DNAQL: A QUERY LANGUAGE FOR DNA STICKER COMPLEXES

ROBERT BRIJDER , JORIS J.M. GILLIS*, AND JAN VAN DEN BUSSCHE

Abstract. DNA computing has a rich history of computing paradigms with great expressive power. However, far less expressive power is needed for data manipulation. Indeed, the relational algebra, the yardstick of database systems, is expressible in first-order logic, and thus less powerful than Turing-complete models. Turing-complete DNA computing models have to account for many and varied scenarios. A DNA implementation of data manipulations might be nimbler and perform its operation faster than a Turing-complete DNA computing model. Hence, we propose a restrictive model for implementing data manipulation operations, focused on implementability in DNA. We call this model the *sticker complex* model. A forte of the sticker complex model, is its ability to detect when hybridization becomes an uncontrolled chain reaction. Such chain reactions make hybridization less predictable and thus less attractive for deterministic computations. Next, we defined a query language on sticker complexes, called *DNAQL*. DNAQL is a typed, applicative functional programming language, powerful enough to simulate the relational algebra on sticker complexes. The type system enjoys a number of desirable properties such as soundness, maximality, and tightness.

Key words. DNA computing, DNA database, Hybridization, Type system, Sticker complex model, DNAQL

AMS subject classifications. 68Q02 92B02

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, 19] and more recent developments [12, 26, 21]. At the same time, DNA computing has also high potential for database applications [4, 8, 28, 23]. Computational models in the DNA computing field aim to be Turing-complete. As a result, models are either hard to implement in the wetlab or they offer such basic primitives that writing a program is vexing.

The *sticker complexes model* is a restricted subclass of DNA complexes, aimed to be both practically viable and theoretically tractable. Special care has been taken to keep hybridization in check. In the sticker complex model a clear distinction is made between long data strands and short stickers, used to manipulate the data strands. Likewise, double-strandedness has a dual abstraction: a distinction is made between short duplexes formed by the interaction of stickers and longer data strands, and long duplexes initiated to withhold parts of data strands from participation in future hybridizations.

Sticker complexes represent the structural content of a test tube. We assume that each component of a sticker complex is redundantly present in a tube. If a DNA complex can hybridize to itself, it can hybridize as well to an identical copy. Often, the copy can hybridize with yet another copy and so forth. We identify this undesirable behavior as non-terminating hybridization. Non-terminating hybridization leads to infinite sticker complexes. In practice, when we have termination of hybridization, a test tube prepared with sufficient quantities of each component of the complex holds, in principle, sufficient material to produce all molecular species that can be the result of hybridization. If sufficient quantities are present, adding even more material will not yield new results. Of course, in practice, a test tube is always finite and the hybridization reaction will, under normal conditions, always "terminate" (reach equilibrium). But the point is that, when hybridization does not terminate

*Ph.D. fellow of the Research Foundation - Flanders (FWO) for a complex, adding ever more material can, in principle, result in ever more new molecular species (MHE components) to be produced. In this sense, the potential result of the hybridization is indeed infinite. Fortunately, it is efficiently decidable for a sticker complex whether it has terminating hybridization.

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 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 [27].

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 [20, 15]. In the present paper, we propose a type system for DNAQL and establish a soundness theorem. In addition, the type system is maximal and tight. That is, if an operation is defined on all complexes of a certain type, the operation's counterpart on types is defined on the considered type. In other words, the type system only forbids the application of an operation if there is a reason to. Secondly, the type system might output types that are too loose, in the sense that the type outputted by an operation can be slimmed down without jeopardizing the soundness of the type system. We prove that the types produced by the type system are not too loose, i.e., the type system is tight. 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, 13, 1].

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 algorithm 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.

Extended abstracts of the DNAQL programming language and its type system were presented, containing selected results often without proofs, at the ANB, DNA 17, and DNA 18 conferences [14, 6, 5].

2. Sticker Complexes. In this section, we present a slightly modified version of the sticker complex model [14]: labels have shifted from the edges to the nodes.

2.1. Alphabet. From the outset we assume a finite alphabet Σ. As customary in formal models of DNA computing [19], 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 [17, 24, 25]. This should always be kept in mind. The alphabet Σ is matched with its negative version $\overline{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$, disjoint from Σ. Thus there is a bijection between Σ and $\overline{\Sigma}$, which is called *complementarity* and is denoted by overlining. 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 $\overline{\Sigma}$ 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.*

2.2. Pre-Complex. We define pre-complexes to contain the overall structure of sticker complexes. A pre-complex is a finite, node-labeled, directed graph where the nodes represent bases in strands and edges indicate direction. Moreover, a pre-complex is equipped with a matching, representing base pairing, and two predicates. One predicate indicates which bases are "immobilized", i.e., do not float freely and can be separated from solution in a controlled manner; the other predicate indicates which bases are "blocked", i.e., cannot participate in base pairing. Formally, a pre-complex is a 6-tuple $(V, L, \lambda, \mu, \iota, \beta)$, 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 total function labeling the nodes with positive and negative alphabet symbols;
- $\mu \subseteq [V]^2 = \{\{u, v\} \mid u, v \in V \wedge u \neq v\}$ is a partial matching on the nodes, i.e., each node occurs in at most on pair;
- $\iota \subseteq V$ is the set of *immobilized* nodes; and
- $\beta \subseteq V$ is the set of *blocked* nodes.

A connected component induced by the edges of L is called a *strand.* The *length* of a strand s, denoted by $|s|$, is the number of edges of L that belongs to s. By *strands*(S) we denote the set of positive strands of pre-complex C.

Both the partial matching μ as the predicate β serve to abstract the notion of double-strandedness. The matchings make explicit where the negative strands are bonded to the positive strands. The predicate β represents longer stretches of double strands.

Components. 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 pre-complex components as a single pre-complex, basically by taking the union. For convenience, whenever it is clear from the context we write $D \in C$

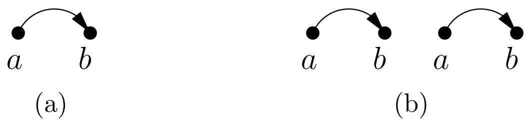


FIGURE 2.1. *An example of two pre-complexes that are non-isomorphic but that are equivalent. These pre-complexes are not isomorphic, as (b) contains twice the number of nodes of (a).*

for a component D and a pre-complex C to denote that D is a component of C, i.e., $D \in comp(C)$.

Subsumption and redundancy. The intention of the model is that a complex defines the structural content of a test tube. A test tube will, however, hold copies in surplus quantity of each component. Thus, each component of a complex stands for multiple occurrences. Two identical components in a pre-complex are thus meaningless. We formalize this using the notions of subsumption, equivalence, and minimality.

A pre-complex C_1 *is subsumed* by pre-complex C_2, denoted by $C_1 \sqsubseteq C_2$, 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, denoted $C_1 \equiv C_2$. 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.

Note that the notions of isomorphism and equivalence are not equal. Indeed, some pre-complexes can be simultaneously non-isomorphic and equivalent, as shown in Figure 2.1.

2.3. Sticker Complex. A *sticker complex* is a pre-complex abiding the following requirements:

1. Each node has at most one incoming and one outgoing edge. Thus each strand has the form of a chain or a cycle.
2. The labels on a chain are “homogeneous”, in the sense that either all nodes are labeled with positive symbols or all nodes are labeled with negative symbols. Naturally, a strand with positive (negative) symbols is called a positive (negative) strand.
3. Negative strands are severely restricted: specifically, every negative strand must be a chain of one or two nodes.
4. Matchings by μ only occur between nodes with complementary labels.
5. Nodes in β 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.

A node u is called *free* if u neither occurs in β nor in μ, and is called *closed* if it is not free. Nodes u and v are called *mutually interacting* if (1) they are both free, (2) u and v are complementary labeled, and (3) u and v do not belong to different immobilized components (i.e., components containing an immobilized node).

Isomorphism of sticker complexes can be decided in polynomial time by depth-first search. Indeed, if C and C' both consist of a single component, v is a node of C, and v' is a node of C', then there is at most one isomorphism from C to C' mapping v to v', and this isomorphism can be traced out by depth-first search, following the chain or cycle shape of strands, and the partial matching μ. Depth-first search is in linear time, which yields an isomorphism check for single components in cubic time (try all combinations of v and v'). This algorithm then easily extends to complexes C and

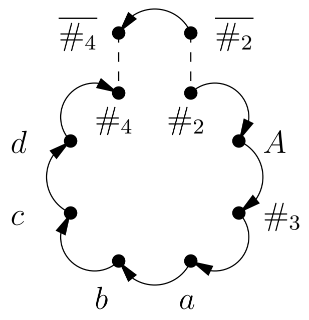


FIGURE 2.2. *A sticker complex with one component. The long positive strand is being circularized by the short sticker* $\overline{\#_4\#_2}$.

C' with multiple components, by matching the components of C to the components of C'. This efficient isomorphism check is in contrast to the problem of general graph isomorphism, which is not known to be decidable in polynomial time. We thus see that sticker complexes form a restricted family of graphs. As a consequent of the efficient isomorphism checking algorithm, the algorithm for minimizing a sticker complex also has polynomial time complexity.

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 on a computer. 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 notion of dimension is now defined as follows. For a fixed value of $\ell \geq 2$, we say that sticker complex C has *dimension* ℓ, if all nodes labeled with an atomic value symbol occur in an ℓ-vector. Note that we do not consider the one-dimensional case.

From now on, we will refer to sticker complexes simply as complexes, and to sticker complexes of dimension ℓ as ℓ-complexes.

EXAMPLE 2.1. *Figure 2.2 shows a sticker complex with one component. The arrowed edges represent L. The dashed edges represent matchings in μ. The long positive strand is being circularized by the short sticker complex* $\overline{\#_4\#_2}$.

3. Operations on Sticker Complexes.

In this section, we recall from [14] a set of operations on complexes that are rather standard in the DNA computing literature, except perhaps the difference. But what is interesting, however, is that we have defined sticker complexes in such a way that each operation always results in a sticker complex when applied to sticker complexes. Moreover, several operations impose additional restrictions on the input, so as to guarantee effective implementability in real DNA.

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

Union. Let $C_1 = (V_1, L_1, \lambda_1, \mu_1, \iota_1, \beta_1)$ and $C_2 = (V_2, L_2, \lambda_2, \mu_2, \iota_2, \beta_2)$ be two complexes. Without loss of generality we assume that V_1 and V_2 are disjoint. Then the union $C_1 \cup C_2$ equals $(V_1 \cup V_2, L_1 \cup L_2, \lambda_1 \cup \lambda_2, \mu_1 \cup \mu_2, \iota_1 \cup \iota_2, \beta_1 \cup \beta_2)$.

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

1. $\mu_1 = \iota_1 = \beta_1 = \emptyset = \mu_2 = \iota_2 = \beta_2$, i.e., all components in C_1 and C_2 are single strands.
2. All strands of C_1 and C_2 are positive, non circular, and all have the same length.
3. Each strand of C_2 ends with $\#_4$ and does not contain $\#_5$.

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

Hybridize. Let $C = (V, L, \lambda, \mu, \iota, \beta)$ and $C' = (V', L', \lambda', \mu', \iota', \beta')$ be two complexes. We say that C' is a *hybridization extension* of C if $V = V'$, $L = L'$, $\lambda = \lambda'$, $\iota = \iota'$, $\beta = \beta'$ and μ' is an extension of μ. Beware that a hybridization extension must satisfy all conditions from the definition of sticker complex. A complex C' is said to be *saturated* if the only hybridization extension of C' is C' itself.

The notion of hybridization extension is not sufficient, however, since we want to allow duplicate copies of components in C to participate in hybridization. (This important issue is glossed over in Reif's formalization [22].)

Let C and C' again be complexes. We call C' a *redundant variation* of C, simply if C subsumes C'. Note that C' may contain redundant components. Hence, the recipe to produce a redundant variation is simply to take, for every component of C, zero, one, or more copies.

Hybridization is now defined in terms of *multiplying hybridization extensions (MHEs)*, which, by applying redundant variations, account for the presence of surplus copies of components participating in the hybridization. Let C and C' again be two complexes. We call C' an MHE of C if C' is a hybridization extension of some redundant variation C'' of C.

The notion of MHEs is invariant under equivalence, both on the input side as on the output side:

PROPOSITION 3.1. *Let C_1 and C_2 be two equivalent complexes.*

1. *A complex C' is an MHE of C_1 if and only if C' is an MHE of C_2.*
2. *C_1 is an MHE of a complex C if and only if C_2 is an MHE of C.*

We are not quite finished with the notion of MHE, however. Indeed, an MHE may have "unfinished" components. Formally, we call a component D of an MHE *unfinished* if there exists another MHE in which D occurs bonded within a larger component; otherwise it is called *finished.* An MHE of a complex C, without any unfinished components is called *saturated with respect* to complex C. Note that if C is saturated, all MHEs are equivalent to C.

A fundamental issue is that the result of hybridization may be infinite, as shown next.

EXAMPLE 3.1. *Consider the simple complex consisting of two strands ab and $\bar{b}\bar{a}$ and no matchings. For any number n, using n copies of ab and n copies of $\bar{b}\bar{a}$, we can produce the MHE component shown in Fig. 3.1 for $n = 3$. This component could also be finished, by matching the remaining a shown on the left with the remaining $\bar{a}$ on the right, effectively creating a ring structure. (As always, in the figure, $\bar{a}$ and $\bar{b}$ are shown as A and B.) Different numbers n yield nonequivalent (non-isomorphic) MHE components, thus the number of potential MHE components is infinite.*

$$\begin{tikzcd}
a \arrow[r] & b \arrow[d, dash, dotted] & a \arrow[r] \arrow[d, dash, dotted] & b \arrow[d, dash, dotted] & a \arrow[r] \arrow[d, dash, dotted] & b \arrow[d, dash, dotted] & \\
& B & A \arrow[l] & B & A \arrow[l] & B & A \arrow[l]
\end{tikzcd}$$

FIGURE 3.1. *Illustration for Example 3.1.*

Mother Nature computes the result of hybridization by composing MHEs using the available material in the test tube. When, for a given complex C, there are actually infinitely many nonequivalent MHEs, we say that *hybridization does not terminate for C*, or shorter, that *C is nonterminating*; otherwise, we say that *hybridization terminates*, or shorter, that *C is terminating.*

In practice, when we have termination of hybridization, a test tube prepared with sufficient quantities of each component of the complex holds, in principle, sufficient material to produce all molecular species that can be the result of hybridization. If sufficient quantities are present, adding even more material will not yield new results. Of course, in practice, a test tube is always finite and the hybridization reaction will, under normal conditions, always "terminate" (reach equilibrium). But the point is that, when hybridization does not terminate for a complex, adding ever more material can, in principle, result in ever more new molecular species (MHE components) to be produced. In this sense, the potential result of the hybridization is indeed infinite.

Let C be a sticker complex. If C has terminating hybridization, `hybridize`(C) is defined as the disjoint union of all finished MHE components. Otherwise, the hybridization of C, i.e., `hybridize`(C), is undefined.

Ligate. The ligate operator concatenates strands that are held together by a sticker. Formally, define a *gap* as a set of four nodes $\{n_1, n_2, n_3, n_4\}$ such that $\{n_1, n_4\} \in \mu$; $\{n_2, n_3\} \in \mu$; n_1 and n_2 (in that order) are consecutive nodes on a negative strand; n_3 is the last node on its (positive) strand; and n_4 is the first node on its (positive) strand. By *filling a gap* we mean modifying the complex so that the (n_3, n_4) is added to L. We now define `ligate`(C) as the complex obtained from C by filling all gaps.

Flush. Quite simply `flush`(C) equals the complex obtained from C by removing all components that do not contain an immobilized node.

Split. Consider a node n in some complex C. By *splitting before* (resp. *after*) n, we mean the following.

– If n has a predecessor (resp. successor) in its strand, denote it by m.
– Remove (m, n) (resp. (n, m)) from L.
– Furthermore, if there exists a node n', such that $\{n, n'\} \in \mu$, then n' is split in an analogous manner.

Now, consider the set of triples shown in Table 3.1. Each triple is called a *splitpoint* and has the form $(\mathit{label}, \mathit{free}, \mathit{place})$. By splitting C at such a splitpoint, we mean splitting C at all nodes labeled *label* (be it before or after, based on the value of *place*), on condition that the node is free (or closed, depending on the boolean value *free*). The result is denoted by `split`(C, label).

Block. Here we assume that C is saturated; if C is not saturated then the block operation on C is considered to be undefined. The operation `block`(C, σ), for any $\sigma \in \Omega \cup \Theta$, equals the complex obtained from C by adding all free nodes labeled σ to β.

Block-From. Here we again assume that C is saturated, otherwise the block-from operation is considered to be undefined.

TABLE 3.1
The allowed split points.

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

Let again $\sigma \in \Sigma$, and consider any contiguous substrand s in C. We call s a *σ-blocking range* if it satisfies two conditions. Firstly, all nodes of the substrand are free. Secondly, the last node of the substrand is labeled with σ. Now we define `blockfrom`(C, σ) to be the complex obtained from C by adding to β all nodes appearing in some σ-blocking range.

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

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

Then `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 `blockexcept`(C, n) is undefined.

Cleanup. The cleanup operator undoes matchings and blockings and removes all strands except for the longest positive strands. This operation is always defined.

3.1. Termination of Hybridization.

A sticker complex with non-terminating hybridization yields an infinite sticker complex. This is undesirable, as a sticker complex is conceived as an abstraction of DNA in test tubes. Clearly, a infinite sticker complex is no abstraction of any test tube. A natural question thus arises: can we efficiently decide, based solely on the sticker complex itself, whether hybridization is terminating? Fortunately, in previous work we have shown it is possible [7]. Next, we repeat the concepts and theorem relevant to the type system.

A *partitioned graph* in general is a triple (V, π, E) where (V, E) is an undirected graph and π is a partition of the node set V. Recall that an undirected graph (V, E) consists of a set V of nodes and a set $E \subseteq \{\{v, w\} \mid v, w \in V \text{ and } v \neq w\}$ of unordered pairs of nodes (undirected edges). Recall that a partition of a set V is a set of nonempty, pairwise disjoint subsets of V, called *blocks*, such that their union equals V.

Now given a complex C, the *hybridization graph for C* is the partitioned graph $H = (V, \pi, E)$ defined as follows:

– V equals the set of nodes of C;
– π contains, for each component D of C, the set of nodes belonging to D as a block;
– Let $F \subseteq V$ be the set of free nodes of C. Then E equals $\{\{v, w\} \mid v, w \in F$ and $\lambda(w) = \overline{\lambda(v)}$ and v and w do not belong to different immobilized components$\}$.

Thus, whereas the matching μ in C represents the pairs of nodes that are *already* annealed, the set E contains the pairs of nodes that *may* still be annealed (typically, in an MHE of C). Note that a complex is saturated if its hybridization graph does not contain any edges.

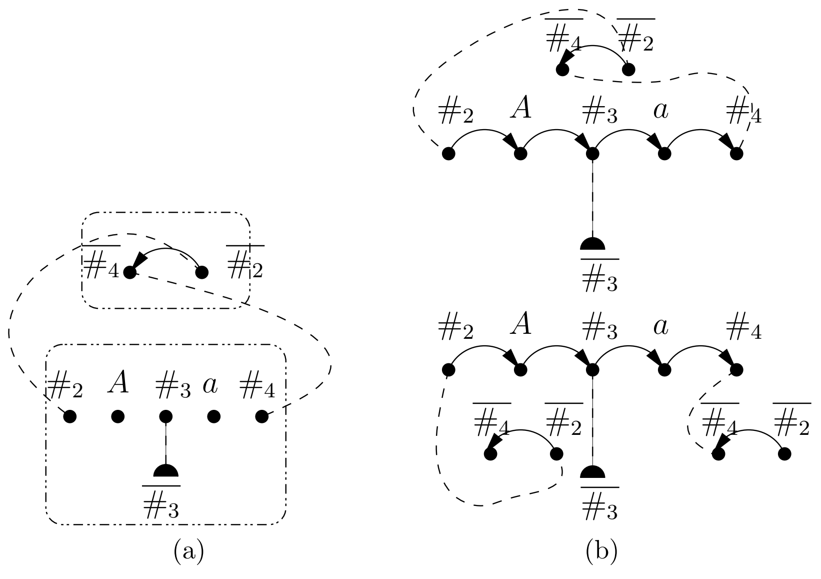


FIGURE 3.2. *Hybridization graph of a sticker complex with one immobilized node.*

The notion of alternating cycle can be defined in general in any partitioned graph $G = (V, \pi, E)$. A *path* in G is a sequence of nodes $v_1, \ldots, v_n$ such that for each i with $1 \leq i < n$, we have either an
edge move: $\{v_i, v_{i+1}\} \in E$, or a
block move: $v_i \neq v_{i+1}$ and they belong to a common block.
The path is said to be *alternating* if edge moves happen for each odd i, and block moves happen for each even i (always for $1 \leq i < n$). When the path is alternating, it is said to be an *alternating cycle* when n is odd and at least 3, and $v_n = v_1$.

In previous work [6], we have shown that a complex C has non-terminating hybridization if and only if there is an alternating cycle in its hybridization graph. This result disregarded immobilized components. The theorem is easily extended:

THEOREM 3.2. *A complex C has non-terminating hybridization if and only if there is an alternating cycle A in the hybridization graph of C, such that A does not pass through a block associated with an immobilized component.*

EXAMPLE 3.2. *Figure 3.2 (immobilized nodes are decorated with the symbol* ⏶*) (a) shows the hybridization graph of a sticker complex with two components. The bigger component has an immobilized node (the one labeled $\overline{\#_3}$). Consequently, the component, to which the node belongs, is immobilized. As each node has a unique label, we use the node labels to point out an alternating cycle: $\#_4, \overline{\#_4}, \overline{\#_2}, \#_2, \#_4$. Despite the cycles in the hybridization graph, this complex has terminating hybridization, because all cycles run through the bigger, immobilized component. Two copies of an immobilized component cannot be bonded together, as the resulting component would have two immobilized nodes.*

Figure 3.2 (b) shows the two components forming the hybridization based on the hybridization graph in (a). In the first case, the positive strand is folded into a circle. In the second case, two stickers are hybridized on both sides of the positive strand.

4. Implementation in DNA. In this section, we argue that the abstract sticker complexes and the operations on them presented above can be implemented by real

DNA complexes. The discussion remains theoretical as we have not performed laboratory experiments. On the one hand, the main purpose is to make the abstract model plausible as a theoretical framework to explore the possibilities and limitations of DNA computing as a database model; on the other hand, we use only rather standard biotechnological techniques.

Each component of an abstract complex is represented by a large surplus of duplicate copies in DNA. Each positive alphabet symbol from Σ is implemented by a strand of (single-stranded) DNA, such that the resulting set of DNA strands forms a set of DNA codewords [17, 24, 25]. If the DNA strand for symbol $a \in \Sigma$ is w, then the DNA strand for the complementary symbol $\bar{a}$, is, naturally, the Watson-Crick complementary strand to w. Then, matching of nodes by μ in an abstract complex is implemented by base pairing in the DNA complex. We will see below how blocking is implemented. Immobilization is implemented as is standard in DNA computing by attachment to surfaces [16] or magnetic beads.

The union operation amounts to mixing two test tubes together.

The difference $C_1 - C_2$ of complexes can be implemented by a subtractive hybridization technique [11]. Let C_1 (C_2) be stored in test tube t_1 (t_2). Because all strands in t_2 end in $\#_4$, we can easily append $\#_5$ to them. Next we add to t_2 an abundance of immobilized short primers $\overline{\#_5}$. Using polymerase we obtain complements to all strands in t_2, still immobilized, so that it is now easy to separate them. It remains to use these complements to remove all strands from t_1 that occurred in t_2. Since all strands have the same length, partial hybridization, leading to false removals, can be avoided by using a very precise melting temperature based on the precise length of the strands.

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

Splitting is achieved as usual by restriction enzymes. A feature of the abstract model is that we require only five recognition sites (Table 3.1). Of course, these recognition sites will have to be integrated in the DNA codeword design.

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

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

The cleanup operation starts by denaturing (warming up) the tube. Immobilized strands are removed from the tube. Next, a gel electrophoresis is carried out to separate the longest DNA molecules from the other molecules. Finally, the positive

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

FIGURE 5.1. *Syntax of DNAQL.*

strands are separated from the negative strands (for example, in the case that a positive strand is complete blocked in a sticker complex), by attaching all the negative alphabet symbols to a surface, thus immobilizing positive strands.

In connection with gel electrophoresis, a complication may arise when shorter circular strands may travel at approximately the same speed as longer linear strands. In the main application of DNAQL, namely the simulation of the relational algebra, presented in Section 9, this will not be an issue. Furthermore, in this paper we introduce a static type system which can be used to predict which species of strands can potentially occur in the test tube. Then for each species a separate gel experiment can be run to predict the different positions of the bands corresponding to the different species. In this way, the complication with circular strands may in many cases be avoided.

5. DNAQL. DNAQL [14] 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 5.1. 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.

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 $\bar{a}$. If $\bar{a} \in \bar{\Lambda}$ we call such a node a *probe*. The expression $\texttt{empty}$ stands for the empty complex.

The semantics of a DNAQL expression e is defined relative to a context consisting of a dimension ℓ, an *ℓ-complex assignment* ν, and an *ℓ-counter assignment* γ. An ℓ-complex assignment is a mapping from complex variables to ℓ-complexes; an ℓ-counter

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

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

$$\frac{[\![e']\!](\nu,\gamma) = C' \qquad C' \text{ has terminating hybridization}}{[\![\texttt{hybridize}(e')]\!](\nu,\gamma) = \texttt{hybridize}(C')}$$

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

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

FIGURE 5.2. *DNAQL Semantics: Part 1*

assignment is a mapping from counters to $\{1,\dots,\ell\}$. Naturally, ν must be defined on all free variables of e, and γ must be defined on all free counters of e. Within such a context, the expression can evaluate to an ℓ-complex, denoted by $[\![e]\!]^\ell(\nu,\gamma)$.

The semantic rules that define this evaluation are shown in Figures 5.2 and 5.3. 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 mapping f updated so that x is mapped to u. Because the operations on complexes are not always defined, the evaluation may fail, so $[\![e]\!]^\ell(\nu,\gamma)$ may be undefined. When e is a program, we denote $[\![e]\!]^\ell(\nu,\emptyset)$ simply by $[\![e]\!]^\ell(\nu)$.

EXAMPLE 5.1. *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 $y_1$ := cleanup(flush(hybridize($x_1 \cup$ immob($\bar{a}$)))) in
let $y_2$ := cleanup(flush(hybridize($x_2 \cup$ immob($\bar{b}$)))) in
if empty($y_1$) then empty else
if empty($y_2$) then empty else
cleanup(ligate(hybridize($y_1 \cup y_2 \cup \overline{\#_5\#_1}$)))
```

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.

6. 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

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

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

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

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

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

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

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

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

FIGURE 5.3. *DNAQL Semantics: Part 2*

and tags explicit, but the dimension and actual values of data entries hidden. In order to obtain a powerful type-checking algorithm 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 all the complexes of a strong type are 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.

6.1. Definition. We begin by introducing four symbols assumed not present in $\Sigma \cup \overline{\Sigma}$:

1. $*$ (*free*) represents an ℓ-core with none of the nodes blocked;
2. $\underline{*}$ (*blocked*) represents an ℓ-core with all nodes blocked; and
3. $\hat{*}$ (*open*) represents an ℓ-core with all nodes except one blocked.

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 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. We extend the complementarity relation for sticker complex types, by defining $\overline{*} = ?$, $\overline{\hat{*}} = ?$ and $\overline{?}$ is undefined, i.e., the immobilized

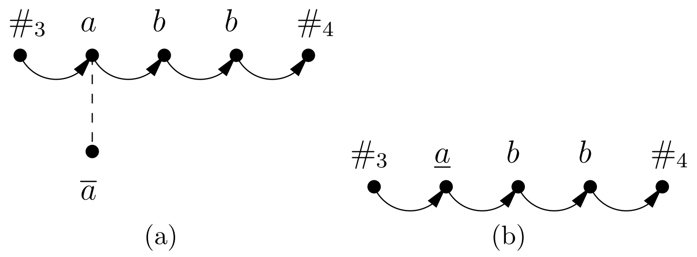


FIGURE 6.1. *Two ill-typed complexes.*

negative atomic value symbol (?) can match with a free or an open ℓ-core. Note that $\underline{*}$ has no complementary symbol, and that the complementarity relation is no longer a bijection.

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 (recall the definition of redundancy from Section 2).

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 belongs to β, and at most one node of r is involved in μ;
- r is of type $\underline{*}$ if all nodes of r belong to β;
- r is of type $\hat{*}$ if all nodes of r but one belong to β.

Now we say that C is *well typed* if

- every ℓ-core in C is of type $*$, $\underline{*}$ or $\hat{*}$;
- negative atomic value symbols can only occur on immobilized nodes (i.e., probes); and
- every immobilized node is labeled with a negative symbol.

EXAMPLE 6.1. *Figure 6.1 shows two ill-typed complexes. The first complex is ill typed because it contains a negative atomic value symbol ($\overline{a}$) that is not immobilized. The second complex is ill typed because the node labeled a in a* 3*-core is blocked (shown by underlining the symbol a). This* 3*-core is thus not of type $*$, as one node is blocked, and it is not of type $\hat{*}$ or $\underline{*}$ as two nodes are not blocked.*

Moreover, if C is well typed, we define *stype*(C) as the sticker complex type obtained by:

- contracting every ℓ-core occurring in C to a single node labeled by the type of the ℓ-core ($*$, $\underline{*}$ or $\hat{*}$);
- replacing the label of a node labeled with an immobilized negative atomic value by ?;
- when a node from an ℓ-core r in C is matched by μ to a node u, then in *stype*(C) the single node representing r is matched to u. Note that, by the previous item, in *stype*(C) node u has label ?. Furthermore, the node repre-

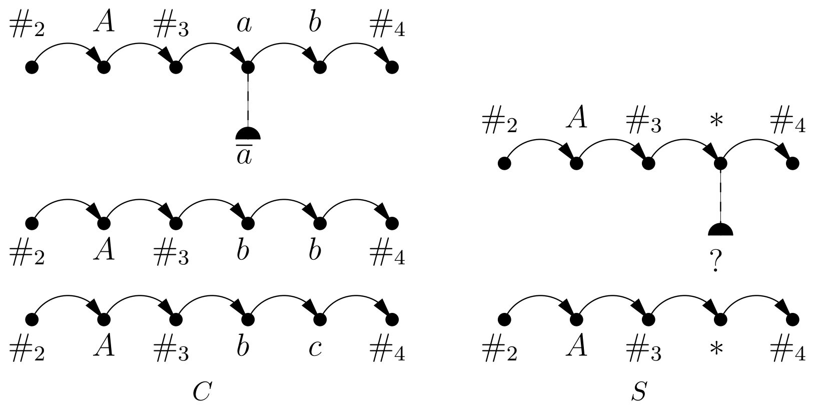


FIGURE 6.2. *A sticker complex C and a sticker complex type S such that C has type S.*

senting r is labeled $*$ or $\hat{*}$.

Note that the subsumption relation among sticker complexes, defined in Section 2, can be adopted naturally to sticker complex types. We have the following lemma.

LEMMA 6.1. *Let C_1 and C_2 be well-typed sticker complexes. If $C_1 \sqsubseteq C_2$, then stype$(C_1) \sqsubseteq$ stype(C_2). If stype$(C_1) \sqsubseteq$ stype(C_2) and stype(C_1) does not contain nodes labeled by symbols of $N = \{*, \hat{*}, \underline{*}, ?\}$, then $C_1 \sqsubseteq C_2$.*

Proof. If $C_1 \sqsubseteq C_2$, then for each component c of C_1 there is a component c' of C_2 isomorphic to c. Hence *stype*(c) is isomorphic to *stype*(c'). If *stype*$(C_1) \sqsubseteq$ *stype*(C_2) and *stype*(C_1) does not contain nodes labeled by symbols of N, then $C_1 \equiv$ *stype*$(C_1) \sqsubseteq$ *stype*(C_2). Let $c \in$ *comp*(C_1), then there is a $c' \in$ *comp*$($*stype*$(C_2))$ such that $c' \equiv c$. Hence c' does not contain nodes labeled by symbols of N. Thus a component isomorphic to c' belongs to C_2. □

We will often use these properties without mention. For a well-typed sticker complex C and a sticker complex type S, we now say that C *has 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.

EXAMPLE 6.2. *Figure 6.2 shows a sticker complex C of dimension 2, and a sticker complex type S. Structurally, C and S are very alike. There are two differences: (i) 2-cores are contracted to one node labeled $*$, and (ii) as the second and third strand of C only differ in their respective 2-cores, only one strand (the bottom strand of S) is needed to represent both. Sticker complex type S is stype(C) and is thus the smallest type for C.*

A sticker complex type is "weak", in the sense that any well-typed sticker complex having as type 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 involving hybridization, where we need to know about components that are sure to be present. We now introduce the notion of a "strong" sticker complex type which can place further restrictions on sticker complexes. A *strong sticker complex type* τ is a triple $(S, \odot, \mathfrak{h})$, where S is a sticker complex type, $\odot$ is a sticker complex type subsumed by S, $\mathfrak{h}$ is a boolean, and moreover if $\mathfrak{h} =$ *true*, then $C \cup \odot$ is saturated for all $C \in$ *comp*(S). Sticker complex type S is called the *weak type* of τ, $\odot$ is called the *mandatory type* of τ, and $\mathfrak{h}$ is called the $\mathfrak{h}$-*bit* of τ.

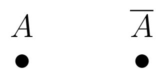
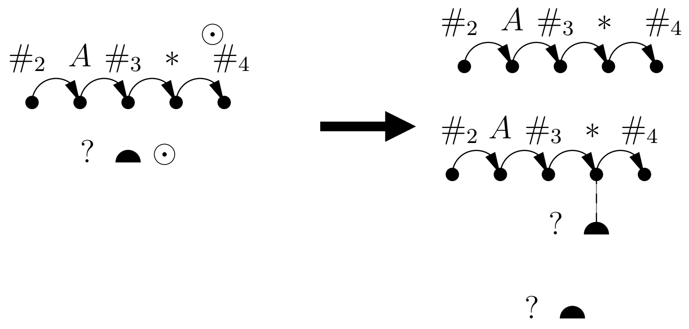


FIGURE 6.3. *A sticker complex type with two single-node components.*



$\odot$
$\#_2$ A $\#_3$ $*$ $\#_4$
? $\odot$
$\#_2$ A $\#_3$ $*$ $\#_4$
$\#_2$ A $\#_3$ $*$ $\#_4$
?
?

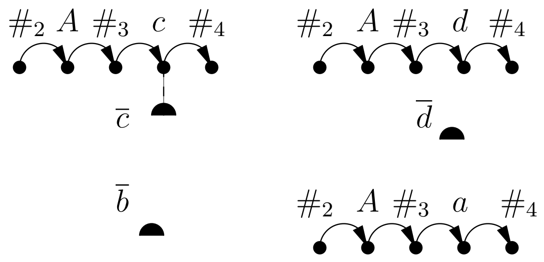
FIGURE 6.4. *A type with two mandatory components on the left. On the right is the hybridization of the type on the left. Despite the fact that all components start as mandatory, the hybridization contains only non-mandatory components.*

EXAMPLE 6.3. *Consider the sticker complex type τ with the weak type shown in Figure 6.3. Assume that the component on the left is the only mandatory component and that $\mathfrak{h} = true$. Then the component on the right is "garbage", in the sense that any complex having type τ cannot contain a node labeled with $\overline{a}$, because such a complex will not be saturated. Indeed, because the component on the left is mandatory, each complex having type τ must contain a node labeled A. This is the raison d'être for the condition that for all $C \in comp(S)$, $c \cup \odot$ has to be saturated if $\mathfrak{h} = true$.*

For a well-typed sticker complex C and a strong sticker complex type $\tau = (S, \odot, \mathfrak{h})$, we now say that C has type τ, denoted $C : \tau$, if $\odot$ is subsumed by *stype*(C), *stype*(C) is subsumed by S (i.e., C has type S), and C is saturated if $\mathfrak{h} = true$. A strong sticker complex type τ is called *saturated* if all complexes having type τ are saturated. From now on, we will refer to sticker complex types as *weak types* and to strong sticker complex types as *types*. Let $\tau = (S, \odot, \mathfrak{h})$ be a type. With $[\![\tau]\!]$ we denote the set of complexes (of any dimension) having type τ.

EXAMPLE 6.4. *The $\mathfrak{h}$-bit in types is essential for typing the block operations, i.e.,* `block`, `blockfrom`, *and* `blockexcept`. *As will become clear in proofs about types, the $\mathfrak{h}$-bit introduces some subtle modeling options. For example, recall the weak type S shown in Figure 6.3. Suppose a type τ, with weak type S, $\odot =$* `empty` *and $\mathfrak{h} = true$. There are three complexes having type τ: the empty complex, the complex consisting of the component on the left and the complex consisting of the component on the right. The complex consisting of both components is not saturated and thus prohibited by the $\mathfrak{h}$-bit.*

EXAMPLE 6.5. *Consider the complex in Figure 6.4, on the left. Although both components are mandatory (indicated by the $\odot$), we will see that the hybridization of this type consists of three non-mandatory components (Figure 6.4 on the right). Let us call this type $\tau = (S, \odot, \mathfrak{h})$. The $\mathfrak{h}$-bit of the resulting type is true. This has important repercussions on the set of complexes having this type. Indeed, consider the complex in Figure 6.5. This complex does* not *have type τ, but it has type $\tau' = (S, \odot, false)$.*



$\#_2$ A $\#_3$ c $\#_4$
$\#_2$ A $\#_3$ d $\#_4$
$\overline{c}$
$\overline{d}$
$\overline{b}$
$\#_2$ A $\#_3$ a $\#_4$

FIGURE 6.5. *A complex with five components. This complex does not have the type on the right of Figure 6.4.*

The definition of a saturated type is semantic. Can we decide, based on the syntax of a type, whether the type is saturated?

LEMMA 6.2. *Type $\tau = (S, \odot, \mathfrak{h})$ is saturated if and only if S is saturated or $\mathfrak{h} = true$.*

Proof. First, we prove the only-if-direction. Suppose τ is not saturated. Then there is a sticker complex C such that $C \in [\![\tau]\!]$ and C is not saturated. Clearly, $\mathfrak{h}$ cannot be *true*, because an unsaturated complex has type τ. Secondly, by $C \in [\![\tau]\!]$ we know that *stype*$(C) \sqsubseteq S$. If a complex subsumed in S is not saturated, S itself cannot be saturated.

Secondly, we prove the if-direction. Suppose that S is not saturated and $\mathfrak{h} = false$. We show that there is a complex C having type τ that is not saturated. Because S is not saturated, there are (at least) two nodes u and v such that adding $\underline{\{u, v\}}$ to μ still results in a valid weak type. Nodes u and v are thus free, $\lambda(u) = \overline{\lambda(v)}$ and the nodes do not connect two (different) immobilized components (note that u and v may be part of the same (immobilized) component). Let C_u resp. C_v be the component of node u resp. v. We define *stype*(C) as the union of $\odot$, C_u and C_v. We split the construction into two cases (without loss of generality we assume that node u has the positive label):

1. $\lambda(u) \notin N$: node u is not labeled with $*$, $\underline{*}$, or $\hat{*}$. Replacing the ℓ-cores in *stype*(C) by any sequence of atomic value symbols, and replacing all the probes by an arbitrary negative atomic value symbols, results in a complex C that is not saturated.
2. $\lambda(u) \in N$: node u is labeled either with $*$ or with $\hat{*}$. Consequently, v is labeled with ?. Fix an atomic value symbol, say $a \in \Lambda$. We let complex C be the complex in which all ℓ-cores are replaced by a^ℓ and all probes, i.e., ?, are replaced by $\overline{a}$.

□

Note that, as a consequence of Lemma 6.2, saturatedness of a type is decidable in polynomial time.

6.2. Subtypes. A desirable property of types is that they are *inhabited*, i.e., for every type τ, the set $[\![\tau]\!]$ is non-empty. Indeed, any complex C with *stype*$(C) \equiv \odot$ belongs to $[\![\tau]\!]$. Indeed, if the $\mathfrak{h}$-bit is set to *false*, any complex D with $\odot \sqsubseteq$ *stype*$(D) \sqsubseteq S$ is of type τ, and thus in particular complex C. On the other hand, if the $\mathfrak{h}$-bit is set to *true*, by the definition of a strong sticker complex type the weak type $\odot$ is saturated, consequently, C is saturated.

Let τ and τ' be two types. We denote $[\![\tau]\!] \subseteq [\![\tau']\!]$ by $\tau \preceq \tau'$. A type τ is subsumed in, or equivalently is a *subtype* of, another type τ' if all complexes having type τ also have type τ'. Two types τ and τ' are called *equivalent* if $\tau \preceq \tau'$ and $\tau' \preceq \tau$.

EXAMPLE 6.6. *Recall the type $\tau = (S, \odot, \mathit{true})$ on the right of Figure 6.4. Let type $\tau' = (S, \odot, \mathit{false})$. Notwithstanding the fact that both τ and τ' have the same weak type and the same set of mandatory components, we have that $\tau \preceq \tau'$ but not $\tau' \preceq \tau$, because the complex shown in Figure 6.5 has type τ' but does not have type τ.*

The notion of subtyping is defined semantically. However, a type can have an infinite number of complexes. An efficiently decidable syntactic characterization of subtyping is thus called for. Proposition 6.3 provides such a characterization.

PROPOSITION 6.3. *Let $\tau = (S, \odot, \mathfrak{h})$ and $\tau' = (S', \odot', \mathfrak{h}')$ be types. Type τ is a subtype of τ' if and only if (i) $S \sqsubseteq S'$; (ii) $\odot' \sqsubseteq \odot$; and (iii) if $\mathfrak{h}' = \mathit{true}$ then τ is saturated.*

Proof. First, we prove the $\Rightarrow$-direction. We know that $\tau \preceq \tau'$, and we assume that one of the three conditions is false, to arrive at a contradiction.

(i) Suppose that $S \sqsupset S'$ holds. Let D be a component in $\mathit{comp}(S) \setminus \mathit{comp}(S')$. Let C be a complex with $\mathit{stype}(C) = \odot \cup D$. Complex C has type τ, even if $\mathfrak{h} = \mathit{true}$. But complex C clearly does not have type τ', because D is not a component of S.

(ii) Suppose that $\odot' \sqsupset \odot$ holds. Let C be a complex with $\mathit{stype}(C) = \odot$. By definition, C has type τ. Complex C does not have type τ', because $\mathit{stype}(C) = \odot \sqsubset \odot'$.

(iii) Suppose that $\mathfrak{h}' = \mathit{true}$ and τ is *not* saturated. If type τ is not saturated, then $\mathfrak{h} = \mathit{false}$ and S is not saturated. Let complex C be a complex with $\mathit{stype}(C) = S$, in which all ℓ-cores are replaced by a sequence of a labeled nodes, with $a \in \Sigma$, and all probes are labeled with $\overline{a}$. Complex C has type τ and is not saturated. Consequently, C does not have type τ'.

The $\Leftarrow$-direction is easier to prove. Let C be a complex having type τ, we show that C also has type τ': (i) $\mathit{stype}(C) \sqsubseteq S \sqsubseteq S'$; (ii) $\odot' \sqsubseteq \odot \sqsubseteq \mathit{stype}(C)$; and (iii) if $\mathfrak{h}' = \mathit{true}$, then τ is saturated and thus C is saturated. ◻

Lemma 6.2 implies that the notion of saturated for types is decidable in polynomial time, and therefore that the notion of subtype is decidable in polynomial time.

We have the following corollary to Proposition 6.3.

COROLLARY 6.4. *Let $\tau = (S, \odot, \mathfrak{h})$ and $\tau' = (S', \odot', \mathfrak{h}')$ be types. Types τ and τ' are equivalent if and only if (i) $S \equiv S'$; (ii) $\odot \equiv \odot'$; and (iii) if $S \equiv S'$ is not saturated, then $\mathfrak{h} = \mathfrak{h}'$.*

Proof. Recall that $S \sqsubseteq S'$ and $S' \sqsubseteq S$ iff $S \equiv S'$ (and similarly for $\odot$ and $\odot'$). Hence τ and τ' are equivalent iff (i) $S \equiv S'$; (ii) $\odot \equiv \odot'$; (iii) if $\mathfrak{h}' = \mathit{true}$ then τ is saturated; and (iv) if $\mathfrak{h} = \mathit{true}$ then τ' is saturated.

By Lemma 6.2, $\tau = (S, \odot, \mathfrak{h})$ is saturated iff S is saturated or $\mathfrak{h} = \mathit{true}$ (and similarly for τ'). Hence, if $S \equiv S'$ is saturated, then conditions (iii) and (iv) hold trivially. If $S \equiv S'$ is not saturated, then condition (iii) says if $\mathfrak{h}' = \mathit{true}$, then $\mathfrak{h} = \mathit{true}$, and condition (iv) says if $\mathfrak{h} = \mathit{true}$, then $\mathfrak{h}' = \mathit{true}$. Consequently, $\mathfrak{h} = \mathfrak{h}'$ in this case. ◻

Obviously, type $\tau = (S, \odot, \mathit{true})$ is a subtype of the type $\tau' = (S, \odot, \mathit{false})$. On the other hand, by Corollary 6.4, $\tau' = (S, \odot, \mathit{false})$ may also be a subtype of τ when S is saturated.

EXAMPLE 6.7. *Figure 6.6 shows types $\tau = (S, \odot, \mathit{false})$ and $\tau' = (S', \odot', \mathit{true})$ with $\tau \preceq \tau'$. Since S is saturated, setting $\mathfrak{h} = \mathit{true}$ in τ yields a type equivalent to τ.*

The next lemma specifies the "tightest" type (up to equivalence) for a given

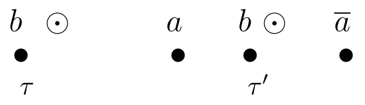


FIGURE 6.6. *Types $\tau = (S, \odot, false)$ and $\tau' = (S', \odot', true)$ with $\tau \preceq \tau'$.*

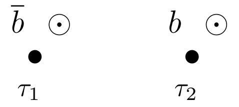


FIGURE 6.7. *Types $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$, having no mandatory components in common. As a result, $\tau_1 \vee \tau_2 = (S_1 \cup S_2, \texttt{empty}, true)$ allows the empty complex, whereas the empty complex is not part of $[\![\tau_1]\!]$ or $[\![\tau_2]\!]$.*

complex.

LEMMA 6.5. *Let C be a complex and τ a type. Then $C : \tau$ iff $(stype(C), stype(C), \mathfrak{h}_C) \preceq \tau$ with $\mathfrak{h}_C = true$ iff C is saturated.*

Proof. Let $\tau = (S, \odot, \mathfrak{h})$. By Proposition 6.3, $(stype(C), stype(C), \mathfrak{h}_C) \preceq \tau$ iff (1) $\odot \sqsubseteq stype(C) \sqsubseteq S$ and (2) if $\mathfrak{h} = true$, then $(stype(C), stype(C), \mathfrak{h}_C)$ is saturated. Now, by Lemma 6.2, $(stype(C), stype(C), \mathfrak{h}_C)$ is saturated iff $stype(C)$ is saturated or C is saturated. If $stype(C)$ is saturated, then C is saturated. Hence, $(stype(C), stype(C), \mathfrak{h}_C)$ is saturated iff C is saturated. By definition, $C : \tau$ iff (1) $\odot \sqsubseteq stype(C) \sqsubseteq S$ and (2) if $\mathfrak{h} = true$, then C is saturated — so the lemma follows. □

6.3. Least upper bound. Let τ_1 and τ_2 be types. A type is called an *upper bound* of τ_1 and τ_2 if $\tau_1 \preceq \tau$ and $\tau_2 \preceq \tau$. A type τ is called the *least upper bound* of τ_1 and τ_2 if τ is an upper bound of τ_1 and τ_2 and for all upper bounds τ' of τ_1 and τ_2, $\tau \preceq \tau'$. Note that if τ and τ' are least upper bounds of τ_1 and τ_2, then τ and τ' are equivalent. We denote the (up to equivalence unique) least upper bound of τ_1 and τ_2 (if it exists) by $\tau_1 \vee \tau_2$.

Let S_1 and S_2 be two weak types. The intersection of S_1 and S_2 is the weak type formed by the components of S_1 having an isomorphic companion in the set of components of S_2. We denote the intersection of S_1 and S_2 by $S_1 \cap S_2$.

PROPOSITION 6.6. *Let $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ be types. The least upper bound of τ_1 and τ_2 exists and is equivalent to $(S_1 \cup S_2, \odot_1 \cap \odot_2, \tau_1$ saturated $\wedge$ τ_2 saturated$)$.*

Proof. First, note that $(S_1 \cup S_2, \odot_1 \cap \odot_2, \tau_1$ saturated $\wedge\, \tau_2$ saturated$)$ is a type. Indeed, as τ_1 and τ_2 are types, $S_1 \cup S_2$ and $\odot_1 \cap \odot_2$ are weak types, and $\odot_1 \cap \odot_2 \sqsubseteq \odot_1 \sqsubseteq S_1 \sqsubseteq S_1 \cup S_2$. Denote the weak type $\odot_1 \cap \odot_2$ with $\odot$. If $\mathfrak{h} = true$, we must show that for all $C \in comp(S)$ it holds that $\odot \cup C$ is saturated. The fact that $\mathfrak{h} = true$ indicates that both τ_1 and τ_2 are saturated. Type τ_i, for $i \in \{1, 2\}$ is saturated iff S_i is saturated or $\mathfrak{h}_i = true$. If $C \in S_i$, then $\odot_i \cup C$ is saturated, because $\odot_1 \cap \odot_2 \sqsubseteq \odot_i$, $\odot \cup C$ is saturated.

Now we must show that $\tau = (S_1 \cup S_2, \odot_1 \cap \odot_2, \tau_1$ saturated $\wedge\, \tau_2$ saturated$)$ is the *least* upper bound. Let $\tau' = (S', \odot', \mathfrak{h}')$ be a type. Type τ' is an upper bound of τ_1 and τ_2 if $\tau_i \preceq \tau'$ for $i \in \{1, 2\}$. By Proposition 6.3, $\tau_i \preceq \tau'$ iff $S_i \sqsubseteq S'$, $\odot' \sqsubseteq \odot_i$ and if $\mathfrak{h}' = true$, then τ_i is saturated. Hence, τ' is an upperbound iff $S_1 \cup S_2 \sqsubseteq S'$, $\odot' \sqsubseteq \odot_1 \cap \odot_2$, and if $\mathfrak{h} = true$, then τ_1 is saturated and τ_2 is saturated. Hence, by Proposition 6.3, τ' is an upper bound of τ_1 and τ_2 iff $\tau \preceq \tau'$. □

In some cases, we will have $[\![\tau_1]\!] \cup [\![\tau_2]\!] \equiv [\![\tau_1 \vee \tau_2]\!]$, however, not in all cases will this be true. Indeed, consider the two types τ_1 and τ_2 shown in Figure 6.7. The empty complex is in $[\![\tau_1 \vee \tau_2]\!]$, whereas the empty complex is not in $[\![\tau_1]\!]$ or $[\![\tau_2]\!]$, because these types have a non-empty mandatory type.

6.4. Greatest lower bound. Let τ_1 and τ_2 be types. A type τ is called a *lower bound* of τ_1 and τ_2 if $\tau \preceq \tau_i$ for all $i \in \{1,2\}$. A type τ is called a *greatest lower bound* of τ_1 and τ_2 if τ is a lower bound of τ_1 and τ_2, and for all lower bounds τ' of τ_1 and τ_2, $\tau' \preceq \tau$. Notice that if τ and τ' are greatest lower bounds, then τ and τ' are equivalent. The (up to equivalence unique) greatest lower bound of τ_1 and τ_2 (if it exists) is denoted by $\tau_1 \wedge \tau_2$.

Proposition 6.7. *Let $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ be types. Then a lower bound of τ_1 and τ_2 exists iff both $\odot_1 \cup \odot_2 \sqsubseteq S_1 \cap S_2$ and if either τ_1 or τ_2 is saturated, then $\odot_1 \cup \odot_2$ is saturated. If a lower bound of τ_1 and τ_2 exists, then there is a greatest lower bound τ, and τ is equivalent to $\tau_g = (S_1 \cap S_2 - Z, \odot_1 \cup \odot_2, \tau_1$ saturated $\vee$ τ_2 saturated$)$, where $Z = \{C \in comp(S_1 \cap S_2) \mid C \cup \odot_1 \cup \odot_2$ is saturated$\}$ if either τ_1 or τ_2 is saturated, and $Z = \emptyset$ otherwise.*

Proof. First note that τ_g is a type iff both $\odot_1 \cup \odot_2 \sqsubseteq S_1 \cap S_2$ and if either τ_1 or τ_2 is saturated, then $\odot_1 \cup \odot_2$ is saturated.

Let $\tau = (S, \odot, \mathfrak{h})$ be a type. Then, by Proposition 6.3, $\tau \preceq \tau_i$ iff $S \sqsubseteq S_i$, $\odot_i \sqsubseteq \odot$, and if $\mathfrak{h}_i = $ *true*, then τ is saturated. Hence, τ is a lower bound of τ_1 and τ_2 iff $S \sqsubseteq S_1 \cap S_2$, $(\odot_1 \cup \odot_2) \sqsubseteq \odot$, and if $\mathfrak{h}_1 = $ *true* or $\mathfrak{h}_2 = $ *true*, then τ is saturated. By Corollary 6.4, if τ_i is saturated and $\mathfrak{h}_i = $ *false*, then the type τ_i' obtained from τ_i by setting $\mathfrak{h}_i$ to *true* is equivalent to τ_i. Thus, τ is a lower bound of τ_1 and τ_2 iff $S \sqsubseteq S_1 \cap S_2$, $(\odot_1 \cup \odot_2) \sqsubseteq \odot$, and if τ_1 or τ_2 is saturated, then τ is saturated. Hence, τ is a lower bound of τ_1 and τ_2 iff both (1) $\odot_1 \cup \odot_2 \sqsubseteq S_1 \cap S_2$, (2) if either τ_1 or τ_2 is saturated, then $\odot_1 \cup \odot_2$ is saturated, and (3) $\tau \preceq \tau_g$, where the $\mathfrak{h}$-bit τ_1 *saturated* $\vee$ τ_2 *saturated* of τ_g follows from Proposition 6.3. □

Example 6.8. *Types τ_1 and τ_2 from Figure 6.7 do not have a greatest lower bound. Indeed, $S_1 \cap S_2$ is the empty complex, while the weak type $\odot_1 \cup \odot_2$ contains two components.*

7. Operations on Sticker Complex Types. We have defined a set of operations on complexes. The type system will mimic the structural changes, effected by the operations on complexes, on types. Therefore we define the set of operations, introduced in Section 3, on types.

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

Union. Let $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ be two types. We let $\tau_1 \cup \tau_2 = (S_1 \cup S_2, \odot_1 \cup \odot_2, \mathfrak{h})$, where $\mathfrak{h} = $ *true* iff both (1) S_1 and S_2 are mutually non-interacting, i.e., there are no vertices u in a component C_1 of S_1 and v in a component C_2 of S_2 such that (a) u and v are free and complementary labeled, and (b) C_1 and C_2 are not both immobilized, and (2) S_i is saturated or $\mathfrak{h}_i = $ *true* for all $i = 1, 2$.

Note that $\tau_1 \cup \tau_2$ is a type as for all $C \in comp(S_1)$, $C \cup \odot_1$ is saturated, and thus $C \cup \odot_1 \cup \odot_2$ is saturated by condition (1) when $\mathfrak{h} = $ *true* (the case $C \in comp(S_2)$ is analogous).

Difference. Let $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ be two types, with $S_i = (V_i, L_i, \lambda_i, \mu_i, \iota_i, \beta_i)$ for $i = 1, 2$, satisfying:

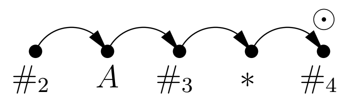


FIGURE 7.1. *A hybridized, strong type with a single mandatory component.*

TABLE 7.1
Two complexes having the type depicted in Figure 7.1.

C_1	C_3
$\#_2A\#_3a\#_4$	$\#_2A\#_3b\#_4$
$\#_2A\#_3b\#_4$	$\#_2A\#_3c\#_4$

1. $\mu_1 = \iota_1 = \beta_1 = \emptyset = \mu_2 = \iota_2 = \beta_2$ and there are no nodes labeled with $\underline{*}$ or $\hat{*}$, i.e., all components in S_1 and S_2 are single strands.
2. All strands of S_1 and S_2 are positive, noncircular. Furthermore, all strands have the same length and the same number of $*$-labeled nodes.
3. Each strand of S_2 ends with $\#_4$ and does not contain $\#_5$.

If these conditions are not satisfied, the operation is undefined.

Let T_1 be the set of all strands in S_1 that do not have an isomorphic copy in S_2:

$$T_1 = \{D \in \mathit{comp}(S_1) \mid \forall E \in \mathit{comp}(S_2), E \ncong D\}$$

Let T_2 be the set of all strands in S_1 that do not have an isomorphic copy in S_2 that is mandatory:

$$T_2 = \{D \in \mathit{comp}(S_1) \mid \forall E \in \odot_2, E \ncong D\}$$

We denote the set of components in a sticker complex type S with a $*$-labeled node by $\mathit{data}(S)$. The difference $\tau_1 - \tau_2$ equals $\big(\mathit{data}(S_1) \cup T_2, \odot_1 \cap T_1, \mathit{true}\big)$. Note that $\tau_1 - \tau_2$ is a type, because all components are positive, noncircular strands. Hence, every subset of $\mathit{data}(S_1) \cup T_2$ is saturated.

EXAMPLE 7.1. *Figure 7.1 shows a type τ with a single mandatory component. The $\mathfrak{h}$-bit is true. There are no matching, blockings nor immobilizations and the strand ends on a $\#_4$ and does not contain a $\#_5$. Consequently, the difference between τ and itself is defined. All complexes having type τ consist of linear strands, differing solely on the atomic value symbols. Let C_1 and C_2 be complexes of dimension 1 having type τ. The content of the complexes is listed in Table 7.1. On the type-level, the cases $C_1 - C_1$ and $C_1 - C_2$ are indistinguishable, however, the resulting complexes are definitely different. The output of $C_1 - C_1$ is the empty complex, whereas the output of $C_1 - C_2$ is the complex containing the strand $\#_2A\#_3a\#_4$. In other words, the data strands (strands with a node labeled $*$) are unpredictable on the type-level. Consequently, they are preserved in the output type, regardless of the content of the second type τ_2.*

Hybridize. The hybridize operator on sticker complexes can naturally be adapted to weak types by incorporating the extended complementarity relation, i.e., with $\overline{*} = ?$ and $\overline{\hat{*}} = ?$ as legal matchings. Denote this adjusted version by $\texttt{hybridize}_t$.

Let $\tau = (S, \odot, \mathfrak{h})$ be a type. If $\mathfrak{h} = \mathit{true}$, then the hybridization of τ, denoted $\texttt{hybridize}(\tau)$, equals τ.

Assume $\mathfrak{h} = \mathit{false}$. If hybridization does not terminate for S (i.e., $\texttt{hybridize}_t(S)$ is not defined), then hybridization of τ is not defined.

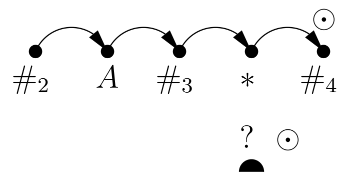


FIGURE 7.2. *A type τ with two mandatory components.*

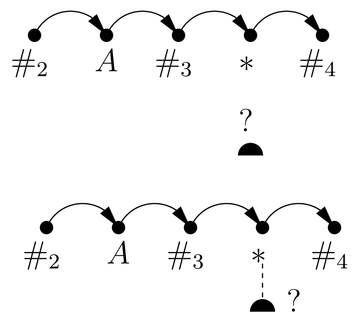


FIGURE 7.3. *Type* `hybridize`(τ)*, where τ is from Figure 7.2.*

We call a component D a *necessary* component of τ if $D \in comp(\odot)$ and D is not isomorphic to `immob`(?). Let NC be the set of necessary components of τ. The hybridization of τ, denoted `hybridize`(τ), equals $(Cs, \odot_h, true)$, where

$$Cs = \left(\bigcup_{NC \sqsubseteq X \sqsubseteq S} \texttt{hybridize}_t(X) \right) \cup \{\texttt{immob}(?) \mid \texttt{immob}(?) \sqsubseteq S\},$$

and $\odot_h$ consists of all components D of Cs such that either (1) D is a component of both $\texttt{hybridize}_t(NC)$ and $\texttt{hybridize}_t(S)$ or (2) $D = \texttt{immob}(?) \in \odot$ and there is no component in S with an free node labeled with $*$ or $\hat{*}$.

Note that `hybridize`(τ) is well defined as $D \cup \odot_h$ not saturated for some $D \in Cs$ would imply that some $D' \in \odot_h$ is unfinished with respect to Cs — a contradiction.

EXAMPLE 7.2. *Consider type τ displayed in Figure 7.2. Type $\tau' =$* `hybridize`(τ) *is shown in Figure 7.3 (except for the $\mathfrak{h}$-bit which is always true). Note that the weak type of τ' consists of three components, all of which are not mandatory.*

Ligate & Flush. The definition of `ligate` and `flush` on sticker complexes is naturally adapted to weak types. Let $\tau = (S, \odot, \mathfrak{h})$ be a type. Then the *ligation* of τ, denoted by `ligate`(τ), equals (`ligate`(S), `ligate`$(\odot)$, $\mathfrak{h}$). Similarly, `flush`(τ) equals (`flush`(S), `flush`$(\odot)$, $\mathfrak{h}$).

Split. The definition of `split` on sticker complexes is naturally adapted to weak types. Let $\tau = (S, \odot, \mathfrak{h})$ be a type. Let *label* be the label of a split point, recall Table 3.1. Then the split of τ, denoted `split`$(\tau, label)$, equals (`split`$(S, label)$, `split`$(\odot, label)$, $\mathfrak{h}$).

Block. The definition of the block operator on sticker complexes is naturally adapted to weak types. Let $\tau = (S, \odot, \mathfrak{h})$ be a type and let $\sigma \in (\Omega \cup \Theta)$ be a tag or an attribute symbol. For `block`(τ, σ) to be defined, it is required that τ is saturated, otherwise, the operation is undefined. We define `block`$(\tau, \sigma) =$ (`block`(S, σ), `block`$(\odot, \sigma)$, $true$).

Block-From. Except for a slightly altered definition of a σ-blocking range, the definition of the block-from operator on sticker complexes is naturally adapted to weak types, as we show next. Let $\tau = (S, \odot, \mathfrak{h})$ be a type and let $\sigma \in (\Omega \cup \Theta)$ be a symbol. Again, τ must be saturated. Otherwise, the operation is undefined.

Consider a substrand s of S. We call s a *σ-blocking range*, in the context of weak types, if it satisfies two conditions. Firstly, all nodes of the substrand are free and none of them is labeled with $\underline{*}$ or with $\hat{*}$. Secondly, the last node of the substrand is labeled with σ. We define for any weak type W with set β of blocked nodes, $\texttt{blockfrom}(W, \sigma)$ to be the weak type obtained from W by adding to β all nodes x appearing in some σ-blocking range, except if x is labeled $*$, in that case x is relabeled with $\underline{*}$.

Block-Except. Operation `blockexcept` is defined on a weak type S iff each of the following conditions hold:

1. every node labeled with $*$,$\hat{*}$, or $\underline{*}$ is preceded by a node labeled $\#_3$ and be followed by a node labeled $\#_4$;
2. every node labeled with $*$ is not matched, and the preceding node and following node (labeled by $\#_3$ and $\#_4$, resp.) are both free;
3. every node labeled with $\hat{*}$ is matched, and every node labeled $\hat{*}$, or $\underline{*}$ is is preceded and followed by a closed node labeled by $\#_3$ and $\#_4$;
4. S is saturated.

If these conditions are satisfied, then $\texttt{blockexcept}(S)$ is obtained from S by, looking for any triple of consecutive, unmatched nodes (n_1, n_2, n_3) on a strand where n_1 is labeled $\#_3$, n_2 is labeled $*$, and n_3 is labeled $\#_4$. For any such triple, we relabel n_2 to $\hat{*}$, and we add n_1 and n_3 to β.

Let $\tau = (S, \odot, \mathfrak{h})$ be a type. We define $\texttt{blockexcept}(\tau)$ by $(\texttt{blockexcept}(S), \texttt{blockexcept}(\odot), \mathit{true})$.

Note that `blockexcept` for types no longer requires a natural number n as parameter. Indeed, the dimension of sticker complexes is abstracted away in sticker complex types.

Cleanup. Let $\tau = (S, \odot, \mathfrak{h})$ be a type. Recall that $\mathit{strands}(S)$ denotes the set of positive strands of S. For any set X, we denote the powerset of X by $\mathcal{P}(X)$. Let us use the function $\omega : \mathit{strands}(S) \to \mathcal{P}(\mathit{comp}(S))$ that maps each positive strand of S to the set of components of S containing the strand. For any $t \in \mathit{strands}(S)$, let $n(t)$ be the length of t and let $a(t)$ be the number of nodes labeled $*$, $\hat{*}$ or $\underline{*}$.

First, we define the weak type of $\texttt{cleanup}(\tau)$ which we will denote by S_{clean}. For any $s \in \mathit{strands}(S)$, we say that s *qualifies for S_{clean}* if there exists a component $D \in \omega(s)$ such that the system of inequalities $\{n(s) + (\ell - 1)a(s) \geq n(t) + (\ell - 1)a(t) \mid t \in \big(\mathit{strands}(\odot) \cup \mathit{strands}(D)\big)\}$ has a positive integer solution in the variable ℓ. Note that $n(t) + (\ell - 1)a(t)$ equals the actual length of a strand represented by t in a complex of dimension ℓ. So, intuitively, s qualifies if and only if for some dimension ℓ and some ℓ-complex of type τ, s has maximal length. The weak type S_{clean} consists of all qualified strands, in which all blockings have been cleared and $\hat{*}$- and $\underline{*}$-labeled nodes are relabeled to $*$.

Furthermore, we say that a strand $s \in S_{clean}$ *qualifies for mandatory*, if for each strand $t \in S_{clean}$, the strict inequality $n(s) + (\ell - 1)a(s) < n(t) + (\ell - 1)a(t)$ has *no* positive integer solution in ℓ. Denote with $\odot_{clean}$ the mandatory weak type of $\texttt{cleanup}(\tau)$. A strand s of S_{clean} belongs to $\odot_{clean}$, if s originates from a mandatory component, i.e., $\exists D \in \omega(s) : D \in \mathit{comp}(\odot)$, and s qualifies for mandatory.

The cleaning of τ, denoted $\texttt{cleanup}(\tau)$, equals $(S_{clean}, \odot_{clean}, \mathit{true})$.

Note that it is easy to decide whether a particular strand s qualifies for S_{clean}.

Indeed, for any D as above, it suffices to consider the inequalities $n(s)+(\ell-1)a(s) \geq n(t)+(l-1)a(t)$ for $t \in \mathit{strands}(\odot) \cup \mathit{strands}(D)$ (one of these inequalities is trivial). Each inequality yields a lower bound (if $a(s) > a(t)$) or an upper bound (if $a(s) < a(t)$) of $(n(t)-n(s))/(a(s)-a(t))+1$ on ℓ; if $a(s) = a(t)$ the inequality amounts to the simple condition $n(s) \geq n(t)$. All inequalities are simultaneously satisfied by some ℓ if and only if the greatest lower bound does not exceed the least upper bound (which must be non-negative) and all simple conditions are satisfied.

LEMMA 7.1. *Let $\tau = (S, \odot, \mathfrak{h})$ be a type; let ℓ be a natural number; let C be an ℓ-complex of type τ; let d be a strand belonging to* `cleanup`(C)*; and let $s = \mathit{stype}(d)$. Then s qualifies for S_{clean}.*

Proof. Strand d originates from a component D of complex C. Let $E = \mathit{stype}(D)$. Since complex C has type τ, we know that $\odot \sqsubseteq \mathit{stype}(C)$. Since d belongs to `cleanup`(C), the length of d is greater than or equal to the length of any strand d' in C. In particular, strand d is at least as long as every strand in component D and strand d is at least as long as every strand in a mandatory component of C. Recall that the length of d equals $n(s)+(\ell-1)a(s)$. In other words: $n(s)+(\ell-1)a(s) \geq n(t)+(\ell-1)a(t)$ for any strand $t \in \mathit{strands}(\odot) \cup \mathit{strands}(D)$. Hence, s qualifies for S_{clean}. $\square$

8. A Type System for DNAQL. In this section we introduce a type system for DNAQL and we show that it enjoys the desirable properties of soundness, maximality, and tightness.

A DNAQL expression e has a set of free variables, denoted $FV(e)$. If a type is fixed for each free variable, all the *complexvar*-subexpressions of e are *well typed* and their types are known. The *constant*-subexpressions of e are always well typed, and their types are known (cf. Figure 8.1). In the previous section we defined for each DNAQL operator, its counterpart operating on types. In this section we extend these rules to incorporate the for, if, and let expressions. By applying these rules, we can derive, from the types of the free variables and constants, for each subexpression of e, and ultimately for e itself, whether it is well typed.

More formally, a *type assignment* Γ is a mapping from a finite set of complex variables, $\mathit{dom}(\Gamma)$, to types. Let e be a DNAQL expression. If $\mathit{dom}(\Gamma) \supseteq FV(e)$, then we say that Γ is a type assignment *on* e.

The *typing relation* for DNAQL is defined in Figure 8.1. Here we write $\Gamma \vdash e : \tau$ to indicate that expression e *is assigned* type τ under type assignment Γ on e. If $\Gamma \vdash e : \tau$, then we call (Γ, τ) a *typing* of e. The domain of Γ is extended from variables to expressions as specified in Figure 8.1. The typing relation given in Figure 8.1 is clearly unambiguous, i.e., if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$. Note that conditions of the typing-rules of the if statement are mutual exclusive.

Recall the formal semantics of DNAQL (Section 5). When ℓ is not important, we refer to an ℓ-complex assignment simply as a complex assignment. Let Γ a type assignment, and let ν be a complex assignment. We naturally say that ν *has type* Γ if $\mathit{dom}(\nu) = \mathit{dom}(\Gamma)$ and for all $x \in \mathit{dom}(\nu)$, we have $\nu(x) : \Gamma(x)$, i.e., complex $\nu(x)$ has type $\Gamma(x)$. The set of all complex assignments of Γ is denoted by $[\![\Gamma]\!]$.

8.1. Sound. Given a DNAQL expression e and given a type assignment Γ on e, e is called *ℓ-safe*, for a fixed dimension ℓ, if for any ℓ-complex assignment ν and any ℓ-counter assignment γ on e, with $\nu \in [\![\Gamma]\!]$, the result $[\![e]\!]^{\ell}(\nu, \gamma)$ is well defined. If e is ℓ-safe for every ℓ, then we say that e is *safe*.

If e is safe under Γ and, moreover, for every dimension ℓ, every ℓ-complex assignment ν and every ℓ-counter assignment γ, if $\nu \in [\![\Gamma]\!]$ then $[\![e]\!]^{\ell}(\nu, \gamma)$ has type τ, then we say that e is *safe* under Γ *with output type* τ. We denote this by $\Gamma \models e : \tau$.

$$\frac{x \in \mathit{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{e \text{ is a } \langle \mathit{constant} \rangle \text{ expression}}{\Gamma \vdash e : (S, S, \mathit{true}) \qquad S = \mathit{stype}(e)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \cup e_2 : \tau_1 \cup \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 - \tau_2 \text{ is well defined}}{\Gamma \vdash e_1 - e_2 : \tau_1 - \tau_2}$$

$$\frac{\Gamma \vdash e : \tau \qquad \texttt{hybridize}(\tau) \text{ is well defined and has terminating hybridization}}{\Gamma \vdash \texttt{hybridize}(e) : \texttt{hybridize}_t(\tau)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ligate}(e) : \texttt{ligate}(\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{flush}(e) : \texttt{flush}(\tau)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \sigma \in \{\#_2, \#_3, \#_4, \#_6, \#_8\}}{\Gamma \vdash \texttt{split}(e, \sigma) : \texttt{split}(\tau, \sigma)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \sigma \in (\Omega \cup \Theta) \qquad \texttt{block}(\tau, \sigma) \text{ is well defined}}{\Gamma \vdash \texttt{block}(e, \sigma) : \texttt{block}(\tau, \sigma)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \sigma \in (\Omega \cup \Theta) \qquad \texttt{blockfrom}(\tau, \sigma) \text{ is well defined}}{\Gamma \vdash \texttt{blockfrom}(e, \sigma) : \texttt{blockfrom}(\tau, \sigma)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \texttt{blockexcept}(\tau) \text{ is well defined}}{\Gamma \vdash \texttt{blockexcept}(e, i) : \texttt{blockexcept}(\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{cleanup}(e) : \texttt{cleanup}(\tau)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x := \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \ := \ e_1 \texttt{ in } e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x := \tau_1] \vdash e_2 : \tau_1}{\Gamma \vdash \texttt{for } x := e_1 \texttt{ iter } i \texttt{ do } e_2 : \tau_1}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash x : (S_x, \odot_x, \mathfrak{h}_x) \\ \odot_x = \emptyset \qquad |\mathit{comp}(S_x)| = 1 \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x := (S_x, \mathit{comp}(S_x), \mathfrak{h}_x)] \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \texttt{if empty}(x) \texttt{ then } e_1 \texttt{ else } e_2 : \tau_1 \vee \tau_2 \qquad \mathfrak{h} = (S_1 \cup S_2 \text{ is saturated})}$$

$$\frac{\Gamma \vdash x : (S_x, \odot_x, \mathfrak{h}_x) \qquad \odot_x = \emptyset \qquad |\mathit{comp}(S_x)| > 1 \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{if empty}(x) \texttt{ then } e_1 \texttt{ else } e_2 : \tau_1 \vee \tau_2 \qquad \mathfrak{h} = (S_1 \cup S_2 \text{ is saturated})}$$

FIGURE 8.1. *Typing relation of DNAQL.*

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 Theorem 9.1 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 output types from the given input types. The type-checking algorithm induced by Figure 8.1, given e and Γ as above, judges whether e is *well typed* under Γ, and, if so, infers its output type τ. This is denoted by $\Gamma \vdash e : \tau$.

Let $\vdash$ denote a typing relation. We say that typing relation $\vdash$ is *sound*, if for every expression e, type assignment Γ on e and type τ, it holds that if $\Gamma \vdash e : \tau$, then $\Gamma \models e : \tau$, i.e., e safe is under Γ with output type τ.

THEOREM 8.1. *The DNAQL typing relation is sound.*

Proof. Let $\Gamma \vdash e : \tau$. By induction on e we show that e is safe under Γ with output type τ. Below we let ℓ be an arbitrary dimension, ν be an ℓ-complex assignment on e with $\nu \in [\![\Gamma]\!]$, and γ an arbitrary ℓ-counter assignment on e. To reduce clutter, the dimension ℓ is often not explicitly mentioned.

Variable. Let $e = x \in \mathit{dom}(\nu)$ be a variable. By Figure 8.1, $\Gamma \vdash x : \Gamma(x)$. Hence $\nu(x) : \Gamma(x) = \tau$. Consequently, $[\![e]\!](\nu, \gamma) = \nu(x) : \tau$ as required.

Constant. If e is a constant, the soundness property holds by definition, noting that every constant in the DNAQL language is saturated.

Union. Let $e = e_1 \cup e_2$. By induction, we assume that $C_1 = [\![e_1]\!](\nu, \gamma)$ and $C_2 = [\![e_2]\!](\nu, \gamma)$ are defined and they are of types $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$. By definition the union of two sticker complexes is defined and so $C = C_1 \cup C_2 = [\![e]\!](\nu, \gamma)$ is defined. We have $\tau = \tau_1 \cup \tau_2 = (S_1 \cup S_2, \odot_1 \cup \odot_2, \mathfrak{h})$. It suffices to show that C is of type τ. We verify the three conditions in the definition a complex having a particular type.

Let $D \in \mathit{comp}(C)$. Hence $D \in \mathit{comp}(C_1)$ or $D \in \mathit{comp}(C_2)$. Consequently, $\mathit{stype}(D)$ is subsumed by S_1 or by S_2, and thus by $S_1 \cup S_2$.

Let $s \in \mathit{comp}(\odot_1 \cup \odot_2)$. If $s \in \mathit{comp}(\odot_1)$, then s is subsumed by $\mathit{stype}(C_1)$, and if $s \in \mathit{comp}(\odot_2)$, then s is subsumed by $\mathit{stype}(C_2)$. Hence s is subsumed by $\mathit{stype}(C)$.

If $\mathfrak{h} = \mathit{true}$, then by definition of union on types, both (1) S_1 and S_2 are mutually non-interacting, and (2) S_i is saturated or $\mathfrak{h}_i = \mathit{true}$ for all $i \in \{1, 2\}$. Assume to the contrary that C is not saturated. Let u and v be mutually interacting nodes of C. In $\mathit{stype}(C)$, nodes u and v are represented by nodes u' and v' respectively, which are mutually interacting nodes of $\mathit{stype}(C)$. Since $\mathit{stype}(C)$ is subsumed by S, nodes u' and v' in $\mathit{stype}(C)$ correspond to nodes u'' and v'' in $S_1 \cup S_2$. By (1), u'' and v'' belong both to S_1 or both to S_2. In particular, nodes u and v must both belong to C_1 or both to C_2. Without loss of generality, we may assume that u and v both belong to C_1 (and thus u'' and v'' both belong to S_1). But then S_1 would not be saturated, whence $\mathfrak{h}_1 = \mathit{true}$ by (2). Hence C_1 is saturated, which is in contradiction with u and v being mutually interacting nodes of C_1.

Difference. Let $e = e_1 - e_2$. By induction, we assume that $C_1 = [\![e_1]\!](\nu, \gamma)$ and $C_2 = [\![e_2]\!](\nu, \gamma)$ are defined and they have types $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ resp. $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$. It is given that τ_1 and τ_2 fulfill the restrictions posed by the definition of difference on types and that e is of type $\tau = \tau_1 - \tau_2 = (S, \odot, \mathfrak{h})$. First we prove that $C = [\![e]\!](\nu, \gamma)$ is defined. The definition of difference on sticker complexes imposes three restrictions on the sticker complexes C_1 and

C_2. We prove that each restriction is met.

1. There are no matchings, no immobilizations, no blockings, no nodes labeled $\underline{*}$ and no nodes labeled $\hat{*}$ in S_1 and S_2. Thus, there can be no immobilizations, matchings or blockings in C_1 or C_2.
2. The components of τ_1 and τ_2 are all positive, noncircular, of equal length and with the same number of nodes labeled $*$. Thus, C_1 and C_2 consist of positive, noncircular and equal length strands.
3. All the strands in τ_2 end on $\#_4$ and do not contain $\#_5$. Thus, all strands in C_2 end on $\#_4$ and do not contain $\#_5$.

We may thus conclude that C is well defined. Next, we prove that C is of type τ.

By the definition of difference on complexes, $D \in \mathit{comp}(C)$ implies that D is subsumed by C_1, but not subsumed by C_2. Consequently, $\mathit{stype}(D)$ is subsumed by S_1 and (1) $\mathit{stype}(D)$ is not subsumed by $\odot_2$ or (2) $\mathit{stype}(D)$ contains $*$, $\hat{*}$, or $\underline{*}$ (or both). Thus $\mathit{stype}(D) \in \mathit{comp}(\mathit{data}(S_1) \cup T_2)$ where T_2 is the complex containing all components of S_1 that are not subsumed by $\odot_2$ — as required.

Let $s \in \mathit{comp}(\odot)$. By definition of $\odot$, $s \in \mathit{comp}(\odot_1)$ and is not subsumed by S_2. Moreover, since $s \cong \mathit{stype}(D)$ is not subsumed by S_2, but $\mathit{stype}(C_2)$ is subsumed by S_2, we have by Lemma 6.1 that D is not subsumed by C_2. Thus $D \in \mathit{comp}(C)$ whence s is subsumed by $\mathit{stype}(C)$ as desired.

By definition $\mathfrak{h} = \mathit{true}$, and indeed the result of the difference operation is a set of positive strands, and therefore trivially saturated.

Hybridize. Let $e = \texttt{hybridize}(e')$. By induction, we assume that $[\![e']\!](\nu, \gamma) = C'$ is defined and is of type $\tau' = (S', \odot', \mathfrak{h}')$. Moreover, $\tau = \texttt{hybridize}(\tau') = (S, \odot, \mathfrak{h})$ is defined.

If $\mathfrak{h}' = \mathit{true}$, then $\tau = \tau'$ and C' is saturated. Hence $\texttt{hybridize}(C') = C'$ and so $\texttt{hybridize}(C')$ is clearly of type τ.

Assume now that $\mathfrak{h}' = \mathit{false}$. Since τ is defined, hybridization terminates for weak type S' (i.e., $\texttt{hybridize}_t(S')$ is defined). By Theorem 3.2 there is no alternating cycle in the hybridization graph of S' (the definition of hybridization graph is straightforwardly extended to sticker complex types by using the extended complementarity relation). Consequently, there is no alternating cycle in the hybridization graph of C', and therefore C' has terminating hybridization (ℓ-cores and ?-labeled probes can never engage in an alternating cycle). Hence $[\![e]\!](\nu, \gamma)$ is well defined.

It remains to show now that $[\![e]\!](\nu, \gamma) = C$ is of type τ. Let $D \in \mathit{comp}(C)$. We show that $\mathit{stype}(D)$ is subsumed by S; recall that

$$S = \left(\bigcup_{NC \sqsubseteq X \sqsubseteq S'} \texttt{hybridize}_t(X) \right) \cup \{\texttt{immob}(?) \mid \texttt{immob}(?) \sqsubseteq S'\},$$

Recall that D (as a component of C) is a finished saturated hybridization extension of the disjoint union of some multiset $\mathcal{D}$ of components of C'. We distinguish three cases:

1. $\mathcal{D}$ contains no probe. Note that $\mathcal{D}$ may contain nodes labeled from $\bar{\Lambda}$, but then these are already matched in C'. In this case $\mathit{stype}(D) \in \mathit{comp}(\texttt{hybridize}_t(\mathit{stype}(C') \setminus \{\texttt{immob}(?)\}))$. Since $\odot' \sqsubseteq \mathit{stype}(C') \sqsubseteq S'$, we can view $\mathit{stype}(C') \setminus \texttt{immob}(?)$ as an X such that $NC \sqsubseteq X \sqsubseteq S'$. Hence $\mathit{stype}(D)$ is clearly subsumed by S in this case.

2. $\mathcal{D}$ consists of a probe. In this case $stype(D) \equiv \texttt{immob}(?)$. In particular, $\texttt{immob}(?)$ occurs as a separate component in $stype(C')$ which is in turn subsumed by S'. Hence, again $stype(D)$ is subsumed by S in this case.
3. $\mathcal{D}$ contains a probe in addition to other components of C'. Since D is a component, the probe is involved in the matching that creates D. Note also that $\mathcal{D}$ contains exactly one probe, since probes are immobilized and components of sticker complexes can contain at most one immobilized node. The *stype* of the probe is $\texttt{immob}(?)$, and the *stype* of the component containing the core r having the atomic value node that is matched to the probe has a node representing r that is labeled by $*$ or $\hat{*}$. Since both $(*, ?)$ and $(\hat{*}, ?)$ are complementary pairs of symbols, we conclude that $stype(D) \in comp(\texttt{hybridize}_t(stype(C')))$. As in Case 1, we can see $stype(C')$ as an X such that $NC \sqsubseteq X \sqsubseteq S'$. Hence $stype(D)$ is subsumed by S.

Let $s \in comp(\odot)$. We show that s is subsumed by $stype(C)$. By definition, either (1) $s \in comp(\texttt{hybridize}_t(NC))$ and $s \in comp(\texttt{hybridize}_t(S'))$ or (2) $s = \texttt{immob}(?) \in comp(\odot')$ and there is no component in S' with an free node labeled with $*$ or $\hat{*}$.

1. Assume case (1) holds. Since $s \in comp(\texttt{hybridize}_t(NC))$, and NC consists of the mandatory components except $\texttt{immob}(?)$, we have $s = stype(D)$ for some MHE component D w.r.t. C' that is a saturated hybridization extension of the disjoint union of some multiset $\mathcal{D}$ of components from C'. Since $\texttt{immob}(?)$ is not in NC, the matchings used to make D do not involve pairs of complementary atomic value nodes. Moreover, since s also belongs to $\texttt{hybridize}_t(S')$, D is finished w.r.t. C'. Hence $s = stype(D)$ is subsumed by $stype(C)$.
2. Assume now that case (2) holds. Since $\odot'$ is subsumed by $stype(C')$, there is a component D' of C' that is a probe. By the given, this probe cannot be involved in the hybridization of C', so D' also occurs as a separate component of C. It follows that $s = stype(D')$ is subsumed by $stype(C)$.

By definition, $\mathfrak{h} = true$ and indeed C, being the result of a hybridization, is saturated.

Ligate. Let $e = \texttt{ligate}(e')$. By induction, we assume that $[\![e']\!](\nu, \gamma)$ is defined and it is of type $\tau' = (S', \odot', \mathfrak{h}')$.
The operation ligate is defined on all complexes, thus $[\![e]\!](\nu, \gamma)$ is defined.
On types, operation $\texttt{ligate}$ is defined as performing ligate on the weak type S and on the mandatory weak type. Ligate does not change the state of $\mathfrak{h}'$. From this it is clear that $[\![e]\!](\nu, \gamma)$ is of type τ.

Flush. Let $e = \texttt{flush}(e')$. By induction, we assume that $C' = [\![e']\!](\nu, \gamma)$ is defined and C' is of type $\tau' = (S', \odot', \mathfrak{h}')$. Let $\tau = (S, \odot, \mathfrak{h})$.
The flush operation is defined on any complex. As a result, $C = \texttt{flush}(C') = [\![e]\!](\nu, \gamma)$ is defined.
Let $D \in comp(C)$. Then $D \in comp(C')$ and $\iota_D \neq \emptyset$, where ι_D is the set of immobilized nodes of D. Since C' is of type τ', there is a $t \in comp(S')$ with $\iota_t \neq \emptyset$ such that $t \equiv stype(D)$. Hence $stype(D)$ is subsumed by S.
Let $s \in comp(\odot)$. Then $s \in comp(\odot')$ and $\iota_s \neq \emptyset$. Since $\odot'$ is subsumed by $stype(C')$, there is a $D \in comp(C')$ such that $s \cong stype(D)$. Since $\iota_s \neq \emptyset$, also $\iota_D \neq \emptyset$, whence $D \in comp(C)$ and thus s is subsumed by $stype(C)$ as

desired.

The flush operation does not change the state of $\mathfrak{h}'$, as required.

Split. Let $e = \texttt{split}(e', \mathit{label})$, with *label* the label of a split point. By induction, we assume that $[\![e']\!](\nu, \gamma)$ is defined and it is of type τ'.

The split operation is defined on any complex, thus $[\![e]\!](\nu, \gamma)$ is defined.

The split operation on types is defined as the split operation on the weak type, and making components mandatory if they stem from a mandatory component. Clearly, $[\![e(\nu, \gamma)]\!]$ is of type τ.

Block. Let $e = \texttt{block}(e', \sigma)$ with σ a symbol in $\Omega \cup \Theta$. By induction, we assume that $C' = [\![e']\!](\nu, \gamma)$ is defined and has type τ'.

By the fact that $\Gamma \vdash e : \tau$, it is known that τ' is saturated, and $[\![e]\!](\nu, \gamma)$ is defined. Hence C' is saturated.

The block operation on types is defined as the block operation on complexes, and mandatory components remain mandatory. Note that the $\mathfrak{h}$-bit of τ is true by definition, hence we must verify that C is saturated. Since C' is saturated and the block operation on complexes preserves saturation, C is indeed saturated. As a result, $[\![e]\!](\nu, \gamma)$ is of type τ.

Block-From. Let $e = \texttt{blockfrom}(e', \sigma)$ with $\sigma \in \Omega \cup \Theta$. By induction, $C' = [\![e']\!](\nu, \gamma)$ is well-defined and of type τ'.

As in the previous case, $C = [\![e]\!](\nu, \gamma)$ is well-defined since C' is saturated. Again the $\mathfrak{h}$-bit of τ is true and indeed C is saturated. To verify that C is of weak type $S = \texttt{blockfrom}(S', \sigma)$, let $D \in \mathit{comp}(C)$. Then D is obtained from a component $D' \in \mathit{comp}(C')$. Any node x in an ℓ-core r occurring in a σ-blocking range of D' is free, so that in $\mathit{stype}(D)$ r is represented by an free node r' labeled $*$. In D, all nodes x of r are blocked, yielding an ℓ-core of type $\underline{*}$. In S, the node r' is relabeled with $\underline{*}$. Hence, $\mathit{stype}(D)$ is subsumed by S as desired. The reasoning that $\odot = \texttt{blockfrom}(\odot', \sigma)$ is subsumed by $\mathit{stype}(C)$ is similar.

Block-Except. Let $e = \texttt{blockexcept}(e', i)$. By induction, we assume that $C' = [\![e']\!](\nu, \gamma)$ is defined and has type $\tau' = (S', \odot', \mathfrak{h}')$.

First, we show that $C = [\![e]\!]^{\ell}(\nu, \gamma) = \texttt{blockexcept}(C', \gamma(i))$ is defined. Three conditions constrain the well-definedness of the block-except operation. First, the natural number must be smaller than the dimension ℓ. By definition, $1 \leq \gamma(i) \leq \ell$. Secondly, for every ℓ-vector of C' either all nodes are free or all nodes are closed. Let v' be an ℓ-vector in C', with ℓ-core r', let v be the representation of v' in $\mathit{stype}(C')$ and let r be the node in $\mathit{stype}(C')$ representing r'. Node r can be of three different types:

1. Node r is of type $*$: none of the nodes of v' are blocked, and none of the nodes are matched, due to Conditions 1 and 2 of the block-except operation on types.
2. Node r is of type $\hat{*}$: a single node x of r' is not blocked. Node x has to be matched, due to Conditions 1 and 3 of the block-except operation on types. Moreover, the $\#_3$ and $\#_4$ of v' are closed.
3. Node r is of type $\underline{*}$: all nodes of r' are closed. Due to Conditions 1 and 3 of the definition of the block-except operation on types, all nodes of v' are closed.

Thirdly, C' is saturated, due to condition 4 of the block-except operation on types.

On types, operation `blockexcept` is defined as performing block-except on the weak type and the mandatory weak type. Since C' is saturated and the block-except operation on complexes preserves saturation, C is indeed saturated.

Cleanup. Let $e = \texttt{cleanup}(e')$. By induction, we assume that $C' = [\![e']\!](\nu, \gamma)$ is defined and has type $\tau' = (S', \odot', \mathit{true})$. Let $\tau = \texttt{cleanup}(\tau') = (S, \odot, \mathit{true})$. The cleanup operation is defined on all complexes, so we must only verify that $C = \texttt{cleanup}(C')$ is of type τ.

Let D be a component of C. Then D is a strand of length m with m the length of the longest positive strand in C'. By Lemma 7.1 (applied to C') we obtain that $\mathit{stype}(D)$ qualifies for $S = S'_{\mathit{clean}}$. Hence $\mathit{stype}(D)$ belongs to S whence $\mathit{stype}(D)$ is subsumed by S as desired.

Let $s \in \mathit{comp}(\odot)$. Consequently, from the definition of $\texttt{cleanup}$ on types, it is known that there is a component $D \in \omega(s)$ such that $D \in \mathit{comp}(\odot')$, s qualifies for S'_{clean} and s qualifies for mandatory. By $D \in \mathit{comp}(\odot')$ there is a component $E \in \mathit{comp}(C')$ such that $D \equiv \mathit{stype}(E)$. In particular, there is a strand $d \in \mathit{strands}(E)$ such that $s \equiv \mathit{stype}(d)$. It remains to be shown that $d \in \mathit{comp}(C)$. Thereto, we must show that the length of d is greater than or equal to the length of d' for any $d' \in \mathit{strands}(C')$. Note that the length of d' is smaller than or equal to the length of $d_{\max}$ for any $d_{\max} \in \mathit{strands}(C)$. By Lemma 7.1, $\mathit{stype}(d_{\max})$ qualifies for S, whence $n(s) + (\ell - 1)a(s) \geq n(\mathit{stype}(d_{\max})) + (\ell - 1)a(\mathit{stype}(d_{\max}))$. Since the number on the left-hand side equals the length of d, and the number on the right-hand side equals the length of $d_{\max}$, we are done.

The result of the cleanup operation is a set of positive strands, and therefore trivially saturated.

Let. Let $e = \texttt{let}\ x := e_1\ \texttt{in}\ e_2$. By induction, we assume that $C_1 = [\![e_1]\!](\nu, \gamma)$ and $C_2 = [\![e_2]\!](\nu[x := C_1], \gamma)$ are defined and of type τ_1 and τ_2, respectively. Hence, $[\![e]\!](\nu, \gamma) = C_2$ is defined and of type τ_2.

For. Let $e = \texttt{for}\ x := e_1\ \texttt{iter}\ i\ \texttt{do}\ e_2$. By induction, we assume that $C_0 = [\![e_1]\!](\nu, \gamma)$ and $[\![e_2]\!](\nu[x := C_{n-1}], \gamma[i := n]) = C_n$ for all $n \in \{1, \ldots, \ell\}$ are defined, and C_0 is of type τ_1. Moreover, by the let part above, if C_{n-1} is of type τ_1, then C_n is of type τ_1 for all $n \in \{1, \ldots, \ell\}$. Hence $C_\ell = [\![e]\!](\nu, \gamma)$ is defined and of type τ_1.

If. Let $e = \texttt{if}\ \texttt{empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. There are four possible ways of typing this expression. By induction, we assume that $[\![e_1]\!](\nu, \gamma)$ and $[\![e_2]\!](\nu, \gamma)$ are defined and have type $\tau_1 = (S_1, \odot, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot, \mathfrak{h}_2)$, respectively. Also, the variable x is defined and typed. Hence $[\![e]\!](\nu, \gamma)$ is also defined.

1. Only the empty complex can have the type with no components. Thus, the then-part of the test is evaluated. By induction, $[\![e]\!](\nu, \gamma)$ is defined and of type τ_1, whence the same holds for $[\![e]\!](\nu, \gamma) = [\![e_1]\!](\nu, \gamma)$.
2. If $\odot_x$ is not the empty complex, then the empty complex cannot have type $\Gamma(x)$. Thus, the else-part of the test is evaluated. By induction, $[\![e_2]\!](\nu, \gamma)$ is defined and has type τ_2, whence the same holds for $[\![e]\!](\nu, \gamma) = [\![e_2]\!](\nu, \gamma)$.
3. If there is exactly one non-mandatory component in $\Gamma(x)$, then effectively, if $\nu(x)$ is nonempty, it is not just of type $\Gamma(x)$ but actually of type $(S_x, S_x, \mathfrak{h}_x)$ as used in the typing rule to type check the else-part. Since the type for e inferred by the rule is the minimal upper bound of the types inferred for the then- and else-parts, soundness follows imme-

diately.

4. The fourth inference rule is proven similar to the third rule.

□

Example 8.1. *Recall the program from Example 5.1 in Section 5.*

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; so we are making it hard on the type checker. The subexpression* $e_1 = \texttt{hybridize}(x_1 \cup \texttt{immob}(\overline{a}))$ *is typed as* $(S_1^?, \emptyset, \mathit{true})$*, where* $S_1^?$ *consists of the following components: (i)* S_1 *itself; (ii)* $\texttt{immob}(?)$*; and (iii) 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* $(S, \emptyset, \mathit{true})$*.*

Example 8.2. *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, 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 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.

8.2. Maximal. Let e be a DNAQL expression and let Γ be a type assignment on e. We say that a typing relation $\vdash$ for DNAQL is *u-maximal* (u is short for uniform) for e if $\Gamma \vdash e : \tau$ for some τ whenever e is safe under Γ. We say that typing relation $\vdash$ is *d-maximal* (d is short for dimension) if $\Gamma \vdash e : \tau$ for some τ whenever there exists some dimension ℓ for which e is ℓ-safe under Γ. Note that d-maximality requires safety only for some fixed dimension, whereas u-maximality requires safety uniformly for all dimensions.

A DNAQL expression consisting of a single operation is called an *atomic expression.* In particular, `if`, `for`, and `let` expressions are not considered to be atomic.

THEOREM 8.2. *For every atomic expression e, the DNAQL type relation is u-maximal for e. In addition, unless e invokes the difference operator, the typing relation is d-maximal for e.*

Proof. We will show that if atomic expression e is ill-typed under Γ, then e is not safe under Γ, i.e., there exists a specific input complex assignment of type Γ on which the evaluation of e is undefined.

Union The union operation is always defined on the type level. The theorem thus holds trivially.

Difference The difference of types $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ is not defined if one of its three conditions is not satisfied. Suppose the types do not adhere to one of these conditions. Next, we construct (for each condition) two complexes having type τ_1 resp. τ_2 such that the difference of these complexes is not defined.

1. Suppose there is a node x that is matched, immobilized, blocked or labeled with $\underline{*}$ or $\hat{*}$ in τ_1 or τ_2. Recall that a node labeled with $\underline{*}$ or $\hat{*}$ represents a (possibly partially) blocked ℓ-core, and so every complex having such a type will have a blocked node (as $\ell \geq 2$ by definition, there will also be blocked node in case of $\hat{*}$). Suppose x is present in τ_1 (the proof is similar if x is in τ_2). Let D be the component of S_1 containing x. Let C be a complex such that $\mathit{stype}(C) = D \cup \odot_1$. Complex C has type τ_1 and therefore has a matching, blocking, or immobilization. So difference is not defined.
2. This case will be split into two sub cases: (a) there is a negative or circular strand in $\mathit{strands}(S_1)$ or $\mathit{strands}(S_2)$, and (b) assuming that there are only positive noncircular strands, two strands s_1 and s_2 in $\mathit{strands}(S_1) \cup \mathit{strands}(S_2)$ are of different length or have a different number of $*$-labeled nodes.
   (a) Let d be a strand in S_1 (S_2) that is negative or circular, and let D be the component in which d occurs. Let C_1 (C_2) be a complex with $\mathit{stype}(C_1) \equiv \odot_1 \cup D$ ($\mathit{stype}(C_2) \equiv \odot_2 \cup D$). Hence, complex C_i is of type τ_i, for $i \in \{1, 2\}$. Moreover, C_1 (C_2) has a negative or circular strand, whence the difference $C_1 - C_2$ is undefined.
   (b) Let s_1 and s_2 in $\mathit{strands}(S_1) \cup \mathit{strands}(S_2)$, having a different length or a different number of $*$-labeled nodes. Denote with $n(s_1)$ resp. $n(s_2)$ the length of s_1 resp. s_2 and denote with $a(s_1)$ resp. $a(s_2)$ the number of $*$-labeled nodes in s_1 resp. s_2. The length of any strand of weak type s_1 resp. s_2 is expressed by $n(s_1) + (\ell - 1)a(s_1)$ resp. $n(s_2) + (\ell - 1)a(s_2)$ (ℓ is the dimension). For the difference operation to be u-maximal, we must show that any two strands having respective types s_1 and s_2 should be of equal length for all values of ℓ. We distinguish three cases, and prove for each case that all strands having respective types s_1 and s_2 have a different length:
      i. Suppose that $n(s_1) = n(s_2)$ and $a(s_1) \neq a(s_2)$. Now, $n(s_1) + (\ell - 1)a(s_1) = n(s_2) + (\ell - 1)a(s_2)$ implies $(\ell - 1)(a(s_1) - a(s_2)) = 0$ — a contradiction as $\ell - 1$ and $a(s_1) - a(s_2)$ are both nonzero.
      ii. Suppose that $n(s_1) \neq n(s_2)$ and $a(s_1) = a(s_2)$. Now, $n(s_1) + (\ell - 1)a(s_1) = n(s_2) + (\ell - 1)a(s_2)$ implies $n(s_1) = n(s_2)$ — a

contradiction.

   iii. $n(s_1) \neq n(s_2)$ and $a(s_1) \neq a(s_2)$: s_1 and s_2 are of equal length if $\ell - 1 = (n(s_2) - n(s_1))/(a(s_1) - a(s_2))$. Without loss of generality we may assume that $a(s_1) > a(s_2)$. If $\ell - 1$ has a value strictly larger than $\max\{n(s_2) - n(s_1)\}$, then the above condition cannot be satisfied, i.e., two strands having respective types s_1 and s_2 will have different lengths.

3. Let D be a strand of S_2 not ending with a node labeled $\#_4$. Let C_1 be a complex having type τ_1. Let C_2 be a complex with *stype*$(C_2) \equiv \odot_2 \cup D$. By definition, C_2 has type τ_2 and has a strand that does not end with a node labeled with $\#_4$. Hence, $C_1 - C_2$ is undefined.
   Let D be a strand of S_2 containing a node labeled $\#_5$. Let C_1 be a complex having type τ_1. Let C_2 be a complex with *stype*$(C_2) \equiv \odot_2 \cup D$. By definition, C_2 has type τ_2 and has a strand with a node labeled with $\#_5$. Hence, $C_1 - C_2$ is undefined.

Hybridize Let $\tau = (S, \odot, \mathfrak{h})$. Assume the hybridize operation is undefined on type τ. Hence both $\mathfrak{h} = \textit{false}$ and S has non-terminating hybridization. Let C be a complex with *stype*$(C) \equiv S$ and with an alternating cycle in its hybridization graph. Note that such C can always be constructed by replacing $*$-nodes in S by ℓ-cores using always the same atomic value symbol and replacing ?-nodes by the complement of the chosen atomic value symbol. Consequently, `hybridize` is not defined for C, and C is of type τ because $\mathfrak{h} = \textit{false}$.

Ligate, Flush, Split These operations are always defined on the type level.

Block The block operation is undefined on type τ if τ is not saturated. By the definition of saturated, there is a complex C having type τ such that the complex is not saturated. The block operation is undefined on unsaturated complexes.

Block-From Similar to the proof for `block`.

Block-Except The block-except operation is not defined if one of its four conditions is violated:

1. If there is a node x labeled with $*$, $\hat{*}$, or $\underline{*}$ that is not preceded by a node labeled $\#_3$ or not followed by a node labeled $\#_4$, then x corresponds to a ℓ-core that is not part of a ℓ-vector for each complex C of type S. Hence C is not an ℓ-complex, and so `blockexcept`(C, i) is undefined for any natural number $1 \leq i \leq \ell$.
2. Let D be a component of S with a $*$-labeled node x such that (1) x is not free, (2) x is not preceded by a free node (labeled $\#_3$), or (3) x is not followed by a free node (labeled $\#_4$). Let C be a complex with *stype*$(C) \equiv \odot \cup D$. Complex C contains an ℓ-vector with both free and closed nodes. By definition, `blockexcept`(C, i) is undefined for any natural number $1 \leq i \leq \ell$.
3. Let D be a component of S with a $\hat{*}$ or $\underline{*}$-labeled node x such that (1) x is free, (2) x is not preceded by a closed node (labeled $\#_3$), or (3) x is not followed by a closed node (labeled $\#_4$). Let C be a complex with *stype*$(C) \equiv \odot \cup D$. Complex C contains an ℓ-vector with both free and closed nodes. By definition, `blockexcept`(C, i) is undefined for any natural number $1 \leq i \leq \ell$.
4. If S is not saturated, by the definition of saturatedness, there is a complex having type S that is not saturated. Consequently, block-except is

undefined on this complex.

Cleanup This operation is always defined on the type level.

∎

8.3. Tightness. Let e be a DNAQL expression. A typing relation $\vdash$ for DNAQL is called *tight* for e if for all type assignments Γ on e, whenever $\Gamma \vdash e : \tau$ and $\Gamma \models e : \tau'$ for some types τ and τ', then $\tau \preceq \tau'$. The notion of tightness was introduced by Papakonstaninou and Velikhov [18].

THEOREM 8.3. *For every atomic expression, the DNAQL type relation is tight.*

Proof. Let e be an atomic expression, and let Γ be a type assignment on e. Let $\Gamma \vdash e : \tau$ and $\Gamma \models e : \tau''$. We show that $\tau \preceq \tau''$. Let $\tau = (S, \odot, \mathfrak{h})$, and $\tau'' = (S'', \odot'', \mathfrak{h}'')$. Let $a, b \in \Lambda$ with $a \neq b$. With a^ℓ (b^ℓ) we denote a sequence of ℓ nodes labeled a (b).

Atomic expression e is of one of the following forms.

Union We have $e = x_1 \cup x_2$. Let $\Gamma(x_1) = \tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\Gamma(x_1) = \tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$. Then $\tau = \tau_1 \cup \tau_2$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.3.

1. Proof of $S \sqsubseteq S''$. Let $D \in \mathit{comp}(S)$. If $D \in \mathit{comp}(S_1)$, let C_1 be such that $\mathit{stype}(C_1) \equiv (\odot_1 \cup D)$, and let C_2 be such that $\mathit{stype}(C_2) \equiv \odot_2$. Complex C_1 has type τ_1 (recall that by the definition of type, C_1 is saturated if $\mathfrak{h}_1 = \mathit{true}$), and complex C_2 has type τ_2. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 \cup C_2 : \tau''$, and so $D \in \mathit{comp}(S'')$.
2. Proof of $\odot'' \sqsubseteq \odot$. We show that if $D \notin \mathit{comp}(\odot)$, then $D \notin \mathit{comp}(\odot'')$. Let $D \in \mathit{comp}(S)$ and $D \notin \mathit{comp}(\odot)$. Let complex C_1 be such that $\mathit{stype}(C_1) \equiv \odot_1$ and let complex C_2 be such that $\mathit{stype}(C_2) \equiv \odot_2$. Complexes C_1 and C_2 are of types τ_1 and τ_2, respectively. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 \cup C_2 : \tau''$. Because $D \notin \mathit{comp}(\odot)$, $D \notin \mathit{comp}(\odot_1)$ and $D \notin \mathit{comp}(\odot_2)$. Hence complex $C_1 \cup C_2$ does not contain a component of type D. Thus, $D \notin \mathit{comp}(\odot'')$.
3. Proof of $\mathfrak{h}'' = \mathit{true}$ implies that τ is saturated. Assume that τ is not saturated. We show that $\mathfrak{h}'' = \mathit{false}$. Since τ is not saturated, $\mathfrak{h} = \mathit{false}$. Thus, by definition of $\tau = \tau_1 \cup \tau_2$, (a) S_1 and S_2 are mutually interacting, or (b) S_i is not saturated and $\mathfrak{h}_i = \mathit{false}$ for some $i \in \{1, 2\}$.
   (a) Suppose that S_1 and S_2 are mutually interacting, i.e., there is a component D_1 in S_1 with a node u and a component D_2 in S_2 with a node v, such that u and v are free and complementary labeled and D_1 and D_2 are not both immobilized. Let C_1 and C_2 be such that $\mathit{stype}(C_1) \equiv (\odot_1 \cup D_1)$ and $\mathit{stype}(C_2) \equiv (\odot_2 \cup D_2)$, respectively. Thus $C_1 \cup C_2$ is not saturated. Complexes C_1 and C_2 are of types τ_1 and τ_2, respectively. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 \cup C_2 : \tau''$. Since $C_1 \cup C_2$ is not saturated, $\mathfrak{h}'' = \mathit{false}$
   (b) Suppose that S_i is not saturated and $\mathfrak{h}_i = \mathit{false}$ for some $i \in \{1, 2\}$. According to Lemma 6.2, τ_i is not saturated. Hence there is a unsaturated complex C of type τ_i. Without loss of generality we assume $i = 1$. Let C' be a complex of type τ_2. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C$ and $\nu(x_2) = C'$. As $\Gamma \models e : \tau''$, $C \cup C' : \tau''$. Since $C \cup C'$ is not saturated, $\mathfrak{h}'' = \mathit{false}$.

Difference We have $e = x_1 - x_2$. Let $\Gamma(x_1) = \tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\Gamma(x_2) = \tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$. Then $\tau = \tau_1 - \tau_2$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.3.

1. Proof of $S \sqsubseteq S''$. Since $\tau_1 - \tau_2$ is defined, neither S_1 nor S_2 contain nodes labeled with $\underline{*}$, $\hat{*}$ or ?. Recall from the definition of $\tau_1 - \tau_2$ that $data(S_1)$ consists of the components of S_1 that have a $*$-labeled node. Let $D \in comp(S)$. Let complex C_1 be obtained from $\odot_1 \cup D$ by replacing each $*$ by a^ℓ. Let complex C_2 be obtained from $\odot_2$ by replacing each $*$ by b^ℓ. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 - C_2 : \tau''$. Since $D \in comp(S)$ we have $D \in data(S_1)$, or $D \notin data(S_1)$ and D does not have an isomorphic copy in $\odot_2$. In the first case, D itself appears as a component in $C_1 - C_2$, because C_1 and C_2 have different ℓ-cores. In the second case, there is, by definition of C_2, a component C in $C_1 - C_2$ with $stype(C) \equiv D$. Since, $C_1 - C_2 : \tau''$, we conclude in both cases that $D \in comp(S'')$.
2. Proof of $\odot'' \sqsubseteq \odot$. Let $D \in comp(S)$ and $D \notin comp(\odot)$. We show that $D \notin comp(\odot'')$. Let complex C_1 be obtained from $\odot_1$ by replacing all $*$-labeled nodes by a^ℓ. Complex C_1 has type τ_1. Let complex C_2 be obtained from $\odot_2 \cup (\odot_1 - T_1)$ by replacing all $*$-labeled nodes by a^ℓ. Recall that T_1 consists of all components in S_1 that have *no* isomorphic copy in S_2. Consequently, $\odot_1 - T_1$ consists of the components in $\odot_1$ having an isomorphic copy in S_2. As a result, C_2 has type τ_2 (note that by the nature of S_1 and S_2 (since $\tau_1 - \tau_2$ is defined), C_2 is always saturated). Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 - C_2 : \tau''$. Complex $C_1 - C_2$ consists solely of components of type $\odot_1 \cap T_1$, whence $C_1 - C_2$ does not contain a component of type D, because $D \notin comp(\odot)$ and $D \notin \odot_1 \cap T_1$. Thus, $D \notin comp(\odot'')$.
3. Proof of $\mathfrak{h}'' = true$ implies that τ is saturated. As $\mathfrak{h} = true$, τ is trivially saturated.

Hybridize We have $e = \texttt{hybridize}(x)$. Let $\Gamma(x) = \tau' = (S', \odot', \mathfrak{h}')$. Then $\tau = \texttt{hybridize}(\tau')$.

We first treat the case $\mathfrak{h}' = true$. Let C be a complex of type τ. We must show that C is also of type τ''. Since $\tau = \tau'$, C is of type τ'. Since $\mathfrak{h} = true$, C is saturated. Hence, $\texttt{hybridize}(C)$ equals C. Since $\Gamma \models e : \tau''$, $\texttt{hybridize}(C)$ is of type τ''. Hence C is of type τ'' as desired.

We now assume $\mathfrak{h}' = false$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.3.

1. Proof of $S \sqsubseteq S''$. Let $D \in comp(S)$. By definition of S, either (a) D is $\texttt{immob}(?)$ or (b) D is a component in $\texttt{hybridize}_t(X)$ for some weak type X, with $NC \sqsubseteq X \sqsubseteq S'$.
   (a) By the definition of the hybridization operation and the fact that $\texttt{immob}(?)$ is in $\texttt{hybridize}(\tau')$, we know that $\texttt{immob}(?)$ is part of τ'. Let C be a complex obtained from $\odot' \cup \texttt{immob}(?)$ by replacing all $*$-, $\hat{*}$-, $\underline{*}$-labeled nodes by a^ℓ, replacing *closed* ?-labeled nodes by $\overline{a}$ and replacing all *free* ?-labeled nodes by $\overline{b}$. Complex C has type τ'. We may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{hybridize}(C) : \tau''$. Since all ℓ-cores of C are equivalent to a^ℓ, and all free immobilized nodes are labeled with $\overline{b}$, there is a

free probe in $\texttt{hybridize}(C)$. Thus, $D \in \mathit{comp}(S'')$.

   (b) Let C be the complex obtained from X by replacing all $*$, $\hat{*}$, and $\underline{*}$-labeled nodes by a^ℓ and the ?-labeled nodes by $\overline{a}$. Moveover, if $\texttt{immob}(?) \in \odot'$ but $\texttt{immob}(?) \notin X$, then we add to C a component $\texttt{immob}(\overline{b})$. Then C has type τ', so we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. Now, $\texttt{hybridize}(C)$ has a component of weak type D. Since $\Gamma \models e : \tau''$, $\texttt{hybridize}(C) : \tau'''$, so $D \in \mathit{comp}(S'')$.

2. Proof of $\odot'' \sqsubseteq \odot$. Let $D \in \mathit{comp}(S)$ and $D \notin \mathit{comp}(\odot)$. We show that $D \notin \mathit{comp}(\odot'')$. As before, the argument is split into two cases: (a) $D = \texttt{immob}?$ or (b) $D \in \mathit{comp}(\texttt{hybridize}_t(X))$ for some $NC \sqsubseteq X \sqsubseteq S$.

   (a) By the fact that $D \equiv \texttt{immob}(?)$ and $D \notin \mathit{comp}(\odot)$, either $D \notin \mathit{comp}(\odot')$, or there is a component $E \in \mathit{comp}(S')$ with a free node labeled with $*$ or $\hat{*}$.

      i. Assume $D \notin \mathit{comp}(\odot')$. Let C be a complex such that $\mathit{stype}(C) \equiv \odot'$. Complex C has type τ', thus we may consider the input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{hybridize}(C) : \tau''$. By definition, there is no component in C of type $\texttt{immob}(?)$. Hence, $D \notin \mathit{comp}(\odot'')$.

      ii. Assume that $D \in \mathit{comp}(\odot')$ and there is a component $E \in \mathit{comp}(S')$ with a free node labeled with $*$ or $\hat{*}$. Let C be a complex with $\mathit{stype}(C) \equiv \odot' \cup E$ in which all ℓ-cores are of the form a^ℓ and all probes are labeled with $\overline{a}$. Complex C has type τ'. Hence, we may consider the assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. Complex C contains a free probe labeled $\overline{a}$ and an ℓ-core with a free node labeled a. Hence, $\texttt{hybridize}(C)$ does not contain a free probe. As $\Gamma \models e : \tau''$, $\texttt{hybridize}(C) : \tau''$. Thus, $D \notin \mathit{comp}(\odot'')$.

   (b) By the fact that $D \notin \mathit{comp}(\odot)$, and by the definition of $\odot$, we know that $D \notin \mathit{comp}(\texttt{hybridize}_t(NC))$ or $D \notin \mathit{comp}(\texttt{hybridize}_t(S'))$.

      i. $D \notin \mathit{comp}(\texttt{hybridize}_t(NC))$: Let C be a complex such that $\mathit{stype}(C) \equiv \odot'$, all ℓ-cores are of the form a^ℓ, all closed probes are labeled $\overline{a}$, and all free probes are labeled $\overline{b}$. By definition $NC \equiv \odot' - \texttt{immob}(?)$, $D \notin \mathit{comp}(\texttt{hybridize}_t(NC))$, and free probes cannot interact with ℓ-cores in C, whence there is no component in $\texttt{hybridize}(C)$ having type D. Complex C has type τ'. Hence, we may consider the assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{hybridize}(C) : \tau''$. Thus, $D \notin \mathit{comp}(\odot'')$.

      ii. $D \notin \mathit{comp}(\texttt{hybridize}_t(S'))$: Let C be a complex such that $\mathit{stype}(C) \equiv S'$, all ℓ-cores are of the form a^ℓ, and all probes are labeled $\overline{a}$. Complex C has type τ', indeed, recall that $\mathfrak{h}' = \mathit{false}$. Hence, we may consider the assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{hybridize}(C) : \tau''$. By our assumption, $D \notin \mathit{comp}(\texttt{hybridize}_t(S'))$, so there is no component in $\texttt{hybridize}(C)$ having type D. Hence, $D \notin \mathit{comp}(\odot'')$.

3. Proof of $\mathfrak{h}'' = \mathit{true}$ implies that τ is saturated. As $\mathfrak{h} = \mathit{true}$, τ is trivially saturated.

Ligate We have $e = \texttt{ligate}(x)$. Let $\Gamma(x) = \tau' = (S', \odot', \mathfrak{h}')$. Then $\tau = \texttt{ligate}(\tau')$.

To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.3.

1. Proof of $S \sqsubseteq S''$. Let $D \in \mathit{comp}(S)$. Let E be a component of S' such that $\texttt{ligate}(E) \equiv D$. Let C be a complex such that $\mathit{stype}(C) \equiv \odot' \cup E$. Complex C has type τ'. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. By definition, $\texttt{ligate}(C)$ contains a component of type D. As $\Gamma \models e : \tau''$, $\texttt{ligate}(C) : \tau''$, and so $D \in \mathit{comp}(S'')$.
2. Proof of $\odot'' \sqsubseteq \odot$. Let $D \in \mathit{comp}(S)$ and $D \notin \mathit{comp}(\odot)$. We show that $D \notin \mathit{comp}(\odot'')$. Let E be the set of components of S' such that for every component $F \in E$ we have $\texttt{ligate}(F) \equiv D$. By definition of $\odot$ and $D \notin \mathit{comp}(\odot)$, we know that $\forall F \in E : F \notin \mathit{comp}(\odot')$. Let C be a complex such that $\mathit{stype}(C) \equiv \odot'$. Complex C has type τ'. Hence, we may consider the assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. By construction of C, there is no component of type D in $\texttt{ligate}(C)$. As $\Gamma \models e : \tau''$, $\texttt{ligate}(C) : \tau''$. Thus, $D \notin \mathit{comp}(\odot'')$.
3. Proof of $\mathfrak{h}'' = \mathit{true}$ implies τ is saturated. Assume that $\mathfrak{h} = \mathit{false}$, otherwise the proof is trivial. If $\mathfrak{h} = \mathit{false}$, so is $\mathfrak{h}'$. The fact $\mathfrak{h}'' = \mathit{true}$ implies that τ'' is saturated. Let C be a complex such that $\mathit{stype}(C) \equiv S'$ with all ℓ-cores equal to a^ℓ and all probes labeled $\overline{a}$. Since $\mathfrak{h}' = \mathit{false}$, complex C has type τ'. As $\Gamma \models e : \tau''$, $\texttt{ligate}(C) : \tau''$. Because $\mathfrak{h}'' = \mathit{true}$, $\texttt{ligate}(C)$ must be saturated. The ligate operator only introduces new edges between nodes, in particular, no new nodes are introduced and no closed nodes are made open. Thus, $\texttt{ligate}(C)$ is saturated, implies C is saturated. Hence, S' is saturated, because all probes and ℓ-cores are labeled complementary. The ligate operator on types also does not introduce new nodes and it does not make closed nodes free. As a result, S is saturated, whence τ is saturated — a contradiction.

Split, Flush Similar to the case of Ligate.

Block, Block-From, Block-Except Similar to the case of Ligate, except that item 3. becomes trivial, because $\mathfrak{h}$ and $\mathfrak{h}'$ are always *true*.

Cleanup We have $e = \texttt{cleanup}(x)$. Let $\Gamma(x) = \tau' = (S', \odot', \mathfrak{h}')$. Then $\tau = \texttt{cleanup}(\tau')$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.3.

1. Proof of $S \sqsubseteq S''$. Let $s \in \mathit{comp}(S)$. By definition, component s is a strand and s *qualifies* for S, i.e., there is a component $D \in \omega(s)$ such that there is a positive integer solution x in the variable ℓ to the system of inequalities $\{n(s) + (\ell - 1)a(s) \geq n(t) + (\ell - 1)a(t) \mid t \in \big(\mathit{strands}(\odot') \cup \mathit{strands}(D)\big)\}$. Let C be a complex with dimension x such that $\mathit{stype}(C) \equiv \odot' \cup D$. As a result, any strand in C having type s is at least as long as all other positive strands in C, whence $\texttt{cleanup}(C)$ contains a component having type s. Complex C has type τ'. Hence, we may consider the assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{cleanup}(C) : \tau''$. Thus, $s \in \mathit{comp}(S'')$.
2. Proof of $\odot'' \sqsubseteq \odot$. Let $s \in \mathit{comp}(S)$ and $s \notin \mathit{comp}(\odot)$. We show that $s \notin \mathit{comp}(\odot'')$. A strand must fulfill two conditions to be mandatory in τ. First of all, there must be a component $D \in \omega(s)$ such that $D \in \odot'$. Secondly, it must qualify for mandatory.
   (a) If there is no component $D \in \omega(s)$ such that $D \in \odot'$, then let C be a complex such that $\mathit{stype}(C) \equiv \odot'$. There is no component in C

having a type from the set $\omega(s)$, whence there is no strand having type s in C, thus there is no strand having type s in $\texttt{cleanup}(C)$. Complex C has type τ'. Hence, we may consider the input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{cleanup}(C) : \tau''$. Thus, $s \notin \mathit{comp}(\odot'')$.

(b) There is a component $E \in \omega(s)$ such that $E \in \mathit{comp}(\odot')$. Strand s does not qualify for mandatory, whence there is a strand $t \in S$ for which the strict inequality $n(s) + (\ell - 1)a(s) < n(t) + (\ell - 1)a(t)$ has a positive integer solution in ℓ. Let x be the positive integer solution to this strict inequality. Let D be a component from $\omega(t)$. Let C be a complex with dimension x such that $\mathit{stype}(C) \equiv \odot' \cup D$. In complex C strands having type t are strictly longer than strands having type s, whence $\texttt{cleanup}(C)$ does not contain a strand having type s. Complex C has type τ'. Hence, we may consider the input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{cleanup}(C) : \tau''$. Thus, $s \notin \mathit{comp}(\odot'')$.

3. Proof of $\mathfrak{h}'' = \mathit{true}$ implies that τ is saturated. By definition $\mathfrak{h} = \mathit{true}$, thus τ is always saturated.

□

9. Relational Algebra Simulation.

In this section we show 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 the type checking algorithm.

9.1. Relational Algebra.

Let us first recall some basic definitions concerning the relational data model [1]. We assume a universe U of *data elements*. A *relation schema* R is a finite set of attributes. A *tuple* t over R is a mapping from R to U. The domain of t is called the *type* of t. A *relation* over R is a finite set of tuples over R. A relation schema R is the type of the relations over R, since all tuples in such relations have type R. A *database schema* is a mapping $\mathcal{D}$ on some finite set of *relation variables* that assigns a relation schema to each relation variable. A database schema is thus a type assignment for relation variables. An *instance* of $\mathcal{D}$ is a mapping I on the same set of relation variables that assigns to each relation variable x a relation over $\mathcal{D}(x)$.

The syntax of the relational algebra 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.

Let us recall the typing rules for the relational algebra [10, 27]. Let $\mathit{relvars}(e)$ be the set of relation variables in a relational algebra expression e. Let $\mathcal{D}$ be a database schema such that $\mathit{relvars}(e) \subseteq \mathit{dom}(\mathcal{D})$, i.e., every relation variable in e is assigned a type. Let R be a relation schema. The rules for when e has type R given $\mathcal{D}$, denoted

$\mathcal{D} \vdash e : R$, are the following:

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

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

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

The semantics of the well-typed relational algebra is well known; we repeat it here for the sake of completeness. Let $\mathcal{D} \vdash e : R$ and let I be an instance of $\mathcal{D}$. Then the evaluation of e on I, denoted by $[\![e]\!](I)$, yields a relation over R defined as follows:

$$\begin{aligned}
[\![x]\!](I) &= I(x) \\
[\![e_1 \cup e_2]\!](I) &= \{t \mid t \in [\![e_1]\!](I) \text{ or } t \in [\![e_2]\!](I)\} \\
[\![e_1 - e_2]\!](I) &= \{t \mid t \in [\![e_1]\!](I) \text{ and } t \notin [\![e_2]\!](I)\} \\
[\![e_1 \times e_2]\!](I) &= \{t_1 \cup t_2 \mid t_1 \in [\![e_1]\!](I) \text{ and } t_2 \in [\![e_2]\!](I)\} \\
[\![\sigma_{A=B}(e)]\!](I) &= \{t \mid t \in [\![e]\!](I) \text{ and } t(A) = t(B)\} \\
[\![\widehat{\pi}_A(e)]\!](I) &= \{t - \{(A, t(A))\} \mid t \in [\![e]\!](I)\} \\
[\![\rho_{A/B}(e)]\!](I) &= \{(t - \{(A, t(A))\}) \cup \{(B, t(A))\} \mid t \in [\![e]\!](I)\}
\end{aligned}$$

9.2. Simulation. We want now to represent relations by complexes. We will store data elements as 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. We may assume a fixed order on the attributes of R, say, $A, \ldots, B$. We denote the order by $\oplus$. If the order is clear from the context, it is left implicit. We then represent t by the following ℓ-complex: (using the constant notation of DNAQL)

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

Example 9.1. *Let $R = \{A, B, C\}$ be a relation schema with three attributes, and order $A \oplus B \oplus C$. Let $\ell = 3$ and let $\Lambda = \{0, 1\}$. Consider the tuple t over R with $t(A) = 000$, $t(B) = 010$, and $t(C) = 111$. Then*

$$\mathit{complex}(t) = \#_2 A \#_3 000 \#_4 \#_2 B \#_3 010 \#_4 \#_2 C \#_3 111 \#_4 \ .$$

A relation r of dimension ℓ is then represented by the ℓ-complex $\bigcup\{\mathit{complex}(t) \mid t \in r\}$ which we denote by $\mathit{complex}(r)$. Under order $\oplus$, this complex has type:

$$\tau_R^{\oplus} = (\#_2 A \#_3 * \#_4 \ldots \#_2 B \#_3 * \#_4, \emptyset, \mathit{true}) \ .$$

Indeed, the type has no mandatory components, as a relation may be empty. If the order is clear from the context, we simply write τ_R to denote the type of a complex representing a relation over relation schema R. A substrand of the form $\#_2 A \#_3 * \#_4$ consists of an attribute and a value, whence it is called an *attribute-value* *block*. Moreover, a database instance I over database schema $\mathcal{D}$ can be represented by the complex assignment $complex(I)$ that maps each relation variable x (used as a complex variable) to $complex(I(x))$. The type assignment corresponding to a database instance I of dimension ℓ, denoted $\Gamma_{\mathcal{D}}$, maps each relation variable x to the type corresponding to its relation schema R, i.e., τ_R.

THEOREM 9.1. *Let e be an arbitrary well-typed relational algebra expression over database schema $\mathcal{D}$, with output relation schema R, i.e., $\mathcal{D} : e \vdash R$. Then e can be translated into a DNAQL expression e^{DNA}, such that the following holds:*

1. *e^{DNA} is well-typed, specifically, $\Gamma_{\mathcal{D}} : e^{DNA} \vdash \tau_R$; and*
2. *e^{DNA} simulates e uniformly over all dimensions ℓ, i.e., for each natural number ℓ and for any ℓ-dimensional database instance I over $\mathcal{D}$:*

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

The proof is by induction on the structure of expression e, i.e., a simulating DNAQL expression is computed for each RA operator. For each construction, we show that the simulating DNAQL expression is well-typed and has the desired output type.

9.2.1. Abbreviations. For the proof we introduce a few useful abbreviations.

Blockfromto. For $a, b \in \Sigma$, we use $blockfromto(x, a, b)$ to abbreviate

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

Remember that the blockfrom operator operates in the inverse direction of a strand.

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 as $connect(x)$ with x a complex variable:

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

Circularization. Let x be a complex variable and let A and B be attributes. We use $circularize(x, A, B)$ to abbreviate

$$\begin{array}{l} \texttt{let } f_2 := \texttt{hybridize}(blockfromto(x, B, A) \cup \texttt{immob}(\overline{\#_3}))\texttt{ in} \\ \texttt{let } f_1 := connect(f_2 \cup \overline{\#_2\#_4})\texttt{ in} \\ \texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(f_1, A), \#_3)) \end{array}$$

Schemas & Types. Let R be a relation schema, then we call S_R, the weak type of $\tau_R = (S_R, \odot_R, \mathfrak{h}_R)$, a *relation-schema-type*. A *pseudo-relation-schema-type* of a relation schema R, resembles the relation-schema-type of R, except that some additional tags outside $\{\#_2, \#_3, \#_4\}$ may be present between attribute-value blocks, furthermore, no additional tags are present at the beginning and end of the strand. More formally, a pseudo-relation-schema-type is of the form $\#_2 A_1 \#_3 * \#_4 S_1 \#_2 A_2 \#_3 * \#_4 S_2 \ldots S_{k-1} \#_2 A_k \#_3 * \#_4$, where $A_1, \ldots, A_k$ are attributes and $S_1, \ldots, S_{k-1}$ are (possibly empty) sequences of tags, different from $\#_2$, $\#_3$, and $\#_4$. Hence, a relation-schema-type is a special case of a pseudo-relation-schema-type, where the sequences of additional nodes are all empty.

The *circularization*, or circular version, of a linear strand s is isomorphic to s, except that the last and first node of s form a directed edge.

LEMMA 9.2. *Let S be a pseudo-relation-schema-type with k attributes denoted A_1 to A_k. In the following we write A_1 as A and A_k as B. Let S_c be the circularization of S. Let $\tau = (S, \odot, \mathfrak{h})$, and let τ_c be a strong type $(S_c, \odot_c, true)$. Let $\odot_c \equiv$ `empty` if $\odot \equiv$ `empty`, otherwise $\odot_c \equiv S_c$. Let Γ be a type assignment such that $\Gamma(x) = \tau$. Then $\Gamma : circularize(x, A, B) \vdash \tau_c$.*

Thus, $circularize(x, A, B)$ will equal the complex obtained from x by circularizing every strand [22, 4].

Proof. We call the positive linear strand of type τ based on weak type S, strand s. Next, we derive the output type of $circularize(x, A, B)$ under type assignment Γ.

1. Γ : $blockfromto(x, B, A) \vdash \tau_1$. The weak type of τ is a pseudo-relation-schema-type, hence, type τ is saturated regardless of the value of $\mathfrak{h}$. Type $\tau_1 = ((V_1, L_1, \lambda_1, \mu_1, \iota_1, \beta_1), \odot_1, \mathfrak{h}_1)$ resembles type τ, in the sense that it has a strand t_1 that is isomorphic to strand s, except that all nodes from the node labeled A up to and including the node labeled B are blocked, i.e., are members of β_1. The bit $\mathfrak{h}_1$ is set to *true*, and $\odot_1$ is empty. Note that the node labeled $\#_3$ directly following the node labeled B is the only free $\#_3$-labeled node in this type. Also note that there is only one free node labeled $\#_2$ resp. $\#_4$, at the beginning resp. end of the strand.
2. Γ : $blockfromto(x, B, A) \cup \texttt{immob}(\overline{\#_3}) \vdash \tau_2$. Type $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ resembles type τ_1. A new node n is introduced, labeled $\overline{\#_3}$. Node n is in $\odot_2$, and is immobilized in both S_2 and $\odot_2$. The strand in S_2, isomorphic to strand t_1, is called t_2. Strand t_2 is non-mandatory and largely blocked (with a single free $\#_3$), whereas probe n is mandatory and free. The $\mathfrak{h}$-bit of τ_2 is set to *false*, because n and t_2 have nodes that can interact.
3. Γ : $\texttt{hybridize}(blockfromto(x, B, A) \cup \texttt{immob}(\overline{\#_3})) \vdash \tau_3$. The set of necessary components consists of probe n, thus there are two sets of components on which we will apply $\texttt{hybridize}_t$, i.e., just n and the combination of strand t_2 and n. The first set results in an isomorphic copy of n. The second set results in a new component C_3 consisting of a probe isomorphic to n and an isomorphic copy of strand t_2, connected by a matching between the the free nodes labeled $\#_3$ and $\overline{\#_3}$. As neither of the components occurs both in $\texttt{hybridize}_t(NC)$ and $\texttt{hybridize}_t(S_2)$, neither of the components is mandatory. The $\mathfrak{h}$-bit of τ_3 is set to *true*.
   From this point on, we regard type assignment $\Gamma_2 = \Gamma \cup \{(f_2, \tau_3)\}$.
4. Γ_2 : $f_2 \cup \overline{\#_2\#_4} \vdash \tau_4$. Type τ_4 resembles type τ_3, except that a mandatory sticker labeled $\overline{\#_2\#_4}$ is present. Call this sticker u. The $\mathfrak{h}$-bit of τ_4 is set to *false*, as the the sticker and the strand are mutually interacting.
5. Γ_2 : $connect(f_2 \cup \overline{\#_2\#_4}) \vdash \tau_5$. The $\mathfrak{h}$-bit of τ_4 is set to *false*, thus hybridization takes place. The sticker u is the only mandatory component. Thus, there are four different sets to apply $hybridize_t$ on:
   (a) $X = NC = \{u\}$: as there is only one sticker, the result of hybridization is isomorphic to u;
   (b) $X = \{u, n\}$: the sticker u and the probe n have no complementary labels, thus the output is isomorphic to the input;
   (c) $X = \{u, C_3\}$: hybridizing the sticker u and the immobilized component C_3 results in two new components. The first component consists of an isomorphic copy of C_3 with a single isomorphic copy of sticker u. The sticker connects the end and beginning of the positive strand in the immobilized component. As a result, the strand is bent into a circle.

However, there is still a gap between the beginning and end of the strand. The second component is based on an isomorphic copy of C_3 and two isomorphic copies of the sticker u. One copy of u will match to the beginning of the copy of C_3. The other copy will match to the end of the copy of C_3. This component has maximal matching because the only free nodes are labeled $*, \overline{\#_2}$, and $\overline{\#_4}$. Clearly, these free nodes have no complementary labeled nodes; and

(d) $X = \{u, n, C_3\}$: the result is isomorphic to the previous case, except that a probe isomorphic to n is also present.

Let $\Gamma_2 : \texttt{hybridize}(f_2 \cup \overline{\#_2\#_4}) \vdash \tau_h$. The weak type of τ_h, denoted with S_h, consists of four components:

(a) Component C_1^h is isomorphic to sticker u;
(b) Component C_2^h is isomorphic to probe n;
(c) Component C_3^h is isomorphic to the component formed by one copy of C_3 and one copy of u; and
(d) Component C_4^h is isomorphic to the component formed by one copy of C_3 and two copies of u.

The mandatory weak type of type τ_h is equivalent to the empty complex, because none of the components in $\texttt{hybridize}_t(NC)$ is isomorphic to a component in $\texttt{hybridize}_t(S_4)$, where S_4 is the weak type of type τ_4. The $\mathfrak{h}$-bit of τ_5 is set to *true*, in accordance with the definition of the hybridization operation on complex types.

In weak type S_h component C_3^h has one gap. Recall that component C_3^h is bent into a circle, but is not circular because it has a gap. The ligate operation fills this gap, creating a new component $C_3^{h'}$ which is circular, i.e., removing the isomorphic copy of the sticker u, would result in a circular strand.

To conclude, the weak type of type τ_5 consists of isomorphic copies of s and the circularization of s, denoted c. There are no blockings, matchings nor immobilizations in the weak type of type τ_5. The mandatory weak type of type τ_5 is isomorphic to the empty complex type. The $\mathfrak{h}$-bit of type τ_5 is set to *true*.

From this point on, we regard type assignment $\Gamma_1 = \Gamma_2 \cup \{(f_1, \tau_5)\}$.

6. $\Gamma_1 : \texttt{blockfrom}(f_1, A) \vdash \tau_6$. Let us examine both strands of type τ_5 separately. In the linear strand s, there is one σ-blocking range, consisting of the first two nodes, because the second one is labeled with A and the first node is the beginning of the strand. In the circular strand c, there is one σ-blocking range, consisting of all nodes in c. Consequently, type τ_6 consists of two strands, denoted c' and s', which are isomorphic copies of respectively strand c and s, except that all nodes of c' are blocked and the first two nodes of s' are blocked. The $\mathfrak{h}$-bit of type τ_6 is set to *true*, because the $\mathfrak{h}$-bit of type τ_5 is *true*.
7. $\Gamma_1 : \texttt{split}(\texttt{blockfrom}(f_1, A), \#_3) \vdash \tau_7$. The split point identified by $\#_3$ splits only at free $\#_3$. The linear strand s' contains k free nodes labeled $\#_3$, because only the first two nodes are blocked and there are k attributes in s', with a $\#_3$ labeled node following each attribute. The circular strand c' is completely blocked and thus has no free nodes labeled $\#_3$.

The weak type of type τ_7 thus consists of $k + 1$ linear strands, obtained by splitting the linear strand s' at each of the k free $\#_3$-labeled nodes. Note that each of the linear strands consists of a maximum of three nodes plus the

length of the longest S_i for $i \in \{1, \ldots, k-1\}$, of which at most one is labeled $*$. Furthermore, the weak type of type τ_7 contains circular strand c'.
The mandatory weak type of type τ_7 is equivalent to the empty complex type. The $\mathfrak{h}$-bit of type τ_7 is set to *true* because the $\mathfrak{h}$-bit of type τ_6 is *true*.

8. Γ_1 : *circularize*$(x, A, B) \vdash \tau_8$. Let m be a linear strand in the weak type of τ_7. From the previous item, it is known that $n(m) \leq 5 + \max_{i\in\{1,\ldots,k-1\}} |S_i|$ and $a(m) \leq 1$. In contrast, $5k + \Sigma_{i=1}^{k-1}|S_i| = n(c')$ and $a(c') = k$. Thus

$$\begin{aligned} n(m) + (\ell-1)a(m) &\leq 5 + \max_{i\in\{1,\ldots,k-1\}} |S_i| + (\ell - 1) \\ &< 5k + \Sigma_{i=1}^{k-1}|S_i| + (\ell-1)k \\ &= n(c') + (\ell-1)a(c') \end{aligned}$$

Consequently, linear strand m will never qualify for type τ_c.
The only strand that qualifies is c'. The cleanup operation, furthermore, removes all matchings, blockings and immobilizations. As a result, the weak type of type τ_8 consists of strand c. The mandatory weak type of type τ_8 is equivalent to the empty complex type. The $\mathfrak{h}$-bit of type τ_8 is set to *true*.

It is clear that $\tau_8 \equiv \tau_c = (S_c, \odot_c, \mathit{true})$, with $\odot_c \equiv$ `empty`. This proves that Γ : *circularize*$(x, A, B) \vdash \tau_c$ if $\odot \equiv$ `empty`, i.e., *circularize*(x, A, B) has the desired type.

Next, we prove that Γ : *circularize*$(x, A, B) \vdash \tau_c$ with $\odot_c \equiv S_c$ if $\odot \equiv S$.

1. Γ : *blockfromto*$(x, B, A) \vdash \tau_1$. Type $\tau_1 = (S_1 = (V_1, L_1, \lambda_1, \mu_1, \iota_1, \beta_1), \odot_1, \mathfrak{h}_1)$ resembles type τ, in the sense that it has a strand t_1 that is isomorphic to strand s, except that all nodes from the node labeled A up to and including the node labeled B are blocked, i.e., are members of β_1. The bit $\mathfrak{h}_1$ is set to *true*, $\odot_1$ is equivalent to weak type S_1. Note that the node labeled $\#_3$ directly following the node labeled B is the only free $\#_3$-labeled node in this type. Also note that there is only one free node labeled $\#_2$ resp. $\#_4$, at the beginning resp. end of the strand.
2. Γ : *blockfromto*$(x, B, A) \cup$ `immob`$(\overline{\#_3}) \vdash \tau_2$. Type $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ resembles type τ_1. A new node n is introduced, labeled $\overline{\#_3}$. Node n is in $\odot_2$, and in ι of both S_2 and $\odot_2$. The $\mathfrak{h}$-bit of τ_2 is set to *false*. The strand in S_2, isomorphic to strand t_1, is called t_2. Strand t_2 is mandatory and largely blocked (with a single free $\#_3$). Probe n is mandatory and free.
3. Γ : `hybridize`$($*blockfromto*$(x, B, A) \cup$ `immob`$(\overline{\#_3})) \vdash \tau_3$. The set of necessary components consists of strand s and probe n, thus there is one set of components on which we will apply `hybridize`$_t$, i.e., strand s and probe n. This results in a new component C_3 consisting of a probe isomorphic to n and an isomorphic copy of strand t_2, connected by a matching between the free nodes labeled $\#_3$ and $\overline{\#_3}$. Component C_3 is mandatory. The $\mathfrak{h}$-bit of τ_3 is set to *true*.
From this point on, we regard type assignment $\Gamma_2 = \Gamma \cup \{(f_2, \tau_3)\}$.
4. Γ_2 : $f_2 \cup \overline{\#_2\#_4} \vdash \tau_4$. Type τ_4 resembles type τ_3, except that a mandatory sticker labeled $\overline{\#_2\#_4}$ is present. Call this sticker u. The $\mathfrak{h}$-bit of τ_4 is set to *false*, as sticker u and the strand of component C_3 are mutually interacting.
5. Γ_2 : *connect*$(f_2 \cup \overline{\#_2\#_4}) \vdash \tau_5$. The $\mathfrak{h}$-bit of τ_4 is set to *false*, thus hybridization takes place. Both component C_3 and the sticker u are mandatory and necessary components. Thus, there is one set to apply *hybridize*$_t$ on, i.e., $\{u, C_3\}$. Hybridizing the sticker u and the immobilized component C_3 results

in two new components. The first component, denoted with C_1^h consists of an isomorphic copy of C_3 with a single isomorphic copy of sticker u. The sticker connects the end and beginning of the positive strand in the immobilized component. As a result, the strand is bent into a circle. However, there is still a gap between the beginning and end of the strand. The second component, denoted with C_2^h, is based on an isomorphic copy of C_3 and two isomorphic copies of the sticker u. One copy of u will match to the beginning of the copy of C_3. The other copy will match to the end of the copy of C_3. This component has maximal matching because the only free nodes are labeled $*, \overline{\#_2}$, and $\overline{\#_4}$. Clearly, these free nodes have no complementary labeled nodes. Both components C_1^h and C_2^h are mandatory.
Recall that component C_1^h is bent into a circle, but is not circular because it has a gap. The ligate operation fills this gap, creating a new component $C_1^{h'}$ which is circular, i.e., removing the isomorphic copy of the sticker u, would result in a circular strand.
Finally, the weak type after the cleanup operation consists of the circular strand, denoted c, isomorphic to circularization of strand s.
To conclude, the weak type of type τ_5 consists of components c and s. There are no blockings, matchings nor immobilizations in the weak type of type τ_5. The mandatory weak type of type τ_5 is isomorphic to the weak type. The $\mathfrak{h}$-bit of type τ_5 is set to *true*.
From this point on, we regard type assignment $\Gamma_1 = \Gamma_2 \cup \{(f_1, \tau_5)\}$.

6. Γ_1 : $\texttt{blockfrom}(f_1, A) \vdash \tau_6$. Let us examine both strands of type τ_5 separately. In the linear strand s, there is one σ-blocking range, consisting of the first two nodes, because the second one is labeled with A and the first node is the beginning of the strand. In the circular strand c, there is one σ-blocking range, consisting of all nodes in c. Consequently, type τ_6 consists of two strands, denoted c' and s', which are isomorphic copies of respectively strand c and s, except that all nodes of c' are blocked and the first two nodes of s' are blocked. The $\mathfrak{h}$-bit of type τ_6 is set to *true*, because the $\mathfrak{h}$-bit of type τ_5 is *true*.
7. Γ_1 : $\texttt{split}(\texttt{blockfrom}(f_1, A), \#_3) \vdash \tau_7$. The split point identified by $\#_3$ splits only at free $\#_3$. The linear strand s' contains k free nodes labeled $\#_3$, because only the first two nodes are blocked and there are k attributes in s', with a $\#_3$ labeled node following each attribute. The circular strand c' is completely blocked and thus has no free nodes labeled $\#_3$.
The weak type of type τ_7 thus consists of $k+1$ linear strands, obtained by splitting the linear strand s' at each of the k free $\#_3$-labeled nodes. Note that each of the linear strands consists of a maximum of three nodes plus the length of the longest S_i for $i \in \{1, \ldots, k-1\}$, of which at most one is labeled $*$. Furthermore, the weak type of type τ_7 contains circular strand c'.
The mandatory weak type of type τ_7 is equivalent the weak type. The $\mathfrak{h}$-bit of type τ_7 is set to *true* because the $\mathfrak{h}$-bit of type τ_6 is *true*.
8. Γ_1 : *circularize*$(x, A, B) \vdash \tau_8$. Let m be a linear strand in the weak type of type. From the previous item, it is known that $n(m) \leq 5 + \max_{i \in \{1,\ldots,k-1\}} |S_i|$

and $a(m) \leq 1$. In contrast, $5k + \Sigma_{i=1}^{k-1}|S_i| = n(c')$ and $a(c') = k$. Thus

$$\begin{aligned} n(m) + (\ell - 1)a(m) &\leq 5 + \max_{i \in \{1,\dots,k-1\}} |S_i| + (\ell - 1) \\ &< 5k + \Sigma_{i=1}^{k-1}|S_i| + (\ell - 1)k \\ &= n(c') + (\ell - 1)a(c') \end{aligned}$$

Consequently, linear strand m will never qualify for type τ_c.

The only strand that qualifies is c. The cleanup operation, furthermore, removes all matchings, blockings and immobilizations.

Strand c qualifies for mandatory. Indeed, let m be a linear strand in the weak type of type τ_7, then there may not be a positive integer solution to

$$n(c) + (\ell - 1)a(c) < n(m) + (\ell - 1)a(m)$$

We know that $n(c) + (\ell - 1)a(c) > n(m) + (\ell - 1)a(m)$. Hence, c qualifies for mandatory.

As a result, the weak type of type τ_8 consists of strand c. The mandatory weak type of type τ_8 consists of strand c. The $\mathfrak{h}$-bit of type τ_8 is set to *true*.

It is clear that $\tau_8 \equiv \tau_c = (S_c, \odot_c, \mathit{true})$, with $\odot_c \equiv S_c$. This proves that Γ : *circularize*$(x, A, B) \vdash \tau_c$ if $\odot \equiv S$, i.e., *circularize*(x, A, B) has the desired type. □

Inserting into a Circle. Let x be a complex variable and let A and B be attributes. We use *insertcirc*(x, A, B, s) to abbreviate

```
let $y_1 :=$ split($\mathit{blockfromto}(x, A, B), \#_4$) in
let $y_2 :=$ hybridize(hybridize($y \cup$ immob($\overline{\#_3}$)) $\cup\ \overline{\#_4\sigma_1} \cup s) \cup \overline{\sigma_2\#_2}$ in
cleanup(split(blockfrom($\mathit{connect}(y_2), B$), $\#_3$))
```

LEMMA 9.3. *Let S be the circularization of a pseudo-relation-schema-type with k attributes denoted $A_1, \dots, A_k$. Let $1 \leq i \leq k$. We denote attribute A_i with A and the attribute following attribute A on the circular strand of S is denoted B. Let S_i be the empty sequence of additional tags between attributes A and B. Let s be a linear strand of length at least two such that no node is labeled with $\#_2$ or $\#_4$. Denote with σ_1, respectively σ_2, the first, respectively last, symbol, of s. We assume that the symbols σ_1 and σ_2 are unique with respect to the circular strand and strand s. Let weak type S' resemble S, except that sequence $S_i = s$.*

Let $\tau = (S, S, \mathfrak{h})$ be a type. Let $\tau' = (S', S', \mathit{true})$ be a type. Let Γ be a type assignment such that $\Gamma(x) = \tau$. Then Γ : insertcirc$(x, A, B, s) \vdash \tau'$.

Proof. We call the positive circular strand of type τ based on weak type S, strand c. Next, we derive the output type of *insertcirc*(x, A, B, s) under type assignment Γ.

1. Γ : *blockfromto*$(x, A, B) \vdash \tau_1$. Because type τ is a pseudo-relation-schema-type, we know that it is saturated regardless of the value of $\mathfrak{h}$. Type τ_1 resembles type τ, in the sense that it has a strand t that is isomorphic to strand c, except that for the four nodes labeled $\#_3$, $*$, $\#_4$ and $\#_2$ directly following the node labeled A, are the only non-blocked nodes. Strand t is mandatory and the $\mathfrak{h}$-bit of type τ_1 is set to *true*. Consequently, the only free node labeled $\#_4$ in strand t is the last node of the attribute-value block of A.
2. Γ : $\texttt{split}(\mathit{blockfromto}(x, A, B), \#_4) \vdash \tau_{y_1}$. Type τ_{y_1} consists of a single, linear, mandatory strand t_1. Strand t_1 resembles strand t except it is linear. The first attribute of t_2 is B and A is the last attribute. All the nodes,

beginning from the node labeled B up to and including the node labeled A, are blocked. The $\mathfrak{h}$-bit is set to *true*.

From this point on, we regard type assignment $\Gamma_1 = \Gamma \cup \{(y_1, \tau_{y_1})\}$.

3. $\Gamma_1 : y \cup \texttt{immob}(\overline{\#_3}) \vdash \tau_2$. Type τ_2 consists of two mandatory components. The first component, denoted t_2, is a strand isomorphic to t_1. The second component consists of a single node n, which is immobilized and is labeled $\overline{\#_3}$. The $\mathfrak{h}_2$-bit is set to *false*, because node n can match with the free node labeled $\#_3$ in strand t_2, following directly behind the node labeled A.
4. $\Gamma_1 : \texttt{hybridize}(y \cup \texttt{immob}(\overline{\#_3})) \vdash \tau_3$. Because $\mathfrak{h}_2 = \textit{false}$, hybridization takes place. Type τ_2 consists of two mandatory components. One of the mandatory components is a probe, but the probe is not labeled with a negative atomic value symbol, thus both components are necessary components. Hence, $\texttt{hybridize}_t$ is applied on just one set X, consisting of the strand t_2 and the probe n. Strand t_2 has one free node labeled $\#_3$ and probe n is labeled $\overline{\#_3}$. The result is a component C_1 consisting of a strand isomorphic to t_2, a probe isomorphic to n and a matching between the probe and the only free node labeled $\#_3$ in the strand. Component C_1 is mandatory, as all components in X are necessary and mandatory in τ_2. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.
5. $\Gamma_1 : \texttt{hybridize}(y \cup \texttt{immob}(\overline{\#_3})) \cup \overline{\#_4\sigma_1} \cup s \vdash \tau_4$. Type τ_4 consists of three components:
   (a) component C_1;
   (b) sticker s_1, labeled $\overline{\#_4\sigma_1}$; and
   (c) strand s

   All three components are mandatory. The $\mathfrak{h}$-bit of type τ_4 is set to *false*, as the sticker can match with component C_1 and with strand s.
6. $\Gamma_1 : \texttt{hybridize}(\texttt{hybridize}(y \cup \texttt{immob}(\overline{\#_3})) \cup \overline{\#_4\sigma_1} \cup s) \vdash \tau_5$. We know that the $\mathfrak{h}$-bit of τ_4 equals *false*, thus hybridization takes place. As all components are mandatory and there are no (free) probes, all components are necessary components. Hence, there is one set $X = \{C_1, s_1, s\}$. Note that there is only one free node in C_1 labeled $\#_4$, namely, the last node of the linear strand in C_1 isomorphic to t_2. Thus, strand s can attach to the end of the linear strand, in component C_1, through sticker s_1, because symbol σ_1 is unique in strand t_2 and s. The hybridization binds the three components of τ_4 into one new component, called C_2. Component C_2 is immobilized. The $\mathfrak{h}$-bit of type τ_5 is set to *true*.
7. $\Gamma_1 : \texttt{hybridize}(\texttt{hybridize}(y \cup \texttt{immob}(\overline{\#_3})) \cup \overline{\#_4\sigma_1} \cup s) \cup \overline{\sigma_2\#_2} \vdash \tau_{y_2}$. Type τ_{y_2} consists of the mandatory component C_2 and a mandatory sticker s_2, labeled $\overline{\sigma_2\#_2}$. The $\mathfrak{h}$-bit of type τ_{y_2} is set to *false*, because sticker s_2 can interact with the first node of the linear strand in component C_2, isomorphic to t_2. Moreover, sticker s_2 can interact with the last node of strand isomorphic to s in component C_2.

   From this point on, we use the type assignment $\Gamma_2 = \Gamma_1 \cup \{(y_2, \tau_{y_2})\}$
8. $\Gamma_2 : \textit{connect}(y_2) \vdash \tau_6$. The connect abbreviation consists of (a) a hybridization, (b) a ligation and (c) a cleanup.
   (a) Type τ_{y_2} consists of the immobilized, mandatory component C_2 and mandatory sticker s_2, labeled $\overline{\sigma_2\#_2}$. Component C_2 consists of an immobilized, largely blocked, linear strand isomorphic to t_2 and the strand s, attached to the former by means of sticker s_1. Component C_2 thus

starts with the only free node labeled $\#_2$. Strand s ends with the only free node labeled σ_2. As $\mathfrak{h}_6$ is set to *false*, hybridization takes place. Because both components in τ_{y_2} are mandatory and do not contain a node labeled ?, both are necessary components. Consequently, there is one set X to hybridize, consisting of both components. The hybridization results in two components.

i. The first component, call it D_1, consists of one isomorphic copy of C_2 and one isomorphic copy of s_2. The sticker binds to the front and end of the immobilized component, bending it into a circle, yet there are still gaps.
ii. The second component, call it D_2, consists of one isomorphic copy of C_2 and two isomorphic copies of s_2. One of the sticker binds to the front of the isomorphic copy of C_2. The other sticker binds to the end of the isomorphic copy of C_2. More formally, the second sticker binds with the last node of the strand isomorphic to s.

Because the set of necessary components is equivalent to the weak type of type τ_{y_2}, both D_1 and D_2 are mandatory in the output of the hybridize operation.

(b) Component D_1 has two gaps: one between the end of the strand isomorphic to t_2 and the beginning of the strand isomorphic to strand s, and one between the end of the strand isomorphic to strand s and the beginning of the strand isomorphic to t_2. The ligate operation closes both gaps, creating a new *circular* strand, denoted t_3, which is the concatenation of strands isomorphic to t_1 and s.
Component D_2 also has one gap between the end of the strand isomorphic to strand s and the beginning of the strand isomorphic to t_2. The ligate operation closes this gap, creating a new *linear* strand, denoted t_4, which is the concatenation of strands isomorphic to t_1 and s.

(c) The cleanup operator removes all blockings, matchings and immobilizations and retains only the longest strands. In components D_1 and D_2 there are two positive strands: t_3 and t_4. Both strands are concatenations of strands t_1 and s and thus have the same length, regardless of the value of ℓ. Consequently, both qualify and both qualify for mandatory.

Type τ_6 consists of the strands t_3 and t_4, both are mandatory, and $\mathfrak{h}_6$, the $\mathfrak{h}$-bit of type τ_6, is set to *true*.
Strand t_3 equals the insertion of s between the attribute-value blocks of attributes A and B. It remains to get rid of the linear strand t_4.

9. Γ_2 : $\texttt{blockfrom}(\mathit{connect}(y_2), B) \vdash \tau_7$. Type τ_6 consists of two strands: t_3 is a circular strand and t_4 is a linear strand. In strand t_3 all nodes will be blocked when starting to block from the node labeled B. In strand t_4, the node labeled B is the second node of the strand. Hence, only two nodes will become blocked in the linear strand. Type τ_7 thus consists of two strands: a circular strand resembling t_3, except that all nodes are blocked, and a linear strand resembling t_4, except that the first two nodes are blocked. Both strands are mandatory and the $\mathfrak{h}$-bit of type τ_7 is set to *true*.
10. Γ_2 : $\texttt{split}(\texttt{blockfrom}(\mathit{connect}(y_2), B), \#_3) \vdash \tau_8$. Type τ_7 consists of two mandatory strands: one is circular, the other linear. The circular strand has no free nodes, in particular there is no free node labeled $\#_3$, whence no splitting can be performed on the circular strand. The linear strand, on

the other hand, has at least two free nodes labeled $\#_3$ (there is one located directly after the node labeled A and another directly after the node labeled B). The linear strand is thus split in at least three parts. All components are mandatory and the $\mathfrak{h}$-bit of type τ_8 is set to *true*.

11. $\Gamma_2 : \texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(\mathit{connect}(y_2), A), \#_3)) \vdash \tau'$. All components in type τ_8 are mandatory. Strand t_3 has k attributes, $k-1$ possibly empty sequences of additional tags, and strand s. Any part of strand t_4 has at most $5 + |s| + \max_{i\in\{1,\ldots,k-1\}} |S_i|$ nodes, of which at most one is labeled with $*$. The length of t_3 is at least $5k + (\ell-1)k + |s| + \Sigma_{i=0}^{k-1}|S_i|$. As a result, the cleanup operator retains only the circular strand t_3, with strand s inserted between A and B. Strand t_3 is mandatory and the $\mathfrak{h}$-bit is set to *true*. Thus, the output type equals τ'.

□

Removing from a Circle. Let x be a complex variable and A and B attributes. We use *removeBetweenCirc*(x, A, B) to abbreviate

$$\texttt{cleanup}(\texttt{split}(\texttt{split}(\mathit{blockfromto}(x, A, B), \#_4), \#_2))$$

LEMMA 9.4. *Let S be a circularization of a pseudo-relation-schema-type with k attributes, denoted $A_1, \ldots, A_k$. We denote with c the circular strand in S. Let A and B be two attributes in S. Let $c_{A\to B}$ be the substrand of c situated between the end of the attribute-value block of A and the beginning of the attribute-value block of B. Let $c_{A\leftarrow B}$ be the substrand of c starting from the attribute-value block of B up to and including the attribute-value block of A. Let $\tau = (S, S, \mathit{true})$ be a type. Let S_r be a weak type with a single component isomorphic to $c_{A\leftarrow B}$. Let $\tau_r = (S_r, S_r, \mathit{true})$ be a type. Let Γ be a type assignment such that $\Gamma(x) = \tau$. If $n(c_{A\to B}) < n(c_{A\leftarrow B})$ and $a(c_{A\to B}) < a(c_{A\leftarrow B})$, then Γ : removeBetweenCirc$(x, A, B) \vdash \tau_r$.*

Proof. We derive the output type of *removeBetweenCirc*(x, A, B) under type assignment Γ.

1. $\Gamma : \mathit{blockfromto}(x, A, B) \vdash \tau_1$. Type τ is saturated. Type τ_1 consists of a strand t_1 that is isomorphic to t, except that all nodes that are *not* between attribute A and B are blocked. As a result, at least one $\#_4$-labeled node is free (the last one of the attribute-value block of attribute A) and at least one $\#_2$-labeled node is free (the first one of the attribute-value block of attribute B). Strand t_1 is mandatory and the $\mathfrak{h}$-bit of type τ_1 is set to *true*.
2. $\Gamma : \texttt{split}(\texttt{split}(\mathit{blockfromto}(x, A, B), \#_4), \#_2) \vdash \tau_2$. The circular strand t_1 is cut at least behind the attribute-value block of attribute A and before the attribute-value block of attribute B. Type τ_2 contains at least two strands. The first strand, t_3 is isomorphic to $c_{A\leftarrow B}$, except all nodes are blocked except for the first node (labeled $\#_2$, just in front of the node labeled B) and the last three nodes (labeled $\#_3$, $*$ and $\#_4$, just after the node labeled A). The other strands, resulting from the double split, bundled in a set called T, are at most as long as $c_{A\to B}$. All strands are mandatory. The $\mathfrak{h}$-bit of type τ_2 is set to *true*.
3. $\Gamma : \texttt{cleanup}(\texttt{split}(\texttt{split}(\mathit{blockfromto}(x, A, B), \#_4), \#_2)) \vdash \tau_r$. Next, we show that strand $c_{A\leftarrow B}$ is longer than any strand in T, for any value of ℓ. Let u be a strand in set T, then the length of u is at most equal to the length of $c_{A\to B}$, which is $n(c_{A\to B}) + (\ell-1)a(c_{A\to B})$. The length of $c_{A\leftarrow B}$ is $n(c_{A\leftarrow B}) + (\ell-1)a(c_{A\leftarrow B})$. We need to show that $n(c_{A\to B}) + (\ell-1)a(c_{A\to B}) < n(c_{A\leftarrow B}) + (\ell-1)a(c_{A\leftarrow B})$, or, $0 < n(c_{A\leftarrow B}) - n(c_{A\to B}) + (\ell-1)(a(c_{A\leftarrow B}) - a(c_{A\to B}))$.

This is true, because $a(c_{A\to B}) < a(c_{A\leftarrow B})$ and $n(c_{A\to B}) < n(c_{A\leftarrow B})$. As a result, strand $c_{A\leftarrow B}$ is the only strand in the output type and the strand also qualifies for mandatory. The $\mathfrak{h}$-bit of the output type is set to *true*.

□

Block Selecting. Let x be a complex variable. Let A and B be attributes. Let a be an atomic value symbol and let i be a counter variable. We use *blockselect*(x, A, B, a, i) to abbreviate

$$\texttt{let } y := \texttt{blockexcept}(\textit{blockfromto}(x, A, B), i) \cup \texttt{immob}(\overline{a}) \texttt{ in}$$
$$\texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(y)))$$

Lemma 9.5. *Let S be a circularization of a relation-schema type with k attributes, denoted $A_1, \ldots, A_k$. Let A and B be two consecutive attributes on the circular strand of S. Let $\tau = (S, S, true)$ be a type. Let $\tau_s = (S, \texttt{empty}, true)$ be a type. Let Γ be a type assignment such that $\Gamma(x) = \tau$. Then Γ : blockselect$(x, A, B, a, i) \vdash \tau_s$.*

Proof. With c we denote the circular strand in S. We derive the output type of *blockselect*(x, A, B, a, i) under type assignment Γ.

1. Γ : *blockfromto*$(x, A, B) \vdash \tau_1$. Type τ is saturated. Type τ_1 contains a single, circular, mandatory strand t isomorphic to c, except that the nodes between the two nodes labeled A and B are the only free nodes. In particular, the only node labeled $*$ is the ℓ-core of attribute A. The $\mathfrak{h}$-bit of type τ_1 is set to *true*.
2. Γ : $\texttt{blockexcept}(\textit{blockfromto}(x, A, B), i) \vdash \tau_2$. Type τ_2 consists of a single, circular, mandatory strand u, isomorphic to strand t, except that the only node labeled $*$ in t is relabeled to $\hat{*}$ in u and the nodes labeled $\#_3$ and $\#_4$ (respectively before and after the ℓ-core of attribute A) are blocked. The $\mathfrak{h}$-bit of type τ_2 is set to *true*.
3. Γ : $\texttt{blockexcept}(\textit{blockfromto}(x, A, B), i) \cup \texttt{immob}(\overline{a}) \vdash \tau_y$. Type τ_y consists of two mandatory complexes: strand u and a probe n labeled ?. As probe n and the $\hat{*}$ labeled node of strand u are free and can match, the $\mathfrak{h}$-bit of type τ_y is set to *false*.
   From this point on, we use the type assignment $\Gamma' = \Gamma \cup \{(y, \tau_y)\}$.
4. Γ' : $\texttt{hybridize}(y) \vdash \tau_3$. Type τ_y consists of two mandatory components, however, probe n is labeled with ?, hence the probe is not a necessary component. As a result, there are two sets X to consider in the hybridization process. The first set X consists solely of the strand u. Strand u is a positive strand, thus no matchings can be added. The second set X consists of strand u and probe n. The hybridization of X results in a single, immobilized component called C_1, built up from a strand isomorphic to u and a probe isomorphic to n, with a matching between the only $\hat{*}$-labeled node of the strand and the probe. Because neither u or C_1 is present in both invocations of $\texttt{hybridize}_t$, both are non-mandatory components. The $\mathfrak{h}$-bit of type τ_y is set to *true*.
5. Γ' : $\texttt{flush}(\texttt{hybridize}(y)) \vdash \tau_4$. Component C_1 is the only immobilized component in type τ_3. Type τ_4 thus consists solely of component C_1. The component is non-mandatory. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.
6. Γ' : $\texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(y))) \vdash \tau_s$. The cleanup operation removes blockings, matchings and probes. What is left, is a strand isomorphic to strand c. The strand is non-mandatory. The $\mathfrak{h}$-bit of the output type is set to *true*.

□

9.2.2. Relational Algebra Expressions. The proof of Theorem 9.1 now goes by induction on the structure of e. By induction, we know that subexpressions e_1 and e_2 of expression e are simulated by the respective well-typed (under type assignment $\Gamma_{\mathcal{D}}$) DNAQL expressions e_1^{DNA} and e_2^{DNA}. Subexpression e_1 (e_2) is over relation schema R (S). Type τ_R (τ_S) is a relation-schema-type of relation schema R (S). Type τ'_R (τ'_S) has a weak type isomorphic to τ_R (τ_S), but in contrast to τ_R (τ_S), which has the empty complex as mandatory weak type, the mandatory weak type of τ'_R (τ'_S) is isomorphic to the weak type of τ'_R (τ'_S).

Union.

LEMMA 9.6. *Let $e = e_1 \cup e_2$, with $\mathcal{D} : e \vdash R$. If expression e^{DNA} is defined as $e_1^{DNA} \cup e_2^{DNA}$, then $\Gamma_{\mathcal{D}} : e^{DNA} \vdash \tau_R$.*

Proof. Because the RA expression is defined, expressions e_1 and e_2 are over the same relation schema $R = S$. As a result, $\Gamma_{\mathcal{D}} : e_1^{DNA} \vdash \tau_R$ and $\Gamma_{\mathcal{D}} : e_2^{DNA} \vdash \tau_R$. The union of two isomorphic types is trivially isomorphic to the input types. $\square$

Difference.

LEMMA 9.7. *Let $e = e_1 - e_2$, with $\mathcal{D} : e \vdash R$. If expression e^{DNA} is defined as $e_1^{DNA} - e_2^{DNA}$, then $\Gamma_{\mathcal{D}} : e_1^{DNA} - e_2^{DNA} \vdash \tau_R$.*

Proof. Because the RA expression e is well typed, expressions e_1 and e_2 are over the same relation schema $R = S$. As a result, $\Gamma_{\mathcal{D}} : e_1^{DNA} \vdash \tau_R$ and $\Gamma_{\mathcal{D}} : e_2^{DNA} \vdash \tau_R$.

If relation schema R is empty, i.e., no attributes, type $\tau_R = (\mathtt{empty}, \mathtt{empty}, \mathit{true})$. The difference operation applied to two empty complex types, results in the empty complex type. Hence, the output is also of type τ_R.

If R is non-empty, the sole strand of S_R is also the sole member of $\mathit{data}(S_R)$. Thus, the weak type of the output is S_R. Furthermore, $\odot_R$ is empty, thus the mandatory weak type of the output is also empty. The $\mathfrak{h}$-bit of the output is set to *true*. Because, τ_R consists solely of nodes labeled with positive symbols, any complex having type τ_R is hybridized.
$\square$

Cartesian product. The cartesian product simulation consists of two parts. First, the strands of e_1 and e_2 are concatenated. In a second step, the attributes are shuffled, to restore the fixed order on the attributes.

Let R and S be relation schemas with respective orders $\oplus_R$ and $\oplus_S$. If $R \cap S = \emptyset$, we define relation schema T as $R \cup S$. We define the *combined order* $\oplus$ of orders $\oplus_R$ and $\oplus_S$, such that for any pair of attributes $X, Y \in T$, $X \oplus Y$ if and only if:

1. $X \in S$ and $Y \in R$; or
2. $X, Y \in R$ and $X \oplus_R Y$; or
3. $X, Y \in S$ and $X \oplus_S Y$.

In other words, the combined order $\oplus$ on relation schema T puts attributes of S in front of attributes of R and respects orders $\oplus_R$ and $\oplus_S$.

LEMMA 9.8. *Let $e = e_1 \times e_2$, with $\mathcal{D} : e \vdash T$, where $T = R \cup S$ and $R \cap S = \emptyset$. Let τ_R (τ_S) be the relation-schema-type of relation schema R (S) with k attributes, denoted $A_1, \ldots, A_k$ ($B_1, \ldots, B_m$). Let $A = A_1$ ($C = B_1$) be the first and $B = A_k$ ($D = B_m$) be the last attribute of relation schema R under order $\oplus_R$. Let $\oplus$ be the combined order of $\oplus_R$ and $\oplus_S$. Let $\tau_T^{\oplus}$ be a relation-schema-type with order $\oplus$. If* *expression* e^{DNA} *is defined as*

```
let $x := e_1^{DNA}$ in
let $y := e_2^{DNA}$ in
if empty$(x)$ then empty else
 if empty$(y)$ then empty else
  let $r :=$ hybridize$(\overline{\#_4\#_5} \cup \#_5)$ in
  let $l :=$ hybridize$(\overline{\#_1\#_2} \cup \#_1)$ in
  let $e_2^a := \mathit{connect}(x \cup r)$ in
  let $e_2^b := \mathit{connect}(y \cup l)$ in
  let $e_2 := \mathit{connect}(e_2^a \cup e_2^b \cup \overline{\#_5\#_1})$ in
  let $e_1 := \mathit{circularize}(e_2, A, D)$ in
   cleanup(split(split($\mathit{blockfromto}(e_1, B, C), \#_2), \#_4$))
```

then $\Gamma_{\mathcal{D}} : e^{DNA} \vdash \tau_T^{\oplus}$.

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

Proof. By the well-typedness of e we know that τ_R and τ_S do not share attributes. We derive the output type of e^{DNA} under type assignment $\Gamma_{\mathcal{D}}$.

1. We extend and augment the type assignment $\Gamma_{\mathcal{D}}$ with $\{(x, \tau_R'); (y, \tau_S')\}$. Type checking the first two let-statements, adds the types for complex variables x and y. The main body of the expression resides inside the else-part of two if-statements. Because τ_R and τ_S both consist of a single, non-mandatory strand, the main body of the expression is type checked with the augmented types τ_R' and τ_S' for x respectively y. We denote the extended and augmented type assignment by Γ'.
2. $\Gamma' : \overline{\#_4\#_5} \cup \#_5 \vdash \tau_0$. Type τ_0 consists of a single-node strand t_0 labeled $\#_5$ and a sticker s_0 labeled $\overline{\#_4\#_5}$. Both components are mandatory. The $\mathfrak{h}$-bit of type τ_0 is set to *false*, because the node of t_0 can match with the node labeled $\overline{\#_5}$ of sticker s_0.
3. $\Gamma' : \mathit{hybridize}(\overline{\#_4\#_5} \cup \#_5) \vdash \tau_1$. Type τ_0 consists of two mandatory components and none of the nodes is labeled ?, thus both components are necessary. Type τ_1 consists of a single, mandatory component, formed by binding strand t_0 on sticker s_0. The $\mathfrak{h}$-bit is set to *true*.
   From this point on, we use type assignment $\Gamma_1 = \Gamma' \cup \{(r, \tau_1)\}$.
4. $\Gamma_1 : \overline{\#_1\#_2} \cup \#_1 \vdash \tau_2'$. Type τ_2' consists of a single-node strand t_2 labeled $\#_1$ and a sticker s_2 labeled $\overline{\#_1\#_2}$. Both components are mandatory. The $\mathfrak{h}$-bit of type τ_2' is set to *false*, because the node of t_2 can match with the node labeled $\overline{\#_1}$ of sticker s_2.
5. $\Gamma_1 :$ hybridize$(\overline{\#_1\#_2} \cup \#_1) \vdash \tau_2$. Type τ_2' consists of two mandatory components and none of the nodes is labeled ?, thus both components are necessary. Type τ_2 consists of a single, mandatory component, formed by binding strand s_2 on sticker s_2. The $\mathfrak{h}$-bit is set to *true*.
   From this point on, we use the type assignment $\Gamma_2 = \Gamma_1 \cup \{(l, \tau_2)\}$.
6. $\Gamma_2 : \mathit{connect}(x \cup r) \vdash \tau_3$. Firstly, τ_R' consists of a single component, denoted t_R. Type τ_1 consists of a single component, denoted C_R. Both components

are mandatory within their respective types. Strand t_R has at least one free node labeled $\#_4$ and component C_R has one free node labeled $\overline{\#_4}$. As a result, the union results in a type called τ' with components t_R and C_R that are both mandatory. The $\mathfrak{h}$-bit is set to *false*.

The connect abbreviation consists of (a) a hybridization, (b) a ligation, and (c) a cleanup:

(a) As both components in τ' are mandatory and no node is labeled ?, both are necessary components. Hence, there is only one hybridization set X containing both components. The hybridization results in a component C_R^2 by matching an isomorphic copy of C_R to every free $\#_4$-labeled node on one isomorphic copy of t_R.

(b) Component C_R^2 has one gap, namely between the strand isomorphic to strand t_R and the isomorphic copy of strand t_0 attached to the last node of t_R. All other copies of component C_R do not create a gap. The ligate operation constructs a new component C_R^3 in which this gap is filled.

(c) There are essentially two strands in the C_R^3, the concatenation of t_R and t_0, called t'_R, and t_0. The length of t'_R is $5k + 1 + (\ell - 1)k$. The length of isomorphic copies of t_0 is 1.

Type τ_3 thus consists of strand t'_R. The strand is mandatory. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.

From this point on, we use the type assignment $\Gamma_3 = \Gamma_2 \cup \{(e_2^a, \tau_3)\}$.

7. Γ_3 : *connect*$(y \cup l) \vdash \tau_4$. An analogous reasoning to the above point reveals that type τ_4 consists of a single strand, called t'_S, which is a concatenation of strand t_2 and t_S, i.e., t'_S is equivalent to t_S except that a node labeled $\#_1$ is attached to the front of it.

   From this point on, we use the type assignment $\Gamma_4 = \Gamma_3 \cup \{(e_2^b, \tau_4)\}$.

8. Γ_4 : *connect*$(e_2^a \cup e_2^b \cup \overline{\#_5 \#_1}) \vdash \tau_5$. Firstly, we combine the mandatory strands t'_R and t'_S with sticker s_3, labeled $\overline{\#_5 \#_1}$. The $\mathfrak{h}$-bit of this combination is set to *false*, because strand t'_R (t'_S) contains a free node labeled $\#_5$ ($\#_1$) which can match with sticker s_3.

   The connect abbreviation consists of (a) a hybridization, (b) a ligation, and (c) a cleanup.

   (a) Because all components are mandatory and there is no ?-labeled node, there is only one set X to hybridize. The result is a component C_1 consisting of one isomorphic copy of t'_R, one isomorphic copy of t'_S and one isomorphic copy of s_3. The sticker concatenates the positive strands. Component C_1 is mandatory and the $\mathfrak{h}$-bit is set to *true*.

   (b) The last node of the isomorphic copy of t'_R and the first node of the isomorphic copy of t'_S form a gap. The ligate operation fills this gap. Hence, component C'_1 consists of one positive strand, called t_{RS}, which is the concatenation of t'_R and t'_S and one sticker isomorphic to s_3. Component C'_1 is mandatory and the $\mathfrak{h}$-bit remains set to *true*.

   (c) As strand t_{RS} is the only strand in component C'_1, the result of the cleanup operation is t_{RS}. Strand t_{RS} also qualifies for mandatory, because it is the only strand and component C'_1 is mandatory. The $\mathfrak{h}$-bit of type τ_5 is set to *true*.

   From this point on, we use the type assignment $\Gamma_5 = \Gamma_4 \cup \{(e_2, \tau_5)\}$.

9. Γ_5 : *circularize*$(e_2, A, D) \vdash \tau_6$. Type τ_5 consists of strand t_{RS}, which is a pseudo-relation-schema-type at this point, because of the two nodes labeled

$\#_1$ and $\#_5$ between attributes B and C. Moreover, strand t_{RS} is also mandatory in τ_5. The $\mathfrak{h}$-bit of type τ_5 is set to *true*. The first (last) attribute of strand t_{RS} is A (D). Hence, by Lemma 9.2 we know that type τ_6 consists of the circular version of strand t_{RS}, called c_{RS}. Moreover, c_{RS} is mandatory in type τ_6 and the $\mathfrak{h}$-bit of type τ_6 is set to *true*.
From this point on, we use the type assignment $\Gamma_6 = \Gamma_5 \cup \{(e_1, \tau_6)\}$.

10. Γ_6 : $\mathit{blockfromto}(e_1, B, C) \vdash \tau_7$. Type τ_7 consists of an isomorphic copy of c_{RS}, except that the nodes starting from the node labeled $\#_3$ directly following the B-labeled node, up to but not including the C-labeled node are the only free nodes. All other nodes are blocked. The strand is mandatory and the $\mathfrak{h}$-bit of type τ_7 is set to *true*.
11. Γ_6 : $\texttt{cleanup}(\texttt{split}(\texttt{split}(\mathit{blockfromto}(e_1, B, C), \#_2), \#_4)) \vdash \tau_8$. In the strand of type τ_7 there is only one free $\#_2$-labeled node and only one free $\#_4$-labeled node, namely, the first node of the attribute-value block of C and the last node of the attribute-value block of B. Between these nodes, there are two nodes, labeled $\#_5$ and $\#_1$. The two split operations cut these two nodes from the strand, forming a new short strand called t_{15}. As a result, the circular strand becomes linear and attribute C is the first attribute. Call this strand t_{SR}. Note that strand t_{SR} is the concatenation of strands t_S and t_R.
The length of strand t_{SR} equals $(4 + \ell)(k + m)$, whereas the length of strand t_{15} equals 2. Hence, the cleanup operation removes the blockings from strand t_{SR} and disposes of strand t_{15}. Strand t_{SR} is mandatory in type τ_8 and the $\mathfrak{h}$-bit is set to *true*.
12.

```
Γ'' :  if empty(y) then empty else
       let r := hybridize($\overline{\#_4\#_5} \cup \#_5$) in
       let l := hybridize($\overline{\#_1\#_2} \cup \#_1$) in
       let $e_2^a$ := $\mathit{connect}(x \cup r)$ in
       let $e_2^b$ := $\mathit{connect}(y \cup l)$ in
       let $e_2$ := $\mathit{connect}(e_2^a \cup e_2^b \cup \overline{\#_5\#_1})$ in
       let $e_1$ := $\mathit{circularize}(e_2, A, D)$ in
       cleanup(split(split($\mathit{blockfromto}(e_1, B, C), \#_2), \#_4$))   $\vdash \tau_T^{\oplus}$
```

By the previous steps, we know that the else-part of the if-statement has type τ_8. By definition we know that Γ'' : $\texttt{empty} \vdash (\texttt{empty}, \texttt{empty}, \mathit{true})$. Hence, combining both types results in $\tau_T^{\oplus}$ as strand t_{SR} in type τ_8 becomes non-mandatory.
13. Γ' : $\texttt{if empty}(x)$ $\texttt{then empty else if empty}(y)$ $\texttt{then empty else}$ $e_y \vdash \tau_T^{\oplus}$. The else-part of the if-test has type $\tau_T^{\oplus}$. The then-part of the if-test has the empty type. Hence, it does not add components to nor does it make components non-mandatory, thus the output type is $\tau_T^{\oplus}$.

□

Reordering is performed by repeated shuffling of attribute-value pairs. Shuffling attribute-value pairs in a tuple is done using a new technique we call *double bridging*. Instead of using a single sticky bridge, two sticky bridges are hybridized onto one strand. A careful placement of the bridges allows us to cut twice in the strand whilst keeping the strand connected. Moreover, the two bridges guide the strand into its new conformation.

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

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

The double bridging will result in non-terminating hybridizations, if the positive strands are not immobilized.

LEMMA 9.9. *Let R be a relation schema with order $\oplus$ on the attributes. Let C be an attribute in R, and let $\otimes$ be the order derived from $\oplus$ by making attribute C the last attribute, i.e., for any attribute $X \in R$, if $X \neq C$ then $X \otimes C$ and for any pair of attributes (X, Y) in R, with $X \neq C$ and $Y \neq C$, $X \oplus Y \Leftrightarrow X \otimes Y$. Let $\tau_R^{\oplus}$ be a relation-schema-type for relation schema R with order $\oplus$. Let A be the first attribute, B be the attribute just in front of attribute C, D be the attribute appearing just after C and E be the last attribute of R under order $\oplus$. Let x be a complex variable. Let e^{DNA} be the following DNAQL expression:*

$$\begin{aligned}
&\texttt{let } f_1 := \mathit{insertcirc}(\mathit{circularize}(x, A, E), B, C, \#_6\#_7) \texttt{ in}\\
&\texttt{let } f_2 := \mathit{insertcirc}(f_1, C, D, \#_8\#_9) \texttt{ in}\\
&\texttt{let } f_3 := \texttt{split}(\mathit{blockfromto}(f_2, E, A), \#_4) \texttt{ in}\\
&\texttt{let } f_4 := \mathit{connect}(\mathit{blockfromto}(f_3, E, A) \cup \#_9\#_1 \cup \overline{\#_4\#_9}) \texttt{ in}\\
&\texttt{let } f_5 := \texttt{hybridize}(f_4 \cup \texttt{immob}(\overline{A})) \cup \overline{\#_6\#_8} \cup \overline{\#_1\#_7} \texttt{ in}\\
&\texttt{let } f_6 := \texttt{cleanup}(\texttt{ligate}(\texttt{split}(\texttt{split}(f_5, \#_6), \#_8))) \texttt{ in}\\
&\texttt{let } f_7 := \mathit{removeBetweenCirc}(\mathit{circularize}(f_6, A, C), B, D) \texttt{ in}\\
&\texttt{let } f_8 := \mathit{removeBetweenCirc}(\mathit{circularize}(f_7, D, B), E, C) \texttt{ in}\\
&\ \mathit{removeBetweenCirc}(\mathit{circularize}(f_8, C, E), C, A)
\end{aligned}$$

Let Γ be a type assignment such that $\Gamma(x) = \tau_R^{\oplus}$. Then $\Gamma : e^{DNA} \vdash \tau_R^{\otimes}$.

Proof. The strand in type $\tau_R^{\oplus}$ is denoted t_R. For the sake of brevity, we have omitted an emptiness-test on complex variable x. Hence, type $\tau_R^{\oplus}$ replaced by its augmented version, in which strand t_R is mandatory.

We derive the output type of e^{DNA} under type assignment Γ.

1. Γ : *circularize*$(x, A, E) \vdash \tau_1$. Strand t_R in type $\tau_R^{\oplus}$ is a positive, linear, and mandatory strand. It does not contain matchings, blockings, or immobilizations. The first attribute occurring on strand t_R is A and the last attribute is E. Hence, by Lemma 9.2 we know that type τ_1 consists of the circular version of strand t_R, called c_R. Strand c_R is mandatory in type τ_1 and the $\mathfrak{h}$-bit is set to *true*.
2. Γ : *insertcirc*(*circularize*$(x, A, E), B, C, \#_6\#_7) \vdash \tau_2$. Strand c_R is circular and it is mandatory in type τ_1. As strand c_R is derived from strand t_R and t_R is a relation-schema-type, there are no nodes between the attribute-value block of B and C. The labels $\#_6$ and $\#_7$ are unique with respect to the strand $\#_6\#_7$ and the c_R. Hence, by Lemma 9.3 we know that type τ_2 consist of a mandatory, circular strand, isomorphic to c_R except that two nodes, labeled

$\#_6$ resp. $\#_7$, are added between the attribute-value blocks of attributes B and C. Call this strand c_R^1. The $\mathfrak{h}$-bit of type τ_2 is set to *true*.
From this point on, we use the type assignment $\Gamma_{f_1} = \Gamma \cup \{(f_1, \tau_2)\}$.

3. Γ_{f_1} : *insertcirc*$(f_1, C, D, \#_8\#_9) \vdash \tau_3$. Strand c_R^1 is circular and it is mandatory in type τ_2. As strand c_R is derived from strand t_R and t_R is a relation-schema-type, there are no nodes between the attribute-value block of C and D. The labels $\#_8$ and $\#_9$ are unique with respect to the strand $\#_8\#_9$ and strand c_R^1. Hence, by Lemma 9.3, we know that type τ_3 consists of a mandatory, circular strand, isomorphic to c_R^1 except that two nodes, labeled $\#_8$ resp. $\#_9$, are inserted between the attribute-value blocks of attributes C and D. Call this strand c_R^2. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.
From this point on, we use the type assignment $\Gamma_{f_2} = \Gamma_{f_1} \cup \{(f_2, \tau_3)\}$.
4. Γ_{f_2} : $\texttt{split}(\textit{blockfromto}(f_2, E, A), \#_4) \vdash \tau_4$. Type τ_4 consists of a strand, called c_R^3, isomorphic to strand c_R^2, except that all the nodes from the first attribute to the last attribute are blocked. Hence, the only free node labeled $\#_4$ is the last node of the attribute-value block of E. The split operation results in a linear version of c_R^3 in which A as the first attribute. Call this strand t_R^4. Strand t_R^4 is mandatory in type τ_4. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.
From this point on, we use the type assignment $\Gamma_{f_3} = \Gamma_{f_2} \cup \{(f_3, \tau_4)\}$.
5. Γ_{f_3} : *blockfromto*$(f_3, E, A) \cup \#_9\#_1 \cup \overline{\#_4\#_9} \vdash \tau_5$. Type τ_5 consists of strand t_R^4, a strand t_1 labeled with $\#_9\#_1$, and a sticker s_1 labeled $\overline{\#_4\#_9}$. All components are mandatory. The $\mathfrak{h}$-bit of type τ_5 is set to *false*, because strand t_R^4 has a free $\#_4$-labeled node, strand t_1 has a free $\#_9$-labeled node and the sticker has a free node labeled $\overline{\#_4}$ and a free node labeled $\overline{\#_9}$.
6. Γ_{f_3} : *connect*$(\textit{blockfromto}(f_3, E, A) \cup \#_9\#_1 \cup \overline{\#_4\#_9}) \vdash \tau_6$. The connect abbreviation consists of (a) a hybridization, (b) a ligation and (c) a cleanup.
   (a) All components are mandatory and no node is labeled with ?. Hence, all components are necessary. Strands t_R^4 and t_1 can be connected by means of sticker s_1. Thus, hybridization forms a new component C_1 consisting of one isomorphic copy of strand t_R^4, one isomorphic copy of t_1 and one isomorphic copy of s_1. There is a gap between the last node of the strand isomorphic to t_R^4 and the first node of the strand isomorphic to t_1. Component C_1 is mandatory. The $\mathfrak{h}$-bit is set to *true*.
   (b) The ligate operation will fill the gap between the two positive strands in component C_1. Let component C_1' be the result of the ligate operation on component C_1. Then there is a single positive strand in component C_1', namely the concatenation of strands isomorphic to t_R^4 respectively t_1.
   (c) There is only one positive strand in component C_1'. Hence, type τ_6 consists of a single strand (with blockings), call it t_R^5. Strand t_R^5 is mandatory. The $\mathfrak{h}$-bit of type τ_6 is set to *true*.

From this point on, we use the type assignment $\Gamma_{f_4} = \Gamma_{f_3} \cup \{(f_4, \tau_6)\}$.
7. Γ_{f_4} : $\texttt{hybridize}(f_4 \cup \texttt{immob}(\overline{A})) \vdash \tau_7$. Firstly, a probe n labeled $\overline{A}$ and strand t_R^5 are combined. Both components are mandatory. The $\mathfrak{h}$-bit is set to *false*, because strand t_R^5 and probe n can match.
Because both components are mandatory and no node is labeled with ?, there is only one set X to hybridize. Hybridization forms a new component C_2 consisting of one isomorphic copy of t_R^5 and one isomorphic copy of n. Node

n and the node labeled A in the isomorphic copy of strand t_R^5 are matched. Component C_2 is mandatory and immobilized. The $\mathfrak{h}$-bit of type τ_7 is set to *true*.

8. Γ_{f_4} : $\texttt{hybridize}(f_4 \cup \texttt{immob}(\overline{A})) \cup \overline{\#_6\#_8} \cup \overline{\#_1\#_7} \vdash \tau_8$. Type τ_8 consists of component C_2, a sticker b_1 labeled $\overline{\#_6\#_8}$ and a sticker b_2 labeled $\overline{\#_1\#_7}$. Stickers b_1 and b_2 are called "bridges". All components are mandatory. The $\mathfrak{h}$-bit of type τ_8 is set to *false*.
9. Γ_{f_4} : $\texttt{hybridize}(\texttt{hybridize}(f_4 \cup \texttt{immob}(\overline{A})) \cup \overline{\#_6\#_8} \cup \overline{\#_1\#_7}) \vdash \tau_9$. Because the $\mathfrak{h}$-bit of type τ_8 is *false*, hybridization takes place. All components are mandatory and there is no ?-labeled probe. Hence, all components are necessary components. Component C_2 is immobilized and has one free node labeled $\#_6$, one free node labeled $\#_8$, one free node labeled $\#_7$, and one free node labeled $\#_1$. Each node of the bridges b_1 and b_2 can thus match to exactly one node in one copy of component C_2.
   Hybridization is performed on components C_2, b_1, and b_2. Four new components are formed.
   (a) The first component, call it C_b^1 consists of one isomorphic copy of C_2, one isomorphic copy of b_1, and one isomorphic copy of b_2.
   (b) The second component, call it C_b^2, consists of one isomorphic copy of C_2, one isomorphic copy of b_1, and two isomorphic copies of b_2 (one binding with the free $\#_1$-labeled node in C_2, the other binding with the free $\#_7$-labeled node in C_2). In this case, only bridge b_1 is successfully placed.
   (c) The third component, call it C_b^3, consists of one isomorphic copy of C_2, two isomorphic copies of b_1 (one binding with the free $\#_6$-labeled node in C_2, the other binding with the free $\#_8$-labeled node in C_2), and one isomorphic copy of b_2. In this case, only bridge b_2 is successfully placed.
   (d) The fourth component, call it C_b^4, consists of one isomorphic copy of C_2, two isomorphic copies of b_1, and two isomorphic copies of b_2. Each bridge matches with one free node. In this case, no bridge is placed successfully.

   Each component C_b^i, for $1 \leq i \leq 4$, is mandatory. The $\mathfrak{h}$-bit of type τ_9 is set to *true*.
   From this point on, we use the type assignment $\Gamma_{f_5} = \Gamma_{f_4} \cup \{(f_5, \tau_9)\}$.
   Type τ_9 is depicted in Figure 9.1.
10. Γ_{f_5} : $\texttt{split}(\texttt{split}(f_5, \#_6), \#_8) \vdash \tau_{10}$. Type τ_9 consists of four mandatory, immobilized components. Each component has one closed node labeled $\#_6$ and one closed node labeled $\#_8$.
   (a) Component C_b^1 has two successfully placed bridges. Cutting at the nodes labeled $\#_6$ and $\#_8$ results in a component C_c^1 in which the attribute-value block of C is pulled to position just after the attribute-value block of E. The substrand starting with the attribute-value block of D is pulled against the attribute-value block of B. This situation is depicted in Figure 9.2. The component consists of three positive strand with gaps between the first and second substrand and the second and third substrand. Note that some additional nodes are still present between some attribute-value blocks.
   (b) In component C_b^2, two copies of b_2 are present. As a result, two new components C_c^2 and C_c^3 are created by the split operations. Component

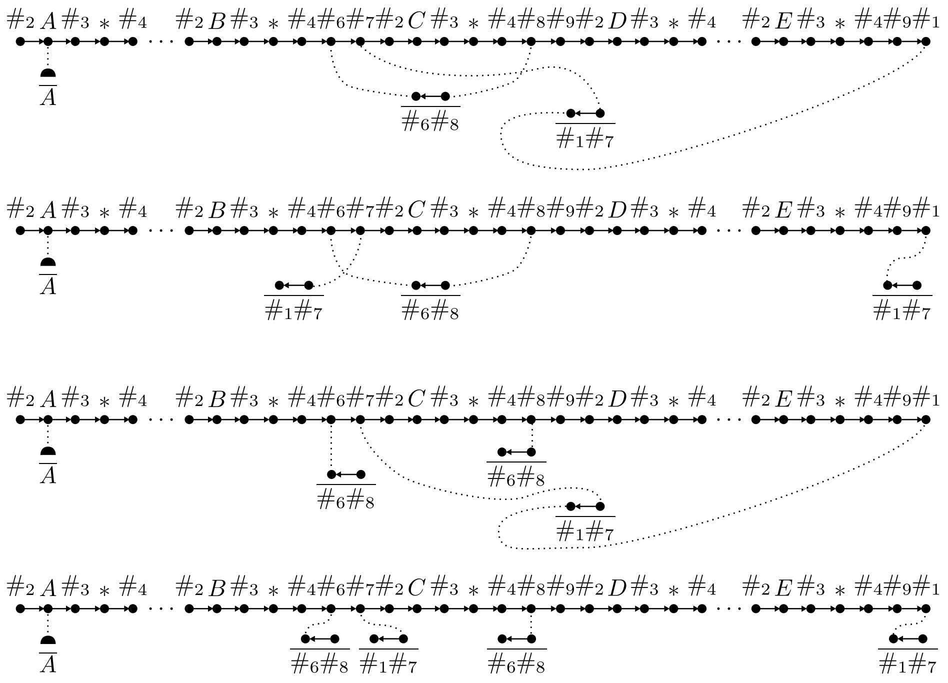

#2 A #3 * #4 · · · #2 B #3 * #4 #6 #7 #2 C #3 * #4 #8 #9 #2 D #3 * #4 · · · #2 E #3 * #4 #9 #1
A
#6#8
#1#7
#2 A #3 * #4 · · · #2 B #3 * #4 #6 #7 #2 C #3 * #4 #8 #9 #2 D #3 * #4 · · · #2 E #3 * #4 #9 #1
A
#1#7
#6#8
#1#7
#2 A #3 * #4 · · · #2 B #3 * #4 #6 #7 #2 C #3 * #4 #8 #9 #2 D #3 * #4 · · · #2 E #3 * #4 #9 #1
A
#6#8
#6#8
#1#7
#2 A #3 * #4 · · · #2 B #3 * #4 #6 #7 #2 C #3 * #4 #8 #9 #2 D #3 * #4 · · · #2 E #3 * #4 #9 #1
A
#6#8
#1#7
#6#8
#1#7

FIGURE 9.1. *Type τ_9: two bridges attached to the data strand.*

C_c^2 consists of two substrands held together by a copy of bridge b_1. A copy of bridge b_2 is also present in this component. The attribute-value block C has been cut from this component. Component C_c^3 consists of a strand labeled with $\#_7$ and the attribute-value block of C and a copy of bridge b_2.

(c) In component C_b^3, two copies of b_1 are present. As a result, two new components C_c^4 and C_c^5 are created by the split operations. Component C_c^4 consists of the attribute-value blocks of attributes A to B, with an additional node labeled $\#_6$. Bridge b_1 is matched to this strand. Component C_c^5 also consists of linear strand, starting with two nodes labeled $\#_8$ and $\#_9$, followed by the attribute-value blocks of attributes D to E, three nodes labeled $\#_9$, $\#_1$ and $\#_7$, and the attribute-value block of attribute C.

(d) The bridges in component C_b^4 do not connect any part of the component isomorphic to C_2. Consequently, three new components C_c^6, C_c^7 and C_c^8 are created by the split operations. Component C_c^6 consists of a linear strand of the attribute-value blocks of attributes A to B followed by a node labeled $\#_6$. Furthermore, a copy of bridge b_1 is matched to the strand. Component C_c^7 consists of a linear strand labeled with $\#_7$ and the attribute-value block of attribute C. A copy of bridge b_2 is matched to the strand. Component C_c^8 consists of a linear strand labeled $\#_8\#9$, followed by the attribute-value blocks of the attributes D to E, and two nodes labeled $\#_9\#_1$. One copy of bridge b_1 and one copy of bridge b_2 is matched to the strand.

Type τ_{10} consists of eight mandatory components C_c^1 to C_c^8. The $\mathfrak{h}$-bit of

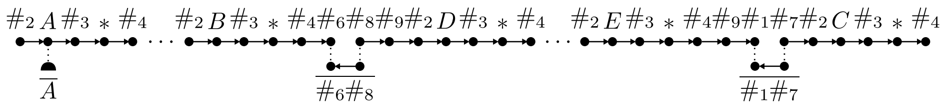

#2 A #3 * #4 · · · #2 B #3 * #4#6#8#9#2 D #3 * #4 · · · #2 E #3 * #4#9#1#7#2 C #3 * #4
A
#6#8
#1#7

FIGURE 9.2. *Type τ_{10}: the two bridges have guided the strand into its new conformation. Note there are still gaps in the data strand.*

type τ_{10} is set to *true*.

11. Γ_{f_5} : $\texttt{cleanup}(\texttt{ligate}(\texttt{split}(\texttt{split}(f_5, \#_6), \#_8))) \vdash \tau_{11}$. There are four gaps in the components of type τ_{10}. In component C_c^1 the ligate operation glues the three linear strand together in their new conformation, creating a new component called D_1. Component D_1 is depicted in Figure 9.2. Component D_1 contains a one positive strand with the all attribute-value blocks. Call thus strand t_2. In component C_c^2 the ligate operation glues the two positive strands together, creating a new component called D_2. Component D_2 contains the attribute-value blocks of all attributes except attribute C. In component C_c^5 the ligate operation glues the two linear strands together, creating a new component called D_3. Component D_3 contains the attribute-value blocks of attributes D to E and C.
    The cleanup operation is applied to eight mandatory components: D_1, D_2, C_c^3, C_c^4, D_3, C_c^5, C_c^6, C_c^7, and C_c^8. All of these components contain a single positive strand and one or more sticker. The lengths of the strands in the components are:
    (a) D_1: $|R|(4+\ell)+6$
    (b) D_2: $(|R|-1)(4+\ell)+5$
    (c) C_c^3: $5+\ell$
    (d) C_c^4: $|\{A,\ldots,B\}|(4+\ell)+1$
    (e) D_3: $|\{D,\ldots,E,C\}|(4+\ell)+5$
    (f) C_c^6: $|\{A,\ldots,B\}|(4+\ell)+1$
    (g) C_c^7: $5+\ell$
    (h) C_c^8: $|\{D,\ldots,E\}|(4+\ell)+4$
    Clearly, strand t_2 in component D_1 is always the longest strand. Moreover, strand t_2 qualifies for mandatory. The $\mathfrak{h}$-bit of type τ_{11} is set to *true*.
    From this point on, we use the type assignment $\Gamma_{f_6} = \Gamma_{f_5} \cup \{(f_6, \tau_{11})\}$.
12. Γ_{f_6} : *removeBetweenCirc*(*circularize*$(f_6, A, C), B, D) \vdash \tau_{12}$. Strand t_2 is a pseudo-relation-schema type that is mandatory in type τ_{11}, moreover it is the only component of type τ_{11}. Hence, by Lemma 9.2, we know that the output type of *circularize*(f_6, A, C) contains the circular version of t_2. Call this strand c_2. Strand c_2 is mandatory, it is a pseudo-relation-schema type and between the attribute-value blocks of attributes B and D there are only three nodes. Hence, by Lemma 9.4, we know that type τ_{12} consists of a linear strand t_3 starting with attribute D and ending with attribute B. Strand t_3 is mandatory in type τ_{12}. The $\mathfrak{h}$-bit of type τ_{12} is set to *true*.
    From this point on, we use the type assignment $\Gamma_{f_7} = \Gamma_{f_6} \cup \{(f_6, \tau_{12})\}$.
13. Γ_{f_7} : *removeBetweenCirc*(*circularize*$(f_7, D, B), E, C) \vdash \tau_{13}$. In a reasoning similar to the previous item, we derive that τ_{13} consists of a strand, call it t_4, that is isomorphic to strand t_3, except that the three nodes labeled $\#_9\#_1\#_7$ between the attribute-value blocks of attributes E and C have been removed. Strand t_4 starts with attribute C and ends with attribute E. It is mandatory

in type τ_{14}, and the $\mathfrak{h}$-bit of τ_{13} is set to *true*.
From this point on, we use the type assignment $\Gamma_{f_8} = \Gamma_{f_7} \cup \{(f_8, \tau_{13})\}$.

14. Γ_{f_8} : *removeBetweenCirc*(*circularize*$(f_8, C, E), C, A) \vdash \tau_R^{\otimes}$. Strand t_4 in type τ_{13} is mandatory and a relation-schema-type. Hence, by Lemma 9.2, we know that the output type of *circularize*(f_8, C, E) consists of a circular version of strand t_4. Call this strand c_4. Strand c_4 is mandatory and circular. There are no nodes between the attribute-value blocks of attributes C and A. Hence, by Lemma 9.4, we know that the output type consists of a linear strand, starting with attribute A and ending with attribute C that is a relation-schema type. Hence, the output type is $\tau_R^{\otimes}$.

□

Projection. Computing the simulating expression for $\widehat{\pi}_C(e_1)$, is split into two cases: (1) relation schema R has three or more attributes, and (2) relation schema R has two attributes.

Three or More Attributes.

Lemma 9.10. *Let $e = \widehat{\pi}_C(e_1)$, with $\mathcal{D} : e_1 \vdash R$, $C \in R$ and $\mathcal{D} : e \vdash S$, where $S = R \setminus \{C\}$. Let τ_R be a relation-schema-type of relation schema R with $k \geq 3$ attributes, and let $S = \{A_1, \dots A_{i-1}, A_{i+1}, \dots, A_k\}$ be a relation-schema-type with $k-1$ attributes. Denote attribute A_i with C, let $A = A_1$, and let $B = A_k$. Consider R to be circular, then let X be the attribute just in front of C and Y be the attribute directly following C. Let A' be the first attribute of S and let B' be the last attribute of S. If expression e^{DNA} is defined as*

```
let $x := e_1^{DNA}$ in if empty($x$) then empty else
  let $y :=$ split($blockfromto(circularize(x, A, B), X, Y), \#_2$) in
  $removeBetweenCirc(circularize($cleanup($y$)$, Y, X), B', A')$
```

then $\Gamma_{\mathcal{D}} : e^{DNA} \vdash \tau_S$.

Proof. Let t_R be the strand in τ_R. We derive the output type of e^{DNA} under type assignment $\Gamma_{\mathcal{D}}$.

1. Firstly, we extend and augment the type assignment $\Gamma_{\mathcal{D}}$ with $\{(x, \tau_R')\}$ and denote the extended and augmented type assignment Γ'. This pair may be added to Γ, because the let-statement introduces variable x with type τ_R. Strand t_R is non-mandatory in type τ_R and the main body of the program is situated in the else-part of the if-statement. By definition, we may thus augment the type of variable x to τ_R' in which strand t_R is mandatory.
2. Γ' : *circularize*$(x, A, B) \vdash \tau_1$. Strand t_R is linear and without blockings, matchings and immobilizations. Furthermore t_R is mandatory in type τ_R'. The first attribute of t_R is A and the last attribute is B. By Lemma 9.2, we know that type τ_1 consists of a circular version of t_R. Denote this circular strand c_R. Strand c_R is mandatory. The $\mathfrak{h}$-bit of type τ_1 is set to *true*.
3. Γ' : *blockfromto*(*circularize*$(x, A, B), X, Y) \vdash \tau_2$. Strand c_R of type τ_1 is circular, and attribute X is just in front of attribute C and attribute Y is just behind attribute C. The blockfromto abbreviation creates a new circular strand, call it c_R^2, in which all nodes, except those in the substrand between attribute X and Y, are blocked. Consequently, only two nodes labeled $\#_2$ are free. The first node of the attribute-value block of C and the first node of the attribute-value block of attribute Y, i.e., the first node after the attribute-value block of C. Strand C_r^2 is mandatory in type τ_2 and the $\mathfrak{h}$-bit of type τ_2 is set to *true*.

4. Γ' : $\texttt{split}(\textit{blockfromto}(\textit{circularize}(x, A, B), X, Y), \#_2) \vdash \tau_3$. The split operation introduces two new strands. The first one, called t'_S, contains the attribute-value blocks of relation schema S. The first attribute of t'_S is Y, the last attribute is X. The nodes from attribute Y up to and including attribute X are blocked. The second strand, called t_C, contains the attribute-value block of attribute C. No nodes are blocked in this strand. Both components are mandatory. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.
   From this point on, we use the type assignment $\Gamma_1 = \Gamma' \cup \{(y, \tau_3)\}$.
5. Γ_1 : $\texttt{cleanup}(y) \vdash \tau_4$. Type τ_3 contains two strands, namely, t'_S and t_C. The length of these strands is $(k-1)(4+\ell)$ respectively $4+\ell$. As $k > 1$, strand t'_S qualifies. Because t'_S is mandatory in τ_3, it qualifies for mandatory. Strand t'_S without blockings is called t''_S. Hence, type τ_4 consists of t''_S as a mandatory strand. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.
6. Γ_1 : $\textit{removeBetweenCirc}(\textit{circularize}(\texttt{cleanup}(y), Y, X), B', A') \vdash \tau_5$. Strand t''_S is mandatory, linear and without blockings and matchings. Its first attribute is X and its last attribute is Y. By Lemma 9.2, we know that c_S is the circular version of t''_S. Strand c_S is mandatory in the output type of the circularize abbreviation. Next, the *removeBetweenCirc* abbreviation cuts open the circle between attributes B' and A'. Type τ_5 thus consists of a linear version of c_S with first attribute A' and last attribute B'. Call this strand t_S. Strand t_S is mandatory in type τ_5. The $\mathfrak{h}$-bit of type τ_5 is set to *true*.
7.

$$\begin{aligned}&\Gamma' : \texttt{if empty}(x) \texttt{ then empty else}\\ &\quad \texttt{let } y := \texttt{split}(\textit{blockfromto}(\textit{circularize}(x, A, B), X, Y), \#_2) \texttt{ in}\\ &\quad \textit{removeBetweenCirc}(\textit{circularize}(\texttt{cleanup}(y), Y, X), B', A') \vdash \tau_T\end{aligned}$$

   The then-part of the if-statement has the empty type, and the else-part of the if-statement has type τ_5. The union of the weak types and the intersection of the mandatory weak types results in τ_S. As the weak type only contains positively labeled nodes, the $\mathfrak{h}$-bit can be set to *true*.

□

Two attributes.

LEMMA 9.11. *Let $e = \widehat{\pi}_C(e_1)$, with $\mathcal{D} : e_1 \vdash R$, $R = \{A, C\}$, and $\mathcal{D} : e \vdash S$, where $S = \{A\}$. Let τ_R be a relation-schema-type of relation schema R, and let τ_S be a relation-schema-type of relation schema S. If expression e^{DNA} is defined as*

$$\begin{aligned}&\texttt{let } x := e_1^{DNA} \texttt{ in if empty}(x) \texttt{ then empty else}\\ &\texttt{cleanup(flush(hybridize(split}(x, \#_4) \cup \texttt{immob}(\overline{A}))))\end{aligned}$$

then $\Gamma_{\mathcal{D}} : e^{DNA} \vdash \tau_S$.

Proof. Let t_R be the linear strand in type τ_R. We derive the output type of e^{DNA} under type assignment $\Gamma_{\mathcal{D}}$.

1. Firstly, we extend and augment the type assignment $\Gamma_{\mathcal{D}}$ with $\{(x, \tau'_R)\}$ and denote the extended and augmented type assignment Γ'. This pair may be added to Γ, because the let-statement introduces variable x with type τ_R. Strand t_R is non-mandatory in type τ_R and the main body of the program is situated in the else-part of the if-statement. By definition, we may thus augment the type of variable x to τ'_R in which strand t_R is mandatory.

2. Γ' : $\texttt{split}(x, \#_4) \vdash \tau_1$. Strand t_R in type τ'_R, has two nodes labeled $\#_4$, namely, the last nodes of the attribute-value blocks of A and C. Consequently, the split operation splits t_R into two linear strands t_A (the attribute-value block of attribute A) and t_C (the attribute-value block of attribute C). Both strands are mandatory in type τ_1. The $\mathfrak{h}$-bit of type τ_1 is set to *true*.
3. Γ' : $\texttt{split}(x, \#_4) \cup \texttt{immob}(\overline{A}) \vdash \tau_2$. Type τ_2 consists of strands t_A, t_C and a probe n labeled $\overline{A}$. Because probe n can match with a node of strand t_A, the $\mathfrak{h}$-bit of type τ_2 is set to *false*. All three components are mandatory.
4. Γ' : $\texttt{hybridize}(\texttt{split}(x, \#_4) \cup \texttt{immob}(\overline{A})) \vdash \tau_3$. All components in type τ_2 are mandatory and no node is labeled with ?. Hence, all three components are necessary. The hybridization produces two mandatory components: the first is formed by the attribute-value block of attribute A and the probe, call it C_1, the second component is isomorphic to t_C. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.
5. Γ' : $\texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(\texttt{split}(x, \#_4) \cup \texttt{immob}(\overline{A})))) \vdash \tau_4$. The flush operation retains component C_1 because it is immobilized. The cleanup operation removes the probe from component C_1, whence extracting strand t_A from C_1. Strand t_A is mandatory in type τ_4. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.
6.

$$\begin{array}{ll}\Gamma' : & \texttt{if empty}(x)\ \texttt{then empty else} \\ & \texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(\texttt{split}(x, \#_4) \cup \texttt{immob}(\overline{A})))) \quad \vdash \tau_T\end{array}$$

The then-part of the if-statement has the empty type. The else-part of the if-statement has type τ_4. Taking the union of the weak types and the intersection of the mandatories, result in type τ_S.

□

Renaming.

LEMMA 9.12. *Let $\mathcal{D}$: $e_1 \vdash R$ with R a relation schema containing at least two attributes and $C \in R$ and $F \notin R$. Let $e = \rho_{C/F}(e_1)$, with $\mathcal{D}$: $e \vdash T$, where $T = (R \setminus \{C\}) \cup \{F\}$ is a relation schema. Let A be the first attribute and let E be the last attribute of R. Consider R to be circular, then let B be the attribute just in front of C and let D be the attribute just after C. If expression e^{DNA} is defined as*

$$\begin{array}{l}\texttt{let}\ x := e_1^{DNA}\ \texttt{in if empty}(x)\ \texttt{then empty else} \\ \texttt{let}\ f_1 := \texttt{cleanup}(\texttt{split}(\textit{blockfromto}(\textit{circularize}(x, A, E), C, D), \#_3))\ \texttt{in} \\ \texttt{let}\ f_2 := \textit{connect}(\textit{blockfromto}(f_1, C, D) \cup \#_2F \cup \overline{F\#_3})\ \texttt{in} \\ \texttt{let}\ f_3 := \texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(f_2, B), \#_2))\ \texttt{in} \\ \ \texttt{cleanup}(\texttt{split}(\textit{blockfromto}(\textit{circularize}(f_3, F, B), E, A), \#_4))\end{array}$$

then $\Gamma : e^{DNA} \vdash \tau_T$.

Proof. Let t_R be the strand in type τ_R. We derive the output type of e^{DNA} under type assignment $\Gamma_{\mathcal{D}}$.

1. Firstly, we extend and augment the type assignment $\Gamma_{\mathcal{D}}$ with $\{(x, \tau'_R)\}$ and denote the extended and augmented type assignment Γ'. This pair may be added to Γ, because the let-statement introduces variable x with type τ_R. Strand t_R is non-mandatory in type τ_R and the main body of the program is situated in the else-part of the if-statement. By definition, we may thus augment the type of variable x to τ'_R in which strand t_R is mandatory.

2. Γ' : $\textit{circularize}(x, A, E) \vdash \tau_1$. Strand t_R is linear, without blockings and matching and mandatory in type τ'_R. By Lemma 9.2, we know that type τ_1 consists of a circular version of strand t_R. Call this circular strand c_R. Strand c_R is mandatory in type τ_1. The $\mathfrak{h}$-bit of type τ_1 is set to *true*.
3. Γ' : $\textit{blockfromto}(\textit{circularize}(x, A, E), C, D) \vdash \tau_2$. The blockfromto abbreviation constructs a new strand, call it c^2_R, that is isomorphic with c_R, except that starting from the node labeled D up to and including the node labeled C, all nodes are blocked. The only nodes that are not blocked are the ones labeled $\#_3, *, \#_4$, and $\#_2$, following the node labeled C. Strand c^2_R is mandatory in type τ_2. The $\mathfrak{h}$-bit of type τ_2 is set to *true*.
4. Γ' : $\texttt{cleanup}(\texttt{split}(\textit{blockfromto}(\textit{circularize}(x, A, E), C, D), \#_3)) \vdash \tau_3$. The only free node labeled $\#_3$ in strand c^2_R of type τ_2, is the node directly following the node labeled C. The split operation makes the circular strand linear again, however, this time the first node is labeled $\#_3$ and the last node is labeled C. Call this strand t_1. There is only one strand in the output of the split operation, whence the cleanup operation removes all blockings from t_1. We call the resulting strand t_2. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.
   From this point on, we use the type assignment $\Gamma_{f_1} = \Gamma' \cup \{(f_1, \tau_3)\}$.
5. Γ_{f_1} : $\textit{blockfromto}(f_1, C, D) \vdash \tau_4$. The blockfromto abbreviation constructs a new strand, call it t_3, from strand t_2 in type τ_3. Strands t_2 and t_3 are isomorphic, except that all nodes in t_3 are blocked, except for the first four nodes, labeled $\#_3 * \#_4 \#_2$. Strand t_3 is mandatory in type τ_4. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.
6. Γ_{f_1} : $\textit{blockfromto}(f_1, C, D) \cup \#_2 F \cup \overline{F\#_3}) \vdash \tau_5$. Type τ_5 consists of strand t_3, strand t_4 labeled $\#_2 F$ and sticker s_1 labeled $\overline{F\#_3}$. All three components are mandatory in type τ_5. The $\mathfrak{h}$-bit of type τ_5 is set to *false*, because matchings are possibly between t_3 and s_1 and between t_4 and s_1.
7. Γ_{f_1} : $\textit{connect}(\textit{blockfromto}(f_1, C, D) \cup \#_2 F \cup \overline{F\#_3}) \vdash \tau_6$. The $\mathfrak{h}$-bit of type τ_5 is *false*, whence hybridization takes place. As all components in type τ_5 are mandatory and no node is labeled ?, there is only one set X to hybridize. The only free node labeled F is the second node of t_4. The only free node labeled $\#_3$ is the first node of t_3. Hence, the result of the hybridization is a new component, consisting of one isomorphic copy of strand t_3, one isomorphic copy of strand t_4, and one isomorphic copy of sticker s_1. The sticker isomorphic to s_1 binds the strand isomorphic to t_4 to the front of the strand isomorphic to strand t_3. Consequently, there is a gap between the positive strands.
   The ligate operation fills the gap between the positive strands, uniting them in a single strand. The cleanup operator constructs a new strand, call it t_6, which is the concatenation of strands t_4 and t_3. Strand t_6 is mandatory in type τ_6. The $\mathfrak{h}$-bit of type τ_6 is set tot *true*.
   From this point on, we use the type assignment $\Gamma_{f_2} = \Gamma_{f_1} \cup \{(f_2, \tau_6)\}$.
8. Γ_{f_2} : $\texttt{blockfrom}(f_2, B) \vdash \tau_7$. The blockfrom operation constructs a new strand, call it t_7, isomorphic to t_6, except that all nodes starting from the first node of the strand up to the node labeled B are blocked. Consequently, the only free node labeled $\#_2$ is the second to last node, just in front of the node labeled C. Strand t_7 is mandatory in type τ_7 and the $\mathfrak{h}$-bit of type τ_7 is set to *true*.
9. Γ_{f_2} : $\texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(f_2, B), \#_2)) \vdash \tau_8$. The split operation con-

structs two new strands, call them t_8 and t_9. Strand t_8 is isomorphic to strand t_7, except that the last two nodes of t_7 are removed. Strand t_9 consists of two nodes labeled $\#_2C$. Both strands are mandatory. The length of strand t_8 is $|R|(4+\ell)$. The length of strand t_9 is 2. Hence, type τ_8 consists of strand t_8, which is mandatory in τ_8. The $\mathfrak{h}$-bit of type τ_8 is set to *true*.
From this point on, we use the type assignment $\Gamma_{f_3} = \Gamma_{f_2} \cup \{(f_3, \tau_8)\}$.

10. Γ_{f_3} : $\mathit{circularize}(f_3, F, B) \vdash \tau_9$. Strand t_8 is mandatory in type τ_8. It starts with attribute F and ends with attribute B. Hence, by Lemma 9.2, we know that type τ_9 consists of the circular version of t_8. Call this circular strand c'_T. The $\mathfrak{h}$-bit of type τ_9 is set to *true*.
11. Γ_{f_3} : $\mathit{blockfromto}(\mathit{circularize}(f_3, F, B), E, A) \vdash \tau_{10}$. Attributes A and E are adjacent on the circle. The blockfromto abbreviation constructs a new strand, call it c''_T, which is isomorphic to c'_T, except that all nodes from E up to and including A are blocked. There is a single free node labeled $\#_2$ in c''_T, namely, the node between the attribute-value blocks of attributes A and E. Strand c''_T is mandatory in type τ_{10}. The $\mathfrak{h}$-bit of type τ_{10} is set to *true*.
12. Γ_{f_3} : $\texttt{cleanup}(\texttt{split}(\mathit{blockfromto}(\mathit{circularize}(f_3, F, B), E, A), \#_4)) \vdash \tau_{11}$.
The split operation splits strand c''_T between attributes E and A. Call the resulting strand t'_T. Attribute A is the first on strand t'_T, attribute E is the last on strand t'_T. There is only one strand, namely, t'_T. Hence, the cleanup operation constructs a new strands t_T isomorphic to strand t'_T except that it has no blockings.
13.
```
Γ′ :  if empty(x) then empty else
      let $f_1 := \texttt{cleanup}(\texttt{split}(\mathit{blockfromto}(\mathit{circularize}(x, A, E), C, D), \#_3))$ in
      let $f_2 := \mathit{connect}(\mathit{blockfromto}(f_1, C, D) \cup \#_2F \cup \overline{F\#_3})$ in
      let $f_3 := \texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(f_2, B), \#_2))$ in
       $\texttt{cleanup}(\texttt{split}(\mathit{blockfromto}(\mathit{circularize}(f_3, F, B), E, A), \#_4)) \vdash \tau_T$
```
The then-part of the if-statement has the empty type. The else-part of the if-statement has type τ_{11}. Combining both types, results in type τ_T.

□

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

Selection.

LEMMA 9.13. *Let $e = \sigma_{B=D}(e_1)$, with $\mathcal{D} : e_1 \vdash R$ and $\mathcal{D} : e \vdash R$. Let A be the first attribute of R and let F be the last attribute of R. Let C be the attribute directly following attribute B. Let E be the attribute directly following attribute D. If expression e^{DNA} is defined as*

```
let $x := e_1^{DNA}$ in for $x_s := x$ iter i do
  if empty($x_s$) then empty else
  let $f :=$
    let $x_c := \mathit{circularize}(x_s, A, F)$ in
    $\bigcup_{a\in\Lambda}$ let $y := \mathit{blockselect}(x_c, B, C, a)$ in
    if empty($y$) then empty else $\mathit{blockselect}(y, D, E, a)$ in
  cleanup(split($\mathit{blockfromto}(f, F, A), \#_4$))
```

then $\Gamma_{\mathcal{D}} : e^{DNA} \vdash \tau_R$.

The alphabet Λ is fixed. The number of atomic value symbols is thus a constant. Hence, the union over all atomic value symbols ($\bigcup_{a\in\Lambda}$) is merely "syntactic sugar" to abbreviate an expression of constant size. Note $A = B$, or $C = D$ or $D = E = F$ is possible; the program will still function correctly.

Proof. Let t_R be the strand in type τ_R. We derive the output type of e^{DNA} under type assignment $\Gamma_{\mathcal{D}}$.

1. Firstly, we extend and augment the type assignment $\Gamma_{\mathcal{D}}$ with $\{(x, \tau'_R)\}$ and denote the extended and augmented type assignment Γ'. This pair may be added to Γ, because the let-statement introduces variable x with type τ_R. Strand t_R is non-mandatory in type τ_R and the main body of the program is situated in the else-part of the if-statement. By definition, we may thus augment the type of variable x to τ'_R in which strand t_R is mandatory.
2. $\Gamma' : \mathit{circularize}(x_s, A, F) \vdash \tau_c$. Strand t_R in τ'_R is mandatory, has first attribute A and last attribute F. Let c_R be the circularized version of the t_R. Type τ_c consists of strand c_R. Strand c_R is mandatory in type τ_c. The $\mathfrak{h}$-bit of type τ_c is set to *true*.
   From this point on, we use the type assignment $\Gamma_c = \Gamma' \cup \{(x_c, \tau_c)\}$.
3. $\Gamma_c : \mathit{blockselect}(x_c, B, C, a) \vdash \tau_y$. Type τ_c consists of a mandatory circular strand, strand c_R. On strand c_R attribute B is followed by attribute C. By Lemma 9.5, we know that type τ_y thus consists of strand c_R. Strand c_R is non-mandatory in type τ_y.
   From this point on, we use the type assignment $\Gamma_y = \Gamma_c \cup \{(y, \tau_y)\}$.
4. Γ_y : `if empty(`y`) then empty else` $\mathit{blockselect}(y, D, E, a) \vdash \tau_y$. Strand c_R in type τ_y is non-mandatory and circular. To type check the else-part of the if-statement, we may assign complex variable y type τ_c. Hence, by Lemma 9.5 we know that the output type of $\mathit{blockselect}(y, D, E, a)$ equals τ_y. Combining the empty type of the then-part with τ_y of the else-part, results again in τ_y.
5. $\Gamma_c : \bigcup_{a\in\Lambda}$ `let` $y := \mathit{blockselect}(x_c, B, C, a)$ `in`
   `if empty(`y`) then empty else` $\mathit{blockselect}(y, D, E, a) \vdash \tau_y$. All parts of the union over all atomic value symbols have the same type, namely, τ_y. Hence, the union has type τ_y.
   From this point on, we use the type assignment $\Gamma_f = \Gamma' \cup \{(f, \tau_y)\}$.
6. Γ_f : `cleanup(split(`$\mathit{blockfromto}(f, F, A), \#_4$`))` $\vdash \tau_R$. The *blockfromto* abbreviation constructs a new circular strand, call it c'_R, isomorphic to c_R except all nodes the nodes from A to F are blocked. Consequently, The only free node labeled $\#_4$ is the last node of the attribute-value block of attribute F. Hence, the split operation constructs a strand t'_R which is isomorphic to t_R except that all nodes from A to F are blocked. Because there is only one strand, the cleanup operation results in a strand isomorphic to t_R. Strand t_R is not mandatory in type τ_R.
7.

```
Γ′ :  if empty($x_s$) then empty else
        let $f$ :=
          let $x_c$ := $circularize(x_s, A, F)$ in
          $\bigcup_{a\in\Lambda}$ let $y$ := $blockselect(x_c, B, C, a)$ in
          if empty($y$) then empty else $blockselect(y, D, E, a)$ in
        cleanup(split($blockfromto(f, F, A), \#_4$))                    $\vdash \tau_R$
```

   Combining the empty type with type τ_R results in type τ_R.

8.

```
Γ' :  for $x_s := x$ iter i do
        if empty($x_s$) then empty else
        let $f :=$
          let $x_c := \mathit{circularize}(x_s, A, F)$ in
          $\bigcup_{a\in\Lambda}$ let $y := \mathit{blockselect}(x_c, B, C, a)$ in
          if empty($y$) then empty else $\mathit{blockselect}(y, D, E, a)$ in
        cleanup(split($\mathit{blockfromto}(f, F, A), \#_4$))                $\vdash \tau_R$
```

Complex variable x_s has type τ_R. The body of the for-statement has type τ_R. Hence, the for-statement has type τ_R.

□

10. Maximality and Tightness for Non-Atomic Expressions. We introduced the notions of maximality and tightness on arbitrary DNAQL expressions. However, Theorems 8.2 and 8.3 apply to atomic expressions only. In this section, we show that a maximal typing relation on DNAQL is undecidable, and that the typing relation is not tight for arbitrary expressions due to the interplay between union and the $\mathfrak{h}$-bit. An interesting future direction of research is to come up with a tight type relation or proving that a tight type relation is undecidable.

Let us first examine the maximality of a DNAQL typing relation. It is undecidable whether a relational algebra expression always outputs the empty relation [1]. Let e be a relational algebra expression. Expression e can be translated to an equivalent DNAQL expression e^{DNA} (as proven in Section 9). Let e_d be a DNAQL expression that is always defined, and let e_u be an expression that is undefined. For example, for e_d we can use the constant expression $\#_2$ and for e_u we can use $\texttt{block}(\#_2 \cup \overline{\#_2}, \#_2)$. We construct the expression

$$e' := \texttt{if empty}(e^{DNA}) \texttt{ then } e_d \texttt{ else } e_u$$

If the DNAQL type system would be maximal, expression e' would type check whenever expression e always outputs the empty relation. This is a contradiction as the emptiness problem is undecidable.

Secondly, we show by counterexample that the DNAQL typing relation is not tight on expressions. Consider the types shown in Figure 10.1. Both types have their $\mathfrak{h}$-bit, $\mathfrak{h}_1$ resp. $\mathfrak{h}_2$, equal to *true*. This implies that the nodes labeled a and $\overline{a}$, in τ_1, cannot be both present in a complex having type τ_1.

Now consider the expression $e = \texttt{hybridize}(\tau_1 \cup \tau_2)$. The type of $\tau_1 \cup \tau_2$ consists of the four components of τ_1 and τ_2. The components with the nodes labeled b and $\overline{b}$ are the mandatory components. Pivotal to this example is the $\mathfrak{h}$-bit of the union. The $\mathfrak{h}$-bit is set to *false*, as the respective weak types of τ_1 and τ_2 are mutually interacting (the node labeled b can match with the node labeled $\overline{b}$). Concretely, the output type of e consists of four components. The first component is mandatory and consists of two nodes, one labeled b, the other labeled $\overline{b}$. The nodes are matched. The second component is a node labeled a. The third component is a node labeled $\overline{a}$. The fourth component consists of two nodes, one labeled a, the other labeled $\overline{a}$. The nodes are matched. The $\mathfrak{h}$-bit of the output type is *true*.

Note however, that any two complexes C_1 and C_2 having type τ_1 resp. τ_2 can never produce a component having the fourth component as its type. Indeed, any complex C_1 having type τ_1 cannot have both the a- and $\overline{a}$-component, and any complex having type τ_2 cannot have either of the components.

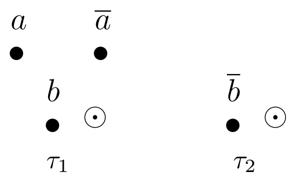


FIGURE 10.1. *Two types τ_1 and τ_2. The types consist of one-node components. Both types have their $\mathfrak{h}$-bit, $\mathfrak{h}_1$ resp. $\mathfrak{h}_2$, set to true.*

11. Discussion.

An interesting problem is to understand the precise expressive power of well-typed DNAQL programs. Theorem 9.1 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.

On the practical level, the obvious research direction is to verify some nontrivial 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 interesting topic for further research.

REFERENCES

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

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

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

[4] M. Arita, M. Hagiya, and A. Suyama. Joining and rotating data with molecules. In *ICEC*, 1997.

[5] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. A Type Sytem for DNAQL. In D. Stefanovic and A. Turberfield, editors, *DNA Computing and Molecular Programming, 18th International Conference*, volume 7433, pages 12–24. Springer, 2012.

[6] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. Graph-theoretic formalization of hybridization in DNA sticker complexes. In Cardelli and Shih [7], pages 49–63.

[7] L. Cardelli and W. Shih, editors. *DNA Computing and Molecular Programming, 17th International Conference, DNA17*, volume 6937 of *Lecture Notes in Computer Science.* Springer, 2011.

[8] 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.

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

[10] J. Van den Bussche and E. Waller. Polymorphic type inference of the relational algebra. *Journal of Computer and System Sciences*, 64:694–718, 2002.

[11] 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. Suppression subtractive hybridization: a method for generating differentially regulated or tissue-specific cDNA probes and libraries. *Proceedings of the National Academy of Sciences*, 93(12):6025–6030, 1996.

[12] R.M. Dirks and N.A. Pierce. Triggered amplification by hybridization chain reaction. *Proceedings of the National Academy of Sciences*, 101(43):15275–15278, 2004.

[13] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book.* Prentice Hall, 2009.

[14] J. Gillis and J. Van den Bussche. A Formal Model for Databases in DNA. In K. Horimoto, M. Nakatsui, and N. Popov, editors, *Algebraic and Numeric Biology*, volume 6479, pages 18–37. Springer, 2010.

[15] C.A. Gunter and J.C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.

[16] Q. Liu, L. Wang, A.G. Frutos, A.E. Condon, R.M. Corn, and L.M. Smith. DNA computing on surfaces. *Nature*, 403:175–179, 2000.

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

[18] Y. Papakonstaninou 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.

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

[20] B.C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[21] L. Qian, D. Soloveichik, and E. Winfree. Efficient Turing-universal computation with DNA polymers. In Y. Sakakibara and Y. Mi, editors, *Proceedings 16th International Conference on DNA Computing and Molecular Programming*, volume 6518 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2011.

[22] J.H. Reif. Parallel Biomolecular Computation: Models and Simulations. *Algorithmica*, 25:142–175, 1999.

[23] J.H. Reif, T.H. LaBean, M. Pirrung, V.S. Rana, B. Guo, C. Kingsford, and G.S. Wickham. Experimental construction of very large scale dna databases with associative search capability. In *Revised Papers from the 7th International Workshop on DNA-Based Computers: DNA Computing*, DNA 7, pages 231–247, London, UK, UK, 2002. Springer-Verlag.

[24] J. Sager and D. Stefanovic. Designing nucleotide sequences for computation: A survey of constraints. In A. Carbone and N. Pierce, editors, *DNA Computing*, volume 3892 of *Lecture Notes in Computer Science*, pages 275–289. Springer Berlin / Heidelberg, 2006.

[25] 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. A thermodynamic approach to designing structure-free combinatorial dna word sets. *Nucleic Acids Research*, 33(15):4965 – 4977, 2005.

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

[27] 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.

[28] M. Yamamoto et al. Development of DNA relational databases and data manipulation experiments. In C. Mao and T. Yokomori, editors, *Proceedings 12th International Meeting on DNA Computing*, volume 4287 of *Lecture Notes in Computer Science*, pages 418–427. Springer, 2006.