lau, Y., et al.: Suppression subtractive hybridization: a method for generating differentially regulated or tissue-specific cDNA probes and libraries. PNAS 93(12), 6025–6030 (1996)
11. Dirks, R., Pierce, N.: Triggered amplification by hybridization chain reaction. PNAS 101(43), 15275–15278 (2004)
12. Garcia-Molina, H., Ullman, J., Widom, J.: Database Systems: The Complete Book. Prentice Hall (2009)
13. Gillis, J., Van den Bussche, J.: A formal model of databases in DNA. Algebraic and Numeric Biology 2010.
14. Gunter, C., Mitchell, J. (eds.): Theoretical Aspects of Object-Oriented Programming. MIT Press (1994)
15. Jonoska, N., McColm, G., Staninska, A.: On stoichiometry for the assembly of flexible tile DNA complexes. Natural Computing 10(3), 1121–1141 (2011)
16. Paun, G., Rozenberg, G., Salomaa, A.: DNA Computing. Springer (1998)
17. Pierce, B.: Types and Programming Languages. MIT Press (2002)
18. Qian, L., Soloveichik, D., Winfree, E.: Efficient Turing-universal computation with DNA polymers. DNA16.
19. Reif, J.: Parallel biomolecular computation: models and simulations. Algorithmica 25(2–3), 142–175 (1999)
20. Reif, J., et al.: Experimental construction of very large scale DNA databases with associative search capability. DNA7.

21. Rozenberg, G., Spaink, H.: DNA computing by blocking. TCS 292, 653–665 (2003)
22. Sager, J., Stefanovic, D.: Designing nucleotide sequences for computation: A survey of constraints. DNA11.
23. Shortreed, M., et al.: A thermodynamic approach to designing structure-free combinatorial DNA word sets. Nucleic Acids Research 33(15), 4965–4977 (2005)
24. Soloveichik, D., Seelig, G., Winfree, E.: DNA as a universal substrate for chemical kinetics. PNAS online (2010).
25. Yamamoto, M., et al.: Development of DNA relational databases and data manipulation experiments. DNA12.