DNAQL: a query language for DNA sticker complexes

Peer-reviewed author version

BRIJDER, Robert; GILLIS, Joris & VAN DEN BUSSCHE, Jan (2021) DNAQL: a query language for DNA sticker complexes. In: Natural Computing, 20(1), p. 161-189.

DOI: 10.1007/s11047-020-09839-7
Handle: http://hdl.handle.net/1942/33464

DNAQL: A Query Language for DNA Sticker Complexes

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

Hasselt University, Belgium

Author version, 18 December 2020

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 define 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.

Keywords: DNA database, Hybridization, Type system, Sticker complex model, DNAQL

1 Introduction

Since Adleman's seminal experiment [2], the field of DNA computing has vastly grown, see the monographs [3, 27] and see [36, 29, 34, 40] for more recent developments. Computational models in the DNA computing field often aim for Turing-completeness. However, DNA computing also has a high potential for database applications. Indeed, the robust (almost indestructible) storage capacity of DNA [15, 21, 7] is very promising from the databases perspective. The potential use of single-stranded DNA as an addressable or searchable memory

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

is indeed well known [6, 31, 14, 37]. Databases, however, are much more than searchable memories: they are structured according to a logical data model such as the relational model, and are queried and manipulated using global operations on data such as the operations of the relational algebra.

Due to its expressive power, Turing-complete DNA computing models generally do not allow for a faithful implementation in the wetlab. The expressive power of query languages, such as the relational algebra, is however distinctly weaker than Turing-completeness. In this paper we develop a database query language using DNA, with the aim of having both practically and theoretically greater tractability than Turing-complete DNA computing models. In particular, special care has been taken to keep hybridization in check. We introduce the *sticker complexes model*, which consists of a number of operations defined on a restricted subclass of DNA complexes. 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 then hybridize with yet another copy and so forth. We identify this undesirable behavior as non-terminating hybridization. Non-terminating hybridization leads to sticker complexes unbounded in size. 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 to be produced. In this sense, the potential result of the hybridization is unbounded. Fortunately, in previous work [11] we have show that it is efficiently decidable for a sticker complex whether it has terminating hybridization.

In this paper we introduce the language *DNAQL*. Similar to languages such as SQL or the relational algebra that are familiar in the field of databases [19], 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. In a companion paper we show that DNAQL is expressive enough to simulate arbitrary relational algebra expressions, when representing relational databases

as DNA complexes [8]. The relational algebra is the applicative language at the core of standard database query languages such as SQL [16, 19, 1].

A difficulty with DNAQL, and with DNA computing in general, is that various manipulations of DNA must make certain assumptions on their input so as to be implementable in the wetlab 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 result is well known for Turing-complete programming languages, it also holds for database languages that are typically not Turing-complete [38].

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 [28, 22]. 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 [26]. 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. Furthermore, tightness mean (informally) that the type output of an operation cannot be slimmed down without jeopardizing the soundness of the type system.

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, containing selected results mostly without proofs, were presented at the ANB and DNA 18 conferences [20, 9], as well as in a keynote talk [10]. The present paper is a completely revised and final research report.

2 Related work

In one of the first papers on DNA computing, Reif already defined a formal data structure of DNA complexes [30]. Our data structures are simpler in an effort to avoid unrealistic or otherwise complicated and unmanageable secondary structures. (Reif avoids these by invoking an oracle for feasibility.) Our simplification is that single strands are either all-positive or all-negative, and moreover, negative strands have length at most two. The short negative strands can be thought of as stickers; thus the name "sticker complexes". Our previous work showed that the restrictions of sticker complexes do not preclude interesting database computations. An important feature of our model, which is lacking

in Reif's model, is the formal distinction between the structural content of a complex, and the complex as used in reactions, with multiples of each connected component present in surplus quantities.

The use of short stickers in DNA computing originates from [32], where stickers were used to turn bits on or off. We use stickers to bind strands together so that possibly complex secondary structures are formed.

The present work also fits in a recent trend of integrating formal methods (such as process calculi in computational systems biology [12]) with DNA computing [13, 24]. Yet the formalisms we use are different from process calculi and comprise mainly set theory, graph theory, and logic-based query languages. The computational power of hybridization in various models of formal languages has been intensively studied, see, e.g., [27, 39].

The paper [42], developed independently of the above mentioned extended abstract [20], has similar goals as the present paper: to perform the relational algebra in DNA by using elements of Reif's model [30]. A fundamental feature of the relational algebra is compositionality, i.e., the output of one operation may serve as the input of another. However, the model of [42] is not compositional. For example, the difference operator assumes the presence of the complements of the key values of a table (which is not present when the table is the output of another operation). Another fundamental difference is that the complications of possible nonterminating hybridization are not taken into account in [42].

In [41] an abbreviated account of achieving relational algebra operations through DNA manipulation is given. Unfortunately, that paper is too sketchy to allow any comparison with our approach. In contrast, the methods presented here are fully formalized, and importantly, identifies restrictions on DNA computing within which relational algebra simulation remains possible. More influential for this work is [5], which demonstrates how one can accomplish concatenation and rotation of DNA strands. Such manipulations, which involve circular DNA, are crucial in the DNA model presented here, and indeed were already crucial in [30].

We can conclude that the idea of performing relational algebra operations on DNA has been suggested repeatedly and independently over time by various researchers; a very recent example is the paper by Appuswamy et al. [4].

3 Sticker Complexes

In this section we define the notion of sticker complex, and notions related to sticker complexes.

3.1 Alphabet

From the outset we assume a finite alphabet Σ. As customary in formal models of DNA computing [27], this alphabet serves as an abstraction of a set of DNA codewords; see Section 5. 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. We also set $\bar{\bar{a}} = a$ so complementarity is symmetric. As usual, $\bar{a}$ 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 $\Theta = \{\#_i \mid i \in \{1, \ldots, 9\}\}$ of *tags*.

Justification for tags Tags will be used as markers and punctuations in our data structures, and serve various purposes. Tags numbered 2, 3, 4, 6 and 8 serve as markers for the split operation (Table 1 in Section 4). Indeed we have found that to do useful data manipulations, we need up to five distinct markers to indicate split points: three for single strands (with an operation that cleaves either *before* $\#_2$ or $\#_3$, or *after* $\#_4$), and two more for double strands. Here, our yardstick for expressive power is the ability to simulate relational algebra in DNA [10]. By having multiple distinct split points, rather than just one, we can restrict the locations at which the split operation cleaves. We consider it a feature that we do not need *more* than five, although it remains open whether the relational algebra can already be simulated using strictly less than five split markers. Detailed examples of the relational algebra simulation have been given in our conference papers [20, 10]. A full constructive proof showing that relational algebra can be simulated in DNAQL will be provided in a companion paper [8]. In the present paper we focus on the formal definitions of our data model and our language, and, not in the least, on the type system.

Moreover, tags 2, 3 and 4 will be used for the representation of relational data in DNA, where $\#_2$ and $\#_3$ will indicate the beginning of an attribute name and the attribute value, respectively, and $\#_4$ will indicate the end of the attribute/value pair.

Furthermore, we assume four more tags, numbered 1, 5, 7 and 9. These tags are needed in the relational algebra simulation to construct "stickers" used to concatenate strands. (Stickers are defined formally in Section 3.3 and their use for concatenation is illustrated in Example 6.1.) Tags 1 and 5 appear in the present paper only in examples; tags 7 and 9 are not mentioned further at all in the present paper, but play a crucial role in the "double bridging" construction that is used to simulate operations of the relational algebra. Again we consider it a feature that we need no more than four additional tages to construct the needed diversity of stickers. Certainly, having just a single sticker available would be too indiscriminatory and would form unwanted hybridizations; again, it remains open whether the relational algebra can already be simulated with strictly less tags than the nine we provide in the language DNAQL.

Finally, we note that $\#_5$ also plays a role in the implementation of the difference operation in DNA, as discussed in Section 5.

3.2 Pre-Complex

We define pre-complexes to contain the overall structure of sticker complexes (the definition is slightly modified with respect to extended abstract [20]: we use now node labels instead of edge labels.) 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 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 one 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 $\mathit{strands}(S)$ 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 Strands s and s' are *bonded* if there is a node v in s and a node v' in s' with $\{v, v'\} \in \mu$. When 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 $\mathit{comp}(C)$ 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, we sometimes denote $D \in \mathit{comp}(C)$ for a component D and a pre-complex C simply by $D \in C$.

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. (These important issues and notions are glossed over in Reif's formalization [30].)

Pre-complexes C_1 and C_2 are considered *isomorphic* if they are equal modulo the identity of the vertices. Consequently, an isomorphism from C_1 to C_2 should respect the labels of the vertices, the matching relation, immobilizations, etc. 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 of C_1 there is an isomorphic component D_2 of C_2. Pre-complexes C_1 and C_2 are *equivalent* if they subsume each other, denoted $C_1 \equiv C_2$. A component D of pre-complex C is *redundant* if there exists a component $D' \neq D$ of 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.

(a) (b)

Figure 1: An example of two pre-complexes that are non-isomorphic but that are equivalent.

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 Fig. 1.

3.3 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. 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. Such negative strands are called *stickers*.
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 without backtracking, following the chain or cycle shape of strands, and the partial matching μ. This search is in linear time, which yields an isomorphism check for single components in quadratic time (for a fixed node v of C, try all possible v' of C'). This algorithm then easily extends to complexes C and C' with multiple components, by matching the components of C to the components of C' — the complexity of the extended algorithm is in cubic time. This efficient isomorphism check is in contrast with the problem of general graph

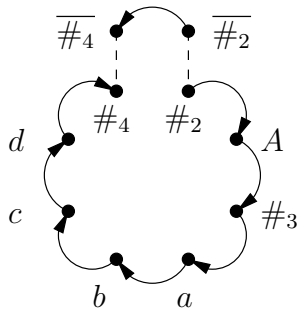

Figure 2: A sticker complex with one component. The positive strand has been circularized by a sticker.

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 consequence 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 often refer to sticker complexes simply as *complexes*, and to sticker complexes of dimension ℓ as *ℓ-complexes*.

Example 3.1. Fig. 2 shows a sticker complex with one component. The directed edges represent L. The dashed edges represent matchings in μ. The positive strand is being circularized by a sticker labeled by $\#_2$ and $\#_4$.

4 Operations on Sticker Complexes

In this section, we define a set of operations on complexes that are rather standard in the DNA computing literature, except perhaps for the difference, see Section 5. 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. The result of each operation is defined up to equivalence (cf. Subsection 3.2).

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 complexes. Without loss of generality we assume that V_1 and V_2 are disjoint. We define the *union* of C_1 and C_2, denoted by $C_1 \cup C_2$, as $(V_1 \cup V_2, L_1 \cup L_2, \lambda_1 \cup \lambda_2, \mu_1 \cup \mu_2, \iota_1 \cup \iota_2, \beta_1 \cup \beta_2)$.

Difference Let C_1 and C_2 be 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 $\#_5$.

We define the *difference* of C_1 and C_2, denoted by $C_1 - C_2$, as 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 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 it has no pair of mutually interacting nodes. In other words, C' is saturated if and only 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.

Let C and C' again be complexes. We call C' a *redundant variation* of C, simply if C' is subsumed by 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 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 4.1. *Let C_1 and C_2 be 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.*

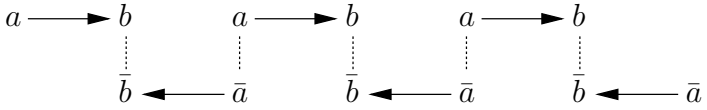

Figure 3: Illustration for Example 4.2.

A complex C' is called *unfinished* with respect to C if there exists a node u in C' and a node v in C such that u and v are mutually interacting; otherwise C' is called *finished* with respect to C. An MHE of a complex C that is finished with respect to C is called *saturated with respect* to complex C. Note that if C is saturated, then all MHEs are equivalent to C.

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

Example 4.2. 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 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. Different numbers n yield nonequivalent (non-isomorphic) MHE components, thus the number of potential MHE components is infinite.

Chemically, hybridization composes 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, then we define the *hybridization* of C, denoted by `hybridize`(C), as the disjoint union of a set S of mutually non-isomorphic finished MHE components of C such that each finished MHE component of C is isomorphic to some component in S. If C does not have a terminating hybridization, then 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

Table 1: The split points.

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

so that the (n_3, n_4) is added to L. We now define $\texttt{ligate}(C)$ as the complex obtained from C by filling all gaps.

Flush Quite simply $\texttt{flush}(C)$ is defined as 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) m in its strand, then (m, n) (resp. (n, m)) is removed from L.
- Furthermore, if there exists a node n' such that $\{n, n'\} \in \mu$ and n' has a successor (resp. predecessor) m' in its strand, then (n', m') (resp. (m', n')) is removed from L.

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

Block Here we assume that C is saturated; if C is not saturated then the block operation on C is considered to be undefined. We define the *block* operation on C with respect to $\sigma \in \Omega \cup \Theta$, denoted by $\texttt{block}(C, \sigma)$, as the complex obtained from C by adding all free nodes labeled with σ to β.

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

Let again $\sigma \in \Sigma$, and consider any contiguous substrand s of C. We call s a *σ-blocking range* if it satisfies the following two conditions. Firstly, all nodes of the substrand are free. Secondly, the last node of the substrand is labeled with σ. Now we define $\texttt{blockfrom}(C, \sigma)$ 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 we define $\texttt{blockexcept}(C, n)$ as the complex obtained from C by blocking, within each ℓ-vector $(e_0, e_1, \ldots, e_\ell, e_{\ell+1})$ that is not yet blocked, all nodes except e_n. If (C, n) does not satisfy the conditions above, then $\texttt{blockexcept}(C, n)$ is undefined.

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

4.1 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 it is shown that this is possible [11]. Next, we briefly recall the concepts and results relevant for the type system.

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

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;
- $E = \{\{v, w\} \subseteq V \mid v \text{ and } w \text{ are mutually interacting}\}$.

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 iff its hybridization graph does not contain any edges.

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 $(1 \leq i < n)$. When the path is alternating, it is said to be an *alternating cycle* when n is odd and at least 3, and $v_n = v_1$.

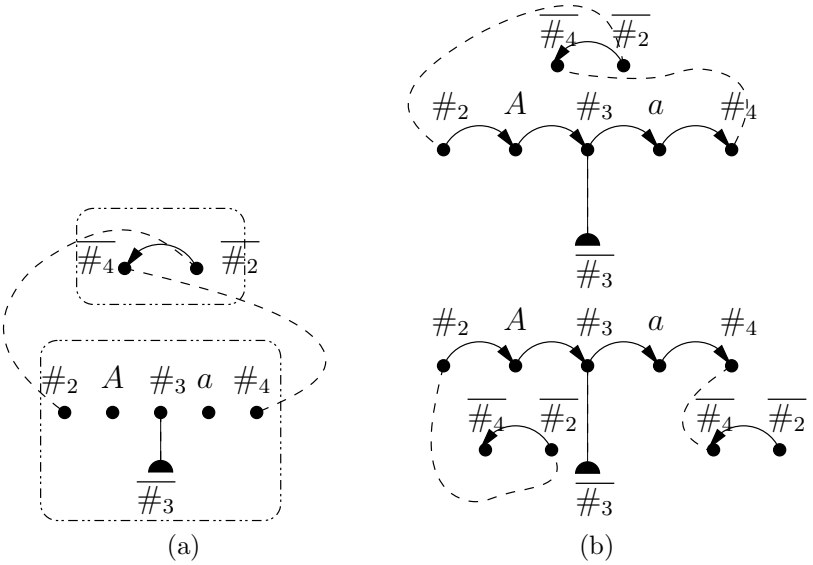

Figure 4: Hybridization graph of a sticker complex with one immobilized node.

In [11], it is shown that a complex C has non-terminating hybridization if and only if there is an alternating cycle in its hybridization graph. Although this result disregards immobilized components, the theorem is easily extended to include immobilized components:

Theorem 4.3. *A complex C has non-terminating hybridization if and only if there is an alternating cycle P in the hybridization graph of C, such that P does not pass through a block associated with an immobilized component.*

Example 4.4. Fig. 4 (a) shows the hybridization graph of a sticker complex with two components — immobilized nodes are decorated with the symbol ◠. The largest component has an immobilized node (the one labeled with $\overline{\#_3}$). 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.

Fig. 4 (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.

5 Implementation in DNA

In this section, we argue that the abstract sticker complexes and the operations on them presented above can be implemented in the wetlab. 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 [25, 33, 35]. 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 [23] 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 [18]. 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 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 $\#_3 v_1 \dots v_\ell \#_4$ (we represent here a strand by its string of labels), with $v_i \in \Lambda$ for $i = 1, \dots, \ell$, by introducing additional markers ϕ_i, so

that we get $\#_3\phi_1 v_1 \ldots \phi_\ell v_\ell \#_4$. 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 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 applications of DNAQL, and in particular in the simulation of the relational algebra [8], 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.

6 DNAQL

DNAQL 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 the basic set of operations on complexes introduced in Section 4. The language provides 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 Fig. 5. Note that expressions can contain two kinds of variables: variables standing for complexes, and counters, ranging from 1 to the dimension ℓ. 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 represent particular complexes. A string $w \in \Sigma^+$ represents a linear (positive) strand s where $w = \lambda(v_1) \cdots \lambda(v_n)$ and $(v_1, \ldots, v_n)$ is the unique path of s containing all vertices of s. A two-letter string $\bar{a}\bar{b}$, for $a, b \in \Sigma - \Lambda$, represents a sticker of the form $x \to y$ with $\lambda(x) = \bar{b}$ and $\lambda(y) = \bar{a}$ for some x and y. The expression `immob`$(\bar{a})$, for $a \in \Sigma$, stands for a negative, immobilized node labeled $\bar{a}$. If $\bar{a} \in \bar{\Lambda}$ we call such a node a *probe*. The expression `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 for e is a mapping from the free variables of e to ℓ-complexes; an ℓ-counter assignment is a mapping from the free counters of e to $\{1, \ldots, \ell\}$. Within such a context, the expression may evaluate to an

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

Figure 5: Syntax of DNAQL.

ℓ-complex, denoted by $[\![e]\!]^\ell(\nu, \gamma)$. Because the operations on complexes are not always defined, the evaluation may fail, so $[\![e]\!]^\ell(\nu, \gamma)$ may be undefined.

The semantic rules that define this evaluation are shown in Figure 6. The superscript ℓ has been omitted in the figure to reduce clutter. The bulk of the evaluation rules simply apply the operations defined in Section 4. The rules for `let` and `for` define these constructs formally. In these rules we use the oft-used notation $f[x := u]$ to denote the function obtained from f by adding the mapping of x to u. When e is a program, we denote $[\![e]\!]^\ell(\nu, \emptyset)$ simply by $[\![e]\!]^\ell(\nu)$.

Example 6.1. 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.

$$
\begin{array}{l}
\texttt{let}\ y_1 := \texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(x_1 \cup \texttt{immob}(\bar{a})))\ \texttt{in} \\
\texttt{let}\ y_2 := \texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(x_2 \cup \texttt{immob}(\bar{b})))\ \texttt{in} \\
\texttt{if empty}(y_1)\ \texttt{then empty else} \\
\texttt{if empty}(y_2)\ \texttt{then empty else} \\
\texttt{cleanup}(\texttt{ligate}(\texttt{hybridize}(y_1 \cup y_2 \cup \overline{\#_5\#_1})))
\end{array}
$$

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 and C_2 the strands whose data entry contains the letter a and b, respectively; if both intermediate results are nonempty, then the program uses the stickers $\overline{\#_5\#_1}$ to concatenate each remaining strand from C_1 with each remaining strand from C_2.

7 Sticker Complex Types

Intuitively, a "weak" 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* occur in the complex, with attribute names and tags explicit, but the dimension and actual values of data entries hidden. In order to

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6: DNAQL Semantics

obtain a powerful type-checking algorithm for DNAQL, these "weak" types S are augmented to obtain "strong" types that also indicate the mandatory components $\odot$, which *must* occur, and a bit $\mathfrak{h}$ indicating whether all the complexes of the 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 (which require saturation to be defined).

7.1 Definition

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

1. $*$ (*unblocked*) 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 by $\Sigma_N = \Omega \cup \Theta \cup N$.

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

A *weak sticker complex type* (or *weak type* for short) is very similar to a sticker complex; it is a structure $S = (V, L, \lambda, \mu, \iota, \beta)$ that satisfies the same definition as that of a sticker complex 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; and
- there are no redundant components (recall the definition of redundancy from Section 3).

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; and
- r is of type $\hat{*}$ if all nodes of r but one belong to β and r is flanked by closed nodes (i.e., on both sides adjacent, w.r.t. L, to closed nodes).

Now we say that C is *well typed* if

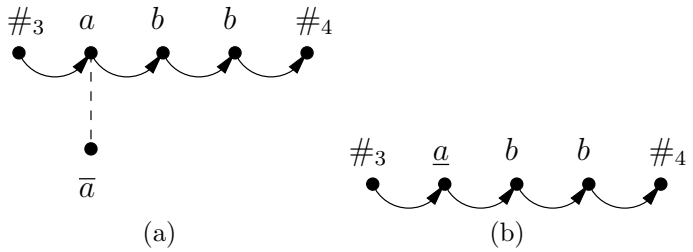

Figure 7: Two ill-typed complexes.

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

Example 7.1. Fig. 7 shows two ill-typed (i.e., not well-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 $\mathit{stype}(C)$ as the weak type obtained by:

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

The definitions of subsumption, mutually interacting, and saturated for sticker complexes, defined in Section 3, are adopted to weak types in the natural way. We have the following lemma, which henceforth will be used without mention.

Lemma 7.2. *Let C_1 and C_2 be well-typed sticker complexes. If $C_1 \sqsubseteq C_2$, then $\mathit{stype}(C_1) \sqsubseteq \mathit{stype}(C_2)$. If $\mathit{stype}(C_1) \sqsubseteq \mathit{stype}(C_2)$ and $\mathit{stype}(C_1)$ does not contain nodes labeled with 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 $\mathit{stype}(c)$ is isomorphic to $\mathit{stype}(c')$. If $\mathit{stype}(C_1) \sqsubseteq \mathit{stype}(C_2)$ and $\mathit{stype}(C_1)$ does not contain nodes labeled with symbols of N, then $C_1 \equiv \mathit{stype}(C_1) \sqsubseteq \mathit{stype}(C_2)$. Let $c \in \mathit{comp}(C_1)$, then there is a $c' \in \mathit{comp}(\mathit{stype}(C_2))$ 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. □

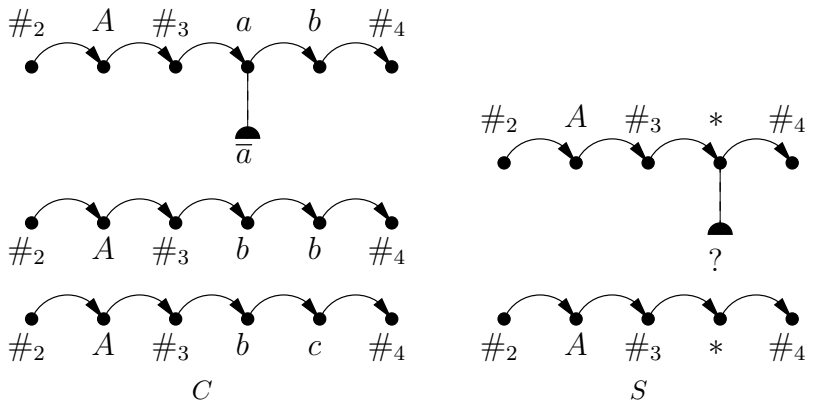

Figure 8: A sticker complex C and a weak type S where C has weak type S.

For a well-typed sticker complex C and a sticker complex type S, we now say that C *has weak type* S, denoted by $C : S$, if *stype*(C) is subsumed by S. For sticker complex C, *stype*(C) is the "smallest" weak type, in the sense that for every weak type S' such that $C : S'$, it holds that S is subsumed by S'.

Example 7.3. Fig. 8 shows a sticker complex C of dimension 2, and a weak 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. Weak type S is *stype*(C) and is thus the smallest type for C.

Lemma 7.4. *A weak type S is saturated if and only if all complexes having weak type S are saturated.*

Proof. We first prove the if direction. Assume S is not saturated, and let u and v be nodes of S that are mutually interacting. If the labels of u and v are not in $\{*, \underline{*}, \hat{*}, ?\}$, then any sticker complex C with *stype*(C) isomorphic to S has corresponding mutually interacting nodes u' and v'. Alternatively, one of u and v, say u, has label ? and the other has a label in $\{*, \hat{*}\}$. In this case choose a sticker complex C with *stype*(C) isomorphic to S such that the ℓ-core corresponding to u has a free node with label complementary to the label of the node v' corresponding to v. Consequently, u' and v' are mutually interacting. In any case, sticker complex C has type S and is not saturated.

We now prove the only-if direction. Assume there is a sticker complex C of weak type S such that C is not saturated. Then there are nodes u and v of C that are mutually interacting. Consequently, *stype*(C) has nodes that are mutually interacting and since *stype*$(C) \sqsubseteq S$, so does S. Thus, S is not saturated. □

A weak type is "weak", in the sense that for any well-typed sticker complex C of weak type S, C is also of weak type S', where S is subsumed by S'. In particular, the empty sticker complex is of every weak type. This is too weak to type nontrivial DNAQL programs involving hybridization, where we need

Figure 9: A weak type having two single-node components.

to know about components that are sure to be present. We now introduce the notion of a "strong" type which can place further restrictions on sticker complexes. A *strong sticker complex type* (or *type* for short) τ is a triple $(S, \odot, \mathfrak{h})$, where S is a weak type, $\odot$ is a weak type subsumed by S, $\mathfrak{h}$ is a boolean, and moreover if $\mathfrak{h} = \mathit{true}$, then $C \cup \odot$ is saturated for all $C \in \mathit{comp}(S)$. Then 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* (or *hybridization bit*) *of* τ.

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

Example 7.5. Consider the weak type S shown in Fig. 9. Let $\mathfrak{h} = \mathit{true}$ and let $\odot$ be the weak type consisting of the component of S depicted on the left-hand side of Figure 9. Then $(S, \odot, \mathfrak{h})$ is not a type since there is a $c \in \mathit{comp}(S)$ (it is the component on the right-hand side of Figure 9), such that $c \cup \odot$ is not saturated.

Note that the saturation condition in the definition of a type avoids "garbage" components. Indeed, if we omitted this condition, then any complex having type τ cannot contain the component on the right-hand side of Figure 9, because such a complex will not be saturated.

Example 7.6. 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 Fig. 9. Let $\tau = (S, \texttt{empty}, \mathit{true})$. There are three (mutually non-isomorphic) 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.

From now on, in graphical depictions of types, mandatory components are indicated by the symbol $\odot$.

Example 7.7. Consider the type τ on the left-hand side of Fig. 10 — the $\mathfrak{h}$-bit of τ is irrelevant in this case. Although both components of τ are mandatory, we will see in Section 8 that the hybridization τ' of τ consists of three non-mandatory components, see the right-hand side of Fig. 10. Let $\tau' = (S, \odot, \mathfrak{h})$. The $\mathfrak{h}$-bit of τ' is *true*. This has important repercussions on the set of complexes having this type. Indeed, consider the complex in Fig. 11. This complex does *not* have type τ', but it has type $(S, \odot, \mathit{false})$.

The definition of a saturated type is semantic. We now show that we can efficiently and syntactically decide whether or not a type is saturated.

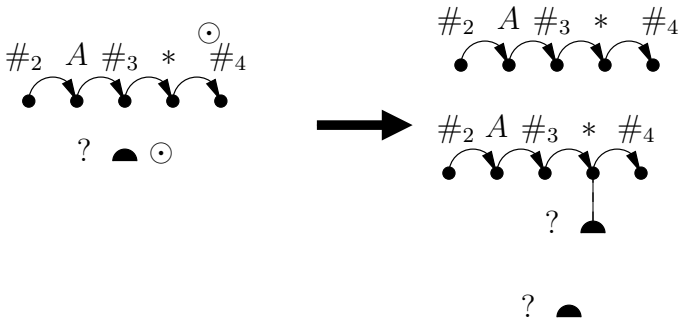

Figure 10: A type with two mandatory components on the left. On the right is the hybridization of the type on the left. Despite the fact that all components start as mandatory, the hybridization contains only non-mandatory components.

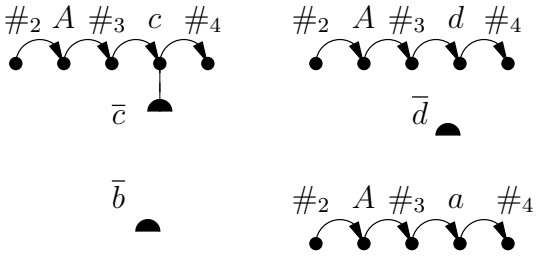

Figure 11: A complex with five components. This complex does not have the type on the right of Fig. 10.

Lemma 7.8. *Type $\tau = (S, \odot, \mathfrak{h})$ is saturated if and only if S is saturated or $\mathfrak{h} = \mathit{true}$.*

Proof. We first prove the if direction. Let $C \in [\![\tau]\!]$. Since C is of weak type S, we have by Lemma 7.4 that if S is saturated, then so is C. If $\mathfrak{h} = \mathit{true}$, then C is saturated by the definition of $C : \tau$.

We now prove the only-if direction. Suppose that S is not saturated and $\mathfrak{h} = \mathit{false}$. Since S is not saturated, by Lemma 7.4, there is a complex C having weak type S that is not saturated. Let C' be a complex with $\mathit{stype}(C') \equiv \odot$. Then $C'' = C \cup C'$ is not saturated and $\odot \sqsubseteq \mathit{stype}(C'') \sqsubseteq S$. Since $\mathfrak{h} = \mathit{false}$, we have that $C'' \in [\![\tau]\!]$. Hence τ is not saturated. □

Note that, as a consequence of Lemma 7.8, saturatedness of a type is decidable in polynomial time.

7.2 Subtypes

A desirable property of types is that they are *inhabited*, i.e., for every type τ, the set $[\![\tau]\!]$ is non-empty. This follows from the next lemma.

Lemma 7.9. *Let $\tau = (S, \odot, \mathfrak{h})$ be a type. For every $D \in \mathit{comp}(S)$, any complex C with $\mathit{stype}(C) \equiv \odot \cup D$ has type τ. In particular, any complex C with $\mathit{stype}(C) \equiv \odot$ has type τ.*

Proof. If $\mathfrak{h} = \mathit{false}$, then any complex C with $\odot \sqsubseteq \mathit{stype}(C) \sqsubseteq S$ is of type τ. If $\mathfrak{h} = \mathit{true}$, by the definition of a type, the weak type $\odot \cup D$ is saturated. Consequently, C is saturated and so C has type τ. In particular, if $D \in \mathit{comp}(\odot)$, then $\mathit{stype}(C) \equiv \odot$ is saturated. The corner case where $\mathit{comp}(\odot) = \emptyset$ also holds, since the empty weak type is saturated by definition. □

Let τ and τ' be 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'. Types τ and τ' are called *equivalent* if $\tau \preceq \tau'$ and $\tau' \preceq \tau$.

Example 7.10. Recall the type $\tau = (S, \odot, \mathit{true})$ on the right-hand side of Fig. 10. Let $\tau' = (S, \odot, \mathit{false})$. We have that $\tau \preceq \tau'$ but not $\tau' \preceq \tau$, because the complex shown in Fig. 11 has type τ' but does not have type τ.

The notion of subtyping is defined semantically. Moreover, a type can have an infinite number of complexes. An efficiently decidable syntactic characterization of subtyping is thus called for. Proposition 7.11 provides such a characterization.

Proposition 7.11. *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. This statement also holds if we replace the condition $\mathfrak{h}' = \mathit{true}$ by τ' is saturated.*

Proof. First, we prove the only-if direction. Assume $\tau \preceq \tau'$. We now verify each of the three conditions.

(i) Suppose that $S \not\sqsubseteq S'$. 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 does not have type τ', because D is not a component of S.

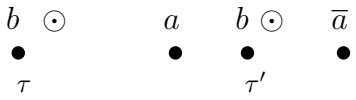

Figure 12: Types $\tau = (S, \odot, \mathit{false})$ and $\tau' = (S', \odot', \mathit{true})$ with $\tau \preceq \tau'$.

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

(iii) If $\mathfrak{h}' = \mathit{true}$, then τ' is saturated and therefore also $\tau \preceq \tau'$.

We now consider the if direction. Let C be a complex having type τ. We easily verify 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.

If S' is saturated, then $S \sqsubseteq S'$ is saturated and, by Lemma 7.8, so is τ. By again using Lemma 7.8 we observe that we thus may replace the condition $\mathfrak{h}' = \mathit{true}$ in the proposition by τ' is saturated. □

Lemma 7.8 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 7.11.

Corollary 7.12. *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. It suffices to show that condition (iii) is equivalent to [$\mathfrak{h}' = \mathit{true}$ implies that τ saturated] $\wedge$ [$\mathfrak{h} = \mathit{true}$ implies that τ' saturated]. By Lemma 7.8, if $S \equiv S'$ is saturated then both conditions hold trivially, and if $S \equiv S'$ is not saturated then both conditions reduce to $\mathfrak{h} = \mathfrak{h}'$. □

Obviously, type $\tau = (S, \odot, \mathit{true})$ is a subtype of the type $\tau' = (S, \odot, \mathit{false})$. On the other hand, by Corollary 7.12, $\tau' = (S, \odot, \mathit{false})$ may also be a subtype of τ when S is saturated.

Example 7.13. Fig. 12 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 complex.

Lemma 7.14. *Let C be a complex and τ a type. Then C is of type τ iff $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C) \preceq \tau$ with $\mathfrak{h}_C = \mathit{true}$ iff C is saturated.*

Proof. Let $\tau = (S, \odot, \mathfrak{h})$. By Proposition 7.11, $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C) \preceq \tau$ iff (1) $\odot \sqsubseteq \mathit{stype}(C) \sqsubseteq S$ and (2) if $\mathfrak{h} = \mathit{true}$, then $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C)$ is saturated. Now, by Lemma 7.8, $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C)$ is saturated iff $\mathit{stype}(C)$ is saturated or C is saturated. If $\mathit{stype}(C)$ is saturated, then C is saturated. Hence, $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C)$ is saturated iff C is saturated. By definition, $C : \tau$ iff (1) $\odot \sqsubseteq \mathit{stype}(C) \sqsubseteq S$ and (2) if $\mathfrak{h} = \mathit{true}$, then C is saturated — so the lemma follows. □

Figure 13: 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, $\tau_1 \vee \tau_2 = (S_1 \cup S_2, \texttt{empty}, \mathit{true})$ allows the empty complex, whereas the empty complex is not part of $[\![\tau_1]\!]$ or $[\![\tau_2]\!]$.

7.3 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 and S' be weak types. The *intersection* of S and S' is the weak type formed by the components of S that are isomorphic to some component of S'. We denote the intersection of S and S' by $S \cap S'$.

Proposition 7.15. *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 the type $(S_1 \cup S_2, \odot_1 \cap \odot_2, \tau_1$ saturated $\wedge\, \tau_2$ saturated$)$.*

Proof. Let $\tau = (S, \odot, \mathfrak{h})$ with $S = S_1 \cup S_2$, $\odot = \odot_1 \cap \odot_2$, and $\mathfrak{h} = \tau_1$ saturated $\wedge$ τ_2 saturated.

First, observe that τ is a type. Indeed, $\odot_1 \cap \odot_2 \sqsubseteq \odot_1 \sqsubseteq S_1 \sqsubseteq S_1 \cup S_2$ and if $\mathfrak{h} = \mathit{true}$, we must show that for all $C \in \mathit{comp}(S)$ it holds that $\odot \cup C$ is saturated. Let $C \in \mathit{comp}(S)$. Then $C \in \mathit{comp}(S_i)$ for some $i \in \{1, 2\}$. If $\mathfrak{h} = \mathit{true}$, then τ_i is saturated. By Lemma 7.8, S_i is saturated or $\mathfrak{h}_i = \mathit{true}$. If S_i is saturated, then so is $\odot \cup C \sqsubseteq \odot_i \cup C \sqsubseteq S_i$. If $\mathfrak{h}_i = \mathit{true}$, then $\odot_i \cup C$ is saturated and so is $\odot \cup C \sqsubseteq \odot_i \cup C$.

Now we must show that τ is the *least* upper bound. Let $\tau' = (S', \odot', \mathfrak{h}')$ be a type. Then τ' is an upper bound of τ_1 and τ_2 iff $\tau_i \preceq \tau'$ for $i \in \{1, 2\}$. By Proposition 7.11, $\tau_i \preceq \tau'$ iff $S_i \sqsubseteq S'$, $\odot' \sqsubseteq \odot_i$ and if $\mathfrak{h}' = \mathit{true}$, 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 $\mathfrak{h} = \mathit{true}$, then τ_1 is saturated and τ_2 is saturated. Hence, by Proposition 7.11, τ' is an upper bound of τ_1 and τ_2 iff $\tau \preceq \tau'$. □

For types τ_1 and τ_2, we have $[\![\tau_1]\!] \cup [\![\tau_2]\!] \subseteq [\![\tau_1 \vee \tau_2]\!]$. The converse inclusion, however, does not hold in general. Indeed, consider the types τ_1 and τ_2 shown in Fig. 13. The empty complex is in $[\![\tau_1 \vee \tau_2]\!]$ but not in $[\![\tau_1]\!]$ or $[\![\tau_2]\!]$, because both types have a non-empty mandatory type.

7.4 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 7.16. *Let $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ be types.*

A lower bound of τ_1 and τ_2 exists if and only if both (1) $\odot_1 \cup \odot_2 \sqsubseteq S_1 \cap S_2$ and (2) if 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 exists a greatest lower bound equivalent to

$$\tau_g = (S_1 \cap S_2 - Z, \odot_1 \cup \odot_2, \tau_1 \text{ saturated} \vee \tau_2 \text{ saturated}),$$

where $Z = \{C \in comp(S_1 \cap S_2) \mid C \cup \odot_1 \cup \odot_2$ is not saturated$\}$ if τ_1 or τ_2 is saturated, and $Z = \emptyset$ otherwise.

Proof. Let $\tau = (S, \odot, \mathfrak{h})$ be a type. Then, by Proposition 7.11, $\tau \preceq \tau_i$ iff $S \sqsubseteq S_i$, $\odot_i \sqsubseteq \odot$, and if τ_i is saturated, then τ is saturated. Hence, τ is a lower bound of τ_1 and τ_2 iff (1) $S \sqsubseteq S_1 \cap S_2$, (2) $(\odot_1 \cup \odot_2) \sqsubseteq \odot$, and (3) if τ_1 or τ_2 is saturated, then τ is saturated.

Since $\odot \sqsubseteq S$, the existence of a lower bound implies that $\odot_1 \cup \odot_2 \sqsubseteq S_1 \cap S_2$. Also, if τ is saturated, then $\odot$ is saturated and therefore so is $(\odot_1 \cup \odot_2) \sqsubseteq \odot$. Thus, the existence of a lower bound also implies that if τ_1 or τ_2 is saturated, then $\odot_1 \cup \odot_2$ is saturated.

Conversely, if the two conditions stated in the proposition hold, then τ_g is a type satisfying the above three conditions for being a lower bound. Indeed, this is clear if τ_1 and τ_2 are not saturated. Assume that τ_1 or τ_2 is saturated. Then $\odot_1 \cup \odot_2$ is saturated. Consequently, $\odot_1 \cup \odot_2 \sqsubseteq S_1 \cap S_2 - Z$ and for every $C \in comp(S_1 \cap S_2 - Z)$, we have that $C \cup \odot_1 \cup \odot_2$ is saturated. So τ_g is a type. We easily observe that the three conditions for being a lower bound hold for τ_g (the last one holds since the $\mathfrak{h}$-bit of τ_g is *true* in this case and so τ_g is saturated).

We finally show that lower bound τ_g is a greatest lower bound. Let τ be such that $\tau_g \preceq \tau$. Then $S_1 \cap S_2 - Z \sqsubseteq S$, $\odot \sqsubseteq \odot_1 \cup \odot_2$, and if τ is saturated, then τ_g is saturated. If τ is moreover a lower bound of τ_1 and τ_2, then the above three conditions hold and so $S_1 \cap S_2 - Z \sqsubseteq S \sqsubseteq S_1 \cap S_2$, and $\odot \equiv \odot_1 \cup \odot_2$, and τ_g is saturated iff τ is saturated. If τ is saturated, then, for all $C \in comp((S_1 \cap S_2) \cap Z)$, we have that $C \cup \odot$ is not saturated and so $S_1 \cap S_2 - Z \equiv S$. Thus τ and τ_g are equivalent, and so τ_g is a greatest lower bound. □

Example 7.17. Types τ_1 and τ_2 from Fig. 13 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.

8 Operations on Sticker Complex Types

In Section 4, 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. Thereto we now proceed to define the operations also on the type level. The result of each operation below is defined up to equivalence.

Let us first introduce some common notations. In the following definitions, propositions, and proofs, the symbols τ, τ', τ_1, and τ_2, invariably stand for

arbitrary types $(S, \odot, \mathfrak{h})$, $(S', \odot', \mathfrak{h}')$, $(S_1, \odot_1, \mathfrak{h}_1)$, and $(S_2, \odot_2, \mathfrak{h}_2)$, respectively. Also, for any $a \in \Lambda$ and natural number ℓ, the notation a^ℓ invariably stands for a sequence of ℓ nodes labeled a.

8.1 Union

We define the *union* of τ_1 and τ_2, denoted by $\tau_1 \cup \tau_2$, as $(S_1 \cup S_2, \odot_1 \cup \odot_2, \mathfrak{h})$, where $\mathfrak{h} = \mathit{true}$ iff both (1) there are no nodes u in S_1 and v in S_2 such that u and v are mutually interacting, and (2) τ_1 and τ_2 are saturated.

Note that $\tau_1 \cup \tau_2$ is a type since for all $C \in \mathit{comp}(S_i)$, with $i \in \{1, 2\}$, $C \cup \odot_i$ is saturated, and thus $C \cup \odot_1 \cup \odot_2$ is saturated by condition (1) when $\mathfrak{h} = \mathit{true}$.

Proposition 8.1. *If $C_1 : \tau_1$ and $C_2 : \tau_2$, then $C_1 \cup C_2 : \tau_1 \cup \tau_2$.*

Proof. Let $C = C_1 \cup C_2$ and $\tau_1 \cup \tau_2 = (S, \odot, \mathfrak{h})$. We verify the three conditions in the definition of a complex having a particular type.

We first verify that $\mathit{stype}(C) \sqsubseteq S$. Let $D \in \mathit{comp}(C)$. Then $D \in \mathit{comp}(C_1)$ or $D \in \mathit{comp}(C_2)$. Consequently, $\mathit{stype}(D)$ is subsumed by S_1 or by S_2, and thus by $S_1 \cup S_2 = S$.

We now verify that $\odot \sqsubseteq \mathit{stype}(C)$. 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 $\mathit{stype}(C)$.

Finally, assume that $\mathfrak{h} = \mathit{true}$. Assume to the contrary that C is not saturated. Let u and v be mutually interacting nodes of C. Since $\mathfrak{h} = \mathit{true}$, τ_1 and τ_2 are both saturated and so u and v do not both belong to C_1 or to C_2. Assume that u belongs to C_1 and v belongs to C_2. 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 mutually interacting nodes u'' and v'' of $S_1 \cup S_2$, where u'' belongs to S_1 and v'' to S_2. However $\mathfrak{h} = \mathit{true}$ implies that such nodes do not exist (see the definition of $\tau_1 \cup \tau_2$) — a contradiction. □

The next proposition shows that $\tau_1 \cup \tau_2$ is the most restrictive type that satisfies Proposition 8.1.

Proposition 8.2. *If $C_1 \cup C_2 : \tau$ for all complexes $C_1 : \tau_1$ and $C_2 : \tau_2$, then $\tau_1 \cup \tau_2 \preceq \tau$.*

Proof. Let $\tau_1 \cup \tau_2 = (S_\cup, \odot_\cup, \mathfrak{h}_\cup)$. To show that $\tau_1 \cup \tau_2 \preceq \tau$, we verify the three conditions of Proposition 7.11.

1. Proof of $S_\cup \sqsubseteq S$. Let $D \in \mathit{comp}(S_\cup)$. If $D \in \mathit{comp}(S_1)$, let C_1 be a complex such that $\mathit{stype}(C_1) \equiv (\odot_1 \cup D)$ and let C_2 be a complex such that $\mathit{stype}(C_2) \equiv \odot_2$. By Lemma 7.9, complex C_1 has type τ_1 and complex C_2 has type τ_2. Since $C_1 \cup C_2 : \tau$ we have $D \in \mathit{comp}(S)$.

2. Proof of $\odot \sqsubseteq \odot_\cup$. Let $D \in \mathit{comp}(S)$ such that $D \not\sqsubseteq \odot_\cup$. We show that $D \not\sqsubseteq \odot$. Let, for $i \in \{1, 2\}$, complex C_i be such that $\mathit{stype}(C_i) \equiv \odot_i$. Then complexes C_1 and C_2 are of types τ_1 and τ_2, respectively. Because $D \not\sqsubseteq \odot_\cup$, we have $D \not\sqsubseteq \odot_1$ and $D \not\sqsubseteq \odot_2$. Hence complex $C_1 \cup C_2$ does not contain a component of weak type D. Thus, $D \not\sqsubseteq \odot$.

3. Proof of $\mathfrak{h} = \mathit{true}$ implies that $\tau_1 \cup \tau_2$ is saturated. Assume that $\tau_1 \cup \tau_2$ is not saturated. We show that $\mathfrak{h} = \mathit{false}$. Since $\tau_1 \cup \tau_2$ is not saturated, $\mathfrak{h}_\cup = \mathit{false}$. Thus, by definition of $\tau_1 \cup \tau_2$, (a) S_1 and S_2 are mutually interacting, or (b) τ_1 or τ_2 are not saturated.

   (a) Suppose that S_1 and S_2 are mutually interacting. 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. Since $C_1 \cup C_2 : \tau$ and $C_1 \cup C_2$ is not saturated, we have $\mathfrak{h} = \mathit{false}$.

   (b) Suppose, without loss of generality, that τ_1 is not saturated. Hence there is an unsaturated complex C of type τ_1. Let C' be a complex of type τ_2. Since $C \cup C' : \tau$ and $C \cup C'$ is not saturated, we have $\mathfrak{h} = \mathit{false}$.

□

8.2 Difference

Assume the weak types $S_i = (V_i, L_i, \lambda_i, \mu_i, \iota_i, \beta_i)$ underlying τ_i, for $i = 1, 2$, satisfy the following conditions:

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 below is undefined.

Let T_1 (T_2, resp.) be the weak type consisting of all strands in $\odot_1$ (S_1, resp.) that do not have an isomorphic copy in S_2 ($\odot_2$, resp.). Let $\mathit{data}(S_1)$ be the set of strands in S_1 having a $*$-labeled node. We define the *difference* of τ_1 and τ_2, denoted by $\tau_1 - \tau_2$, as $\big(\mathit{data}(S_1) \cup T_2, T_1, \mathit{true}\big)$. Note that $\tau_1 - \tau_2$ is a type, since $T_1 \sqsubseteq T_2$ and $\mathit{data}(S_1) \cup T_2$ is saturated because all components are positive, noncircular strands.

Example 8.3. Fig. 14 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 2. On the type-level, the cases $C_1 - C_1$ and $C_1 - C_2$ are indistinguishable, however, the resulting complexes are 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 $\#_2 A \#_3 a \#_4$. 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.

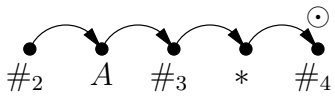

Figure 14: A hybridized type with a single mandatory component.

Table 2: Two complexes having the type depicted in Fig. 14.

C_1	C_3
$\#_2 A \#_3 a \#_4$	$\#_2 A \#_3 b \#_4$
$\#_2 A \#_3 b \#_4$	$\#_2 A \#_3 c \#_4$

Proposition 8.4. *$\tau_1 - \tau_2$ is defined if and only if for all complexes C_1 of type τ_1 and C_2 of type τ_2, we have that $C_1 - C_2$ is defined. In this case, $C_1 - C_2 : \tau_1 - \tau_2$.*

Proof. First assume that $\tau_1 - \tau_2$ is defined. Let C_1 be a complex of type τ_1 and C_2 be a complex of type τ_2. We prove that $C_1 - C_2$ is defined by showing that each of the three input restrictions in the definition of difference for complexes 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 thus conclude that $C_1 - C_2$ is well defined.

Conversely, assume that $\tau_1 - \tau_2$ is not defined. We show that there are complexes C_1 of type τ_1 and C_2 of type τ_2 such that $C_1 - C_2$ is not defined. Assume that $\tau_1 - \tau_2$ is not defined. Then one of the three conditions in the definition of difference on types is not satisfied. We consider each of these cases separately.

1. Suppose there is a node x that is matched, immobilized, blocked or labeled with $\underline{*}$ or $\hat{*}$ in τ_1 (the proof is similar if x is in 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 a blocked node in case of $\hat{*}$). Let D be the component of S_1 containing x. Let C_1 be a complex such that $\mathit{stype}(C_1) \equiv D \cup \odot_1$. Complex C_1 has type τ_1 by Lemma 7.9 and therefore has a matching, blocking, or immobilization. So the difference $C_1 - C_2$ is not defined for any complex C_2 of type τ_2 (such a complex C_2 exists by Lemma 7.9).

2. This case will be split into two subcases: (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, strands s_1 and s_2 in $\mathit{strands}(S_1) \cup \mathit{strands}(S_2)$ are of different lengths or have a different number of $*$-labeled nodes.

   (a) Let d be a strand in S_i for some $i \in \{1, 2\}$ that is negative or circular, and let D be the component in which d occurs. Let C_i be a complex with $\mathit{stype}(C_i) \equiv \odot_i \cup D$. Hence, complex C_i is of type τ_i by Lemma 7.9. Moreover, C_i 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). We distinguish the following cases, and exhibit, as claimed, two strands having respective types s_1 and s_2, of different lengths, so that their difference is not defined:

      i. If $n(s_1) \neq n(s_2)$ and $a(s_1) = a(s_2)$, then clearly the lengths $n(s_1) + (\ell - 1)a(s_1)$ and $n(s_2) + (\ell - 1)a(s_2)$ are different.

      ii. Otherwise, $a(s_1) \neq a(s_2)$. Then 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 $\ell-1 > \max\{n(s_2)-n(s_1)\}$, 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 either containing a node labeled $\#_5$ or not ending with a node labeled $\#_4$. Let C_1 be a complex having type τ_1. Let C_2 be a complex with $\mathit{stype}(C_2) \equiv \odot_2 \cup D$. By definition, C_2 has type τ_2 and has a strand that either contains a node labeled $\#_5$ or does not end with a node labeled $\#_4$. Hence, $C_1 - C_2$ is undefined.

Next, we prove that $C = C_1 - C_2$ is of type $\tau = \tau_1 - \tau_2$. Let $\tau = (S, \odot, \mathfrak{h})$.

We first verify that $\mathit{stype}(C) \sqsubseteq S$. 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 $*$. 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.

We now verify that $\odot \sqsubseteq \mathit{stype}(C)$. Let $s \in \mathit{comp}(\odot)$. By the definition of $\odot$, $s \sqsubseteq \odot_1$ and $s \not\sqsubseteq S_2$. Since $s \sqsubseteq \odot_1$, we have $s \equiv \mathit{stype}(D)$ for some $D \in \mathit{comp}(C_1)$. Since $\mathit{stype}(D) \equiv s \not\sqsubseteq S_2$ and $\mathit{stype}(C_2) \sqsubseteq S_2$, we have by the first part of Lemma 7.2 that $D \not\sqsubseteq C_2$. Thus $D \sqsubseteq C_1 - C_2 = C$ and again by the first part of Lemma 7.2 we have $s \equiv \mathit{stype}(D) \sqsubseteq \mathit{stype}(C)$ as desired.

Finally, since C contains only positive strands, C is saturated. Hence, $\mathfrak{h} = \mathit{true}$ is fine. □

The next proposition shows that $\tau_1 - \tau_2$ is the most restrictive type that satisfies Proposition 8.4.

Proposition 8.5. *Assume* $|\Lambda| \geq 2$. *If* $C_1 - C_2 : \tau$ *for all complexes* $C_1 : \tau_1$ *and* $C_2 : \tau_2$, *then* $\tau_1 - \tau_2 \preceq \tau$.

Proof. Let $\tau_1 - \tau_2 = (S_-, \odot_-, \mathfrak{h}_-)$. Let $a, b \in \Lambda$ with $a \neq b$.

To show that $\tau_1 - \tau_2 \preceq \tau$, we verify the three conditions of Proposition 7.11.

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 $\mathit{data}(S_1)$ consists of the components of S_1 that have a $*$-labeled node. Let $D \in \mathit{comp}(S_-)$. 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. Since $D \in \mathit{comp}(S_-)$, we have $D \in \mathit{data}(S_1)$ or both $D \notin \mathit{data}(S_1)$ and D does not have an isomorphic copy in $\odot_2$. In the first case, there is a component C in $C_1 - C_2$ with $\mathit{stype}(C) \equiv D$, 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 $\mathit{stype}(C) \equiv D$. Since, $C_1 - C_2 : \tau$, we conclude in both cases that $D \sqsubseteq \mathit{comp}(S)$.

2. Proof of $\odot \sqsubseteq \odot_-$. 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 $\odot_2 \cup (\odot_1 - T_1) \sqsubseteq S_2$ by replacing all $*$-labeled nodes by a^ℓ. Note that $\odot_1 - T_1$ consists of the components in $\odot_1$ having an isomorphic copy in S_2. As a result, C_2 has type τ_2 (since $\tau_1 - \tau_2$ is defined). Since we replaced $*$-labeled nodes both in C_1 and in C_2 by a^ℓ, complex $C_1 - C_2$ consists solely of components of weak type $T_1 \equiv \odot_-$, i.e., $\mathit{stype}(C_1 - C_2) \sqsubseteq \odot_-$. By assumption, $C_1 - C_2 : \tau$, whence $\odot \sqsubseteq \mathit{stype}(C_1 - C_2)$. We conclude $\odot \sqsubseteq \odot_-$ as desired.

3. Proof that $\mathfrak{h} = \mathit{true}$ implies saturation of $\tau_1 - \tau_2$. As $\mathfrak{h}_- = \mathit{true}$, τ is trivially saturated.

□

8.3 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 $\texttt{hybridize}_t$.

We now proceed to define the *hybridization* of τ, denoted by $\texttt{hybridize}(\tau)$. First, if $\mathfrak{h} = \mathit{true}$, then this is simply τ itself.

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

Otherwise, we call a component D a *necessary* component of τ if $D \in \mathit{comp}(\odot)$ and D is not isomorphic to $\texttt{immob}(?)$. Let NC be the weak type consisting of all necessary components of τ. Then we define $\texttt{hybridize}(\tau) = (Cs, \odot_h, \mathit{true})$, where

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

and $\odot_h$ is the weak type that 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

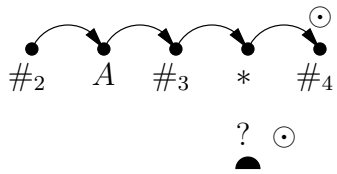

Figure 15: A type τ with two mandatory components.

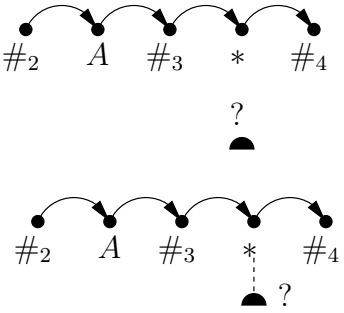

Figure 16: Type $\texttt{hybridize}(\tau)$, where τ is from Fig. 15.

(2) $D = \texttt{immob}(?) \in \mathit{comp}(\odot)$ and there is no component in S with an free node labeled with $*$ or $\hat{*}$.

Note that $\texttt{hybridize}(\tau)$ is well defined as $D \cup \odot_h$ not saturated for some $D \in \mathit{comp}(\mathit{Cs})$ would imply that some $D' \in \mathit{comp}(\odot_h)$ is unfinished with respect to Cs — a contradiction.

Example 8.6. Consider type τ displayed in Fig. 15. Type $\tau' = \texttt{hybridize}(\tau)$ is shown in Fig. 16 (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.

Proposition 8.7. $\texttt{hybridize}(\tau)$ *is defined if and only if for all complexes* C *of type* τ*, we have that* $\texttt{hybridize}(C)$ *is defined. In this case,* $\texttt{hybridize}(C) : \texttt{hybridize}(\tau)$.

Proof. Assume first that $\mathfrak{h} = \mathit{true}$. Then $\texttt{hybridize}(\tau)$ is defined and equal to τ. Also, if C is a complex of type τ, then C is saturated. Hence $\texttt{hybridize}(C)$ is defined and equal to C. In this case $\texttt{hybridize}(C) = C : \tau = \texttt{hybridize}(\tau)$.

Assume now that $\mathfrak{h} = \mathit{false}$.

First assume that $\texttt{hybridize}(\tau)$ is defined. Let $C : \tau$. We show that $\texttt{hybridize}(C)$ is defined. Since $\texttt{hybridize}(\tau)$ is defined, $\texttt{hybridize}_t(S)$ is defined. Thus hybridization terminates for weak type S. By Theorem 4.3 there is no alternating cycle in the hybridization graph of S (the definition of hybridization graph is straightforwardly extended to weak types by using the extended complementarity relation). Since ℓ-cores and ?-labeled probes can never engage in an alternating cycle, there is no alternating cycle in the hybridization graph of C, and therefore C has terminating hybridization. Hence $\texttt{hybridize}(C)$ is defined.

Conversely, assume that $\texttt{hybridize}(\tau)$ is not defined. Hence both $\mathfrak{h} = \mathit{false}$ and S has non-terminating hybridization. Let C be a complex with $\mathit{stype}(C) \equiv S$ 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, $\texttt{hybridize}(C)$ is not defined, and C is of type τ because $\mathfrak{h} = \mathit{false}$.

Assume now that $\texttt{hybridize}(\tau)$ is defined and let C be a complex of type τ. We show that $\texttt{hybridize}(C) : \texttt{hybridize}(\tau)$. Let $C' = \texttt{hybridize}(C)$ and let $\tau' = \texttt{hybridize}(\tau) = (\mathit{Cs}, \odot_h, \mathfrak{h}')$.

We first verify that $\mathit{stype}(C') \sqsubseteq \mathit{Cs}$. Let $D \in \mathit{comp}(C')$. We show that $\mathit{stype}(D) \sqsubseteq \mathit{Cs}$. 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. Denote $\mathit{stype}(C) \backslash \texttt{immob}(?)$ by X. We have $\mathit{stype}(D) \sqsubseteq \texttt{hybridize}_t(X)$. Since $\odot \sqsubseteq \mathit{stype}(C)$, we have $NC \sqsubseteq X$. Since $\mathit{stype}(C) \sqsubseteq S$, we have $X \sqsubseteq S$. Thus $\mathit{stype}(D) \sqsubseteq \mathit{Cs}$.

2. $\mathcal{D}$ consists solely of a probe. In this case $\mathit{stype}(D) \equiv \texttt{immob}(?) \sqsubseteq \mathit{stype}(C) \sqsubseteq S$. Hence, $\mathit{stype}(D) \sqsubseteq \mathit{Cs}$.

3. $\mathcal{D}$ contains a probe together with some other copies of components of C. Note that $\mathcal{D}$ can only contain one probe, since probes are immobilized and components of sticker complexes can contain at most one immobilized node. Now since D is a component, the probe is involved in the matching that creates D. Let r be the core having the atomic value node that is matched to the probe, and let E be the component holding r. Then $\mathit{stype}(E)$ has a node (representing r) that is labeled by $*$ or $\hat{*}$. The probe's stype is clearly $\texttt{immob}(?)$. Since both $(*, ?)$ and $(\hat{*}, ?)$ are complementary pairs of symbols, we conclude that $\mathit{stype}(D) \sqsubseteq \texttt{hybridize}_t(\mathit{stype}(C))$. We have $NC \sqsubseteq \odot \sqsubseteq \mathit{stype}(C) \sqsubseteq S$. Hence $\mathit{stype}(D) \sqsubseteq \texttt{hybridize}_t(\mathit{stype}(C)) \sqsubseteq \mathit{Cs}$.

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

1. Assume case (1) holds. Since $s \sqsubseteq \texttt{hybridize}_t(NC)$, and NC consists of the mandatory components except $\texttt{immob}(?)$, we have $s = \mathit{stype}(D)$ 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 $s = \mathit{stype}(D)$ is subsumed by $\mathit{stype}(C')$.

2. Assume now that case (2) holds. Since $\odot \sqsubseteq \mathit{stype}(C)$, 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 $s = \mathit{stype}(D)$ is subsumed by $\mathit{stype}(C')$.

Finally, by definition, $\mathfrak{h} = \mathit{true}$, and indeed C', being the result of a hybridization, is saturated. □

The next proposition shows that $\texttt{hybridize}(\tau)$ is the most restrictive type that satisfies Proposition 8.7.

Proposition 8.8. *If* $\texttt{hybridize}(C) : \tau'$ *for all complexes* $C : \tau$*, then* $\texttt{hybridize}(\tau) \preceq \tau'$*.*

Proof. We first treat the case where $\mathfrak{h} = \mathit{true}$. Let C be a complex of type $\texttt{hybridize}(\tau)$. We show that C is of type τ'. Since $\mathfrak{h} = \mathit{true}$, we have $\texttt{hybridize}(\tau) = \tau$, and, because C is saturated, $\texttt{hybridize}(C) = C$. By the assumption of the lemma, $C = \texttt{hybridize}(C)$ is of type τ'.

We now assume $\mathfrak{h} = \mathit{false}$. Let $\texttt{hybridize}(\tau) = (Cs, \odot_h, \mathfrak{h}_h)$ and $\tau' = (S', \odot', \mathfrak{h}')$. To show $\texttt{hybridize}(\tau) \preceq \tau'$ we verify the three conditions of Proposition 7.11.

1. Proof of $Cs \sqsubseteq S'$. Let $D \in \mathit{comp}(Cs)$. Let $a \in \Lambda$. We distinguish two cases.

   (a) Assume that $D \equiv \texttt{immob}(?)$. Then $D \sqsubseteq S$. 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, and so $\texttt{hybridize}(C) : \tau'$. 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, $D \sqsubseteq S'$.

   (b) Assume that $D \not\equiv \texttt{immob}(?)$. By the definition of S, D is a component in $\texttt{hybridize}_t(X)$ for some weak type X, with $NC \sqsubseteq X \sqsubseteq S$. 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 $\texttt{immob}(?) \sqsubseteq \odot$ but $\texttt{immob}(?) \not\sqsubseteq X$, then we add to C the component $\texttt{immob}(\overline{b})$. Then C has type τ, and so $\texttt{hybridize}(C) : \tau'$. Since $\texttt{hybridize}(C)$ has a component of weak type D, we have $D \sqsubseteq S'$.

2. Proof of $\odot' \sqsubseteq \odot_h$. Let $D \in \mathit{comp}(Cs)$ and $D \not\sqsubseteq \odot_h$. We show that $D \not\sqsubseteq \odot'$. Let $a \in \Lambda$. We distinguish two cases.

   (a) Assume that $D \equiv \texttt{immob}(?)$. We again distinguish two cases.

      i. Assume that $D \not\sqsubseteq \odot$. Let C be a complex such that $\mathit{stype}(C) \equiv \odot$. Complex C has type τ, thus $\texttt{hybridize}(C) : \tau'$. By definition, there is no component in C of type $\texttt{immob}(?)$. Hence, $D \not\sqsubseteq \odot'$.

      ii. Assume that $D \sqsubseteq \odot$. Since $D \equiv \texttt{immob}(?)$ and $D \not\sqsubseteq \odot_h$, there is a component E of 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, and so $\texttt{hybridize}(C) : \tau'$. 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. Thus, $D \not\sqsubseteq \odot'$.

(b) Assume that $D \not\equiv \texttt{immob}(?)$. Then $D \sqsubseteq \texttt{hybridize}_t(X)$ for some $NC \sqsubseteq X \sqsubseteq Cs$. By the fact that $D \not\sqsubseteq \odot_h$, and by the definition of $\odot_h$, we have again two possibilities:

i. $D \not\sqsubseteq \texttt{hybridize}_t(NC)$. 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 $\overline{b}$. Since $NC = \odot - \texttt{immob}(?)$, $D \not\sqsubseteq \texttt{hybridize}_t(NC)$, and free probes cannot interact with ℓ-cores in C, there is no component in $\texttt{hybridize}(C)$ having type D. Complex C has type τ, and so $\texttt{hybridize}(C) : \tau'$. Thus, $D \not\sqsubseteq \odot'$.

ii. $D \not\sqsubseteq \texttt{hybridize}_t(S)$. 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. Hence, $\texttt{hybridize}(C) : \tau'$. By our assumption, $D \not\sqsubseteq \texttt{hybridize}_t(S)$, so there is no component in $\texttt{hybridize}(C)$ having type D. Hence, $D \not\sqsubseteq \odot'$.

3. Proof of $\mathfrak{h}' = \mathit{true}$ implies that $\texttt{hybridize}(\tau)$ is saturated. Since $\mathfrak{h}_h = \mathit{true}$, we have that $\texttt{hybridize}(\tau)$ is trivially saturated.

□

8.4 Ligate, Flush, and Split

The definitions of `ligate`, `flush`, and `split` on complexes are naturally adapted to weak types. Given this, we now define these operations on types.

The *ligation* of τ, denoted by $\texttt{ligate}(\tau)$, equals $(\texttt{ligate}(S), \texttt{ligate}(\odot), \mathfrak{h})$. Moreover, the *flush* of τ, denoted by $\texttt{flush}(\tau)$, equals $(\texttt{flush}(S), \texttt{flush}(\odot), \mathfrak{h})$. Finally, the *split* of τ on $\sigma \in \{\#_2, \#_3, \#_4, \#_6, \#_8\}$ (i.e., σ is the label of a split point), denoted by $\texttt{split}(\tau, \sigma)$, equals $(\texttt{split}(S, \sigma), \texttt{split}(\odot, \sigma), \mathfrak{h})$.

Since `ligate`, `flush`, and `split` have nothing to do with atomic value symbols, the following is easy to verify.

Proposition 8.9. *If* $C : \tau$*, then* $\texttt{ligate}(C) : \texttt{ligate}(\tau)$*,* $\texttt{flush}(C) : \texttt{flush}(\tau)$*, and* $\texttt{split}(C, \sigma) : \texttt{split}(\tau, \sigma)$*.*

The next proposition shows that $\texttt{ligate}(\tau)$, $\texttt{flush}(\tau)$, and $\texttt{split}(\tau, \sigma)$ are the most restrictive types that satisfies Proposition 8.9.

Proposition 8.10. *If* $\texttt{ligate}(C) : \tau'$ *for all complexes* $C : \tau$*, then* $\texttt{ligate}(\tau) \preceq \tau'$*. Analogous statements hold for* $\texttt{flush}(\tau)$ *and* $\texttt{split}(\tau, \sigma)$*.*

Proof. Let $a \in \Lambda$. To show that $\texttt{ligate}(\tau) \preceq \tau'$, we verify the three conditions of Proposition 7.11.

1. Proof of $\texttt{ligate}(S) \sqsubseteq S'$. Let $D \in \mathit{comp}(\texttt{ligate}(S))$. 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 by Lemma 7.9. Hence, $\texttt{ligate}(C) : \tau'$. By definition, $\texttt{ligate}(C)$ contains a component of weak type D, and so $D \sqsubseteq S'$.

2. Proof of $\odot' \sqsubseteq \texttt{ligate}(\odot)$. Let $D \in \mathit{comp}(\texttt{ligate}(S))$ and $D \not\sqsubseteq \texttt{ligate}(\odot)$. We show that $D \not\sqsubseteq \odot'$. Let C be a complex such that $\mathit{stype}(C) \equiv \odot$.

Complex C has type τ by Lemma 7.9. Hence, $\texttt{ligate}(C) : \tau'$. By the construction of C, there is no component of weak type D in $\texttt{ligate}(C)$. Thus, $D \not\sqsubseteq \odot'$.

3. Proof of $\mathfrak{h}' = \mathit{true}$ implies $\texttt{ligate}(\tau)$ is saturated. If the $\mathfrak{h}$-bit of $\texttt{ligate}(\tau)$ is *true*, then the implication is trivial. Assume now that the $\mathfrak{h}$-bit of $\texttt{ligate}(\tau)$ is *false*. Then $\mathfrak{h} = \mathit{false}$. 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 and so $\texttt{ligate}(C) : \tau'$. 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, saturation of $\texttt{ligate}(C)$ implies saturation of C. 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, $\texttt{ligate}(S)$ is saturated, whence $\texttt{ligate}(\tau)$ is saturated.

The proofs for $\texttt{flush}(\tau)$ and $\texttt{split}(\tau, \sigma)$ are similar, except for the proof that $\mathfrak{h}' = \mathit{true}$ implies $\texttt{flush}(\tau)$ is saturated. In this case, we alter the proof by letting C be a complex such that $\mathit{stype}(C) \equiv S'$ with all ℓ-cores equal to a^ℓ and all probes labeled $\overline{a}$ and S' is the weak type obtained from S by retaining all immobilized components. We observe as before that S' is saturated, and so $\texttt{flush}(S) \equiv \texttt{flush}(S')$ is saturated, and so $\texttt{flush}(\tau)$ is saturated. □

8.5 Block

The block operator $\texttt{block}(C, \sigma)$, on a sticker complex C with $\sigma \in \Omega \cup \Theta$ a tag or an attribute symbol, has nothing to do with atomic value symbols. Hence this operator is naturally adapted to weak types S to obtain $\texttt{block}(S, \sigma)$. Given this, we now define the operator on types. First, if τ is not saturated, then $\texttt{block}(\tau, \sigma)$ is undefined. Otherwise, define $\texttt{block}(\tau, \sigma) = (\texttt{block}(S, \sigma), \texttt{block}(\odot, \sigma), \mathit{true})$.

Proposition 8.11. $\texttt{block}(\tau, \sigma)$ *is defined if and only if for every complex* C *of type* τ*, we have that* $\texttt{block}(C, \sigma)$ *is defined. In this case,* $\texttt{block}(C, \sigma) : \texttt{block}(\tau, \sigma)$.

Proof. If $\texttt{block}(\tau, \sigma)$ is defined, then τ is saturated, and so any complex C of type τ is saturated too. Hence $\texttt{block}(C, \sigma)$ is defined.

Conversely, if $\texttt{block}(\tau, \sigma)$ is not defined, then τ is not saturated. By the definition of saturated, there is an unsaturated complex C having type τ. Thus $\texttt{block}(C, \sigma)$ is not defined.

We now prove the second statement. Assume therefore that $\texttt{block}(\tau, \sigma)$ is defined, i.e., τ is saturated. Let complex C be of type τ. Since $\mathit{stype}(C)$ is subsumed by the weak type of τ, $\mathit{stype}(\texttt{block}(C, \sigma))$ is subsumed by the weak type of $\texttt{block}(\tau, \sigma)$. Similarly, the mandatory type of $\texttt{block}(\tau, \sigma)$ is subsumed by $\mathit{stype}(\texttt{block}(C, \sigma))$. Since C is saturated and the block operation on complexes preserves saturation, $\texttt{block}(C, \sigma)$ is also saturated. Therefore, it is fine that the $\mathfrak{h}$-bit of $\texttt{block}(\tau, \sigma)$ is *true*. □

8.6 Block-From

As for the block operator, we assume here that τ is saturated; otherwise, the operation below is undefined.

Let $\sigma \in \Omega \cup \Theta$. 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.

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 s 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 $\underline{*}$.

We now define $\texttt{blockfrom}(\tau, \sigma) = (\texttt{blockfrom}(S, \sigma), \texttt{blockfrom}(\odot, \sigma), \mathit{true})$.

Proposition 8.12. $\texttt{blockfrom}(\tau, \sigma)$ *is defined if and only if for every complex* C *of type* τ*, we have that* $\texttt{blockfrom}(C, \sigma)$ *is defined. In this case,* $\texttt{blockfrom}(C, \sigma) : \texttt{blockfrom}(\tau, \sigma)$.

Proof. The proof of the first statement is similar as in the proof for `block` (Proposition 8.11).

We now prove the second statement. Assume therefore that $\texttt{blockfrom}(\tau, \sigma)$ is defined, i.e., τ is saturated. Let complex C be of type τ. To show that $\texttt{blockfrom}(C, \sigma)$ is of type $\texttt{blockfrom}(\tau, \sigma)$, we first verify that $\texttt{blockfrom}(C, \sigma)$ is of weak type $\texttt{blockfrom}(S, \sigma)$. Since C is well typed, an ℓ-core in C either occurs entirely in a σ-blocking range, or is entirely disjoint from it. Any node x in an ℓ-core r occurring in a σ-blocking range of C is free, so that in $\mathit{stype}(C)$ the ℓ-core r is represented by a free node r' labeled by $*$. In $\texttt{block}(C, \sigma)$, all nodes x of r are blocked, yielding an ℓ-core of type $\underline{*}$. In $\texttt{blockfrom}(S, \sigma)$, the node r' is relabeled with $\underline{*}$. Hence, $\mathit{stype}(\texttt{blockfrom}(C, \sigma))$ is subsumed by $\texttt{blockfrom}(S, \sigma)$ as desired. The reasoning that $\texttt{blockfrom}(\odot, \sigma)$ is subsumed by $\mathit{stype}(\texttt{blockfrom}(C, \sigma))$ is similar.

Since C is saturated and the block operation on complexes preserves saturation, $\texttt{block}(C, \sigma)$ is also saturated. Therefore, it is fine that the $\mathfrak{h}$-bit of $\texttt{block}(\tau, \sigma)$ is *true*. □

8.7 Block-Except

Operation `blockexcept` is defined on a weak type S if and only if each of the following conditions hold:

1. every node labeled with $*$ is not matched, and is preceded and followed by a free node;
2. every node labeled with $\hat{*}$ is matched;
3. every node labeled $\hat{*}$ or $\underline{*}$ is preceded and followed by a closed node;

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_2 is labeled $*$. For any such triple, we relabel n_2 to $\hat{*}$, and we add n_1 and n_3 to β.

We now say that $\texttt{blockexcept}(\tau)$ is defined if and only if $\texttt{blockexcept}(S)$ is defined and τ is saturated. In this case, $\texttt{blockexcept}(\tau)$ is defined as $(\texttt{blockexcept}(S), \texttt{blockexcept}(\odot), \mathit{true})$.

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.

Proposition 8.13. *Let $\ell > 2$ and let $n \in \{1, \ldots, \ell\}$. Then $\texttt{blockexcept}(\tau)$ is defined if and only if for every ℓ-complex C of type τ, we have that $\texttt{blockexcept}(C, n)$ is defined. In this case, $\texttt{blockexcept}(C, n) : \texttt{blockexcept}(\tau)$.*

Proof. Assume that $\texttt{blockexcept}(\tau)$ is defined and let C be an ℓ-complex of type τ. We show that $\texttt{blockexcept}(C, n)$ is defined by verifying the three conditions in its definition. Condition (1) holds by assumption. Condition (3) holds since τ is saturated. Condition (2) states that 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. type $*$: none of the nodes of v are blocked, and none of the nodes are matched, due to the second condition of the block-except operation on types.
2. type $\hat{*}$: a single node x of r is not blocked. Node x has to be matched, due to the third condition of the block-except operation on types. Moreover, the $\#_3$ and $\#_4$ of v' are closed.
3. type $\underline{*}$: all nodes of r are closed. Due to the third condition of the definition of the block-except operation on types, all nodes of v are closed.

Consequently, Condition (2) holds and we thus conclude that $\texttt{blockexcept}(C, n)$ is defined.

Conversely, assume that $\texttt{blockexcept}(\tau)$ is not defined. Then one of these conditions holds:

1. There is a component D of S with
   (a) a $*$-labeled node x such that (i) x is not free, (ii) x is not preceded by a free node, or (iii) x is not followed by a free node, or
   (b) a $\hat{*}$-labeled node x that is free, or
   (c) a $\hat{*}$ or $\underline{*}$-labeled node x that is not preceded or not followed by a closed node.
2. τ is not saturated.

In the first case, let C be a complex with $\mathit{stype}(C) \equiv \odot \cup D$. By Lemma 7.9, C has type τ. Complex C contains an ℓ-vector with both free and closed nodes. Thus, by definition, $\texttt{blockexcept}(C, n)$ is not defined. In the last case, by the definition of saturation, there is a complex C of type τ that is not saturated. Consequently, $\texttt{blockexcept}(C, n)$ is not defined. So, in each case, $\texttt{blockexcept}(C, n)$ is not defined.

We now prove the second statement. Assume therefore that $\texttt{blockexcept}(\tau)$ is defined and let complex C be of type τ. Given that $\texttt{blockexcept}(C, n)$

is defined, one easily verifies that the mandatory type of $\texttt{blockexcept}(\tau)$ is subsumed by $\mathit{stype}(\texttt{blockexcept}(C, n))$, which is in turn subsumed by the weak type of $\texttt{blockexcept}(\tau)$. Since C is saturated and the block-except operation on complexes preserves saturation, $\texttt{blockexcept}(C, n)$ is also saturated. Therefore, it is fine that the $\mathfrak{h}$-bit of $\texttt{blockexcept}(\tau)$ is *true*. □

We notice that a result analogous to Proposition 8.10 holds for `block`, `blockfrom`, and `blockexcept`. The proof is similar as the proof of Proposition 8.10, except that the third condition becomes trivial because the $\mathfrak{h}$-bit of the result is *true*.

Proposition 8.14. *If* $\texttt{block}(C, \sigma) : \tau'$ *for all complexes* $C : \tau$*, then* $\texttt{block}(\tau, \sigma) \preceq \tau'$*. Analogous statements holds for* $\texttt{blockfrom}(\tau, \sigma)$ *and* $\texttt{blockexcept}(\tau)$*.*

8.8 Cleanup

Recall that $\mathit{strands}(S)$ denotes the set of positive strands of weak type 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 an isomorphic copy of 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 $\underline{*}$. Note that $n(t) + (\ell - 1)a(t)$ equals the length of a strand represented by t in a complex of dimension ℓ.

We now define the *cleanup* of τ, denoted $\texttt{cleanup}(\tau)$, to be $(S_{\mathit{clean}}, \odot_{\mathit{clean}}, \mathit{true})$, where S_{clean} and $\odot_{\mathit{clean}}$ are defined as follows.

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 an integer solution in the variable $\ell \geq 2$. So, s qualifies if and only if for some dimension ℓ and some ℓ-complex of type τ, s has maximum length. The weak type S_{clean} is defined to be consisting 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 of S_{clean} *qualifies for mandatory*, if for each strand t of S_{clean}, the strict inequality $n(s)+(\ell-1)a(s) < n(t)+(\ell-1)a(t)$ has *no* integer solution in $\ell \geq 2$. So, s qualifies for mandatory if and only if for every dimension ℓ and ℓ-complex of type τ, s has maximum length. Now, $\odot_{\mathit{clean}}$ is defined to consist of those strands s of S_{clean} that both qualify for mandatory and originate from a component that is mandatory in τ.

Example 8.15. Assume the weak type S consists of the following four components: D_1 is the strand $\#_3 * \#_4 \#_3 * \#_4$, D_2 is the strand $\#_3 * \#_4$, D_3 is the strand $\#_2\#_2\#_2\#_2\#_2$, and D_4 is the strand $\#_2\#_2\#_2$.

If D_2 is the only component of $\odot$, then $S_{\mathit{clean}} \equiv D_1 \cup D_2 \cup D_3$ and $\odot_{\mathit{clean}}$ is the empty weak type. Indeed, $S_{\mathit{clean}} \equiv D_1 \cup D_2 \cup D_3$ because $n(D_1) + (\ell - 1)a(D_1) = 6 + (\ell - 1) \cdot 2 = 2\ell + 4 \geq \ell + 2 = n(D_2) + (\ell - 1)a(D_2)$ for any ℓ and $n(D_3) + (\ell - 1)a(D_3) = 5 \geq \ell + 2$ if $\ell = 2$, but $n(D_4) + (\ell - 1)a(D_4) = 3 < \ell + 2$ for all $\ell \geq 2$. Moreover, $\odot_{\mathit{clean}}$ is in this case the empty weak type because, e.g., $n(D_2) + (\ell - 1)a(D_2) < n(D_3) + (\ell - 1)a(D_3)$ for $\ell = 2$.

If D_1 is the only component of $\odot$, then $S_{\mathit{clean}} \equiv D_1 \equiv \odot_{\mathit{clean}}$. Indeed, $n(D_1) + (\ell - 1)a(D_1)$ is greater than or equal to $n(D_i) + (\ell - 1)a(D_i)$ for all $i \in \{1, \ldots, 4\}$ and $\ell \geq 2$.

Since the result of the cleanup operation is a set of positive strands, $\texttt{cleanup}(C)$ of a complex C is trivially saturated and so it is fine to have the $\mathfrak{h}$-bit of $\texttt{cleanup}(\tau)$ set to *true*.

From the above observations we obtain the following.

Proposition 8.16. *If* $C : \tau$*, then* $\texttt{cleanup}(C) : \texttt{cleanup}(\tau)$.

The next proposition shows that $\texttt{cleanup}(\tau)$ is the most restrictive type that satisfies Proposition 8.16.

Proposition 8.17. *If* $\texttt{cleanup}(C, \sigma) : \tau'$ *for all complexes* $C : \tau$*, then* $\texttt{cleanup}(\tau) \preceq \tau'$.

Proof. Let $\texttt{cleanup}(\tau) = (S_{clean}, \odot_{clean}, \mathfrak{h}_{clean})$. To show $\texttt{cleanup}(\tau) \preceq \tau'$ we verify the three conditions of Proposition 7.11.

1. Proof of $S_{clean} \sqsubseteq S'$. Let $s \in \mathit{comp}(S_{clean})$. By definition, component s is a strand and s *qualifies* for S_{clean}, 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 weak type s is at least as long as all other positive strands in C, whence $\texttt{cleanup}(C)$ contains a component having weak type s. Complex C has type τ. Hence, $\texttt{cleanup}(C) : \tau'$. Thus, $s \sqsubseteq S''$.

2. Proof of $\odot' \sqsubseteq \odot_{clean}$. Let $s \sqsubseteq S_{clean}$ and $s \not\sqsubseteq \odot_{clean}$. We show that $s \not\sqsubseteq \odot'$. Recall that 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 $D \sqsubseteq \odot$, then let C be a complex such that $\mathit{stype}(C) \equiv \odot$. There is no component in C having $\mathit{stype}(C)$ in $\omega(s)$, whence there is no strand having weak type s in C, thus there is no strand having type s in $\texttt{cleanup}(C)$. Complex C has type τ. Hence, $\texttt{cleanup}(C) : \tau'$. Thus, $s \not\sqsubseteq \odot'$.

   (b) There is a component $D \in \omega(s)$ such that $D \in \mathit{comp}(\odot)$, but strand s does not qualify for mandatory. Hence there is a strand t of S_{clean} for which the strict inequality $n(s) + (\ell - 1)a(s) < n(t) + (\ell - 1)a(t)$ has a positive integer solution in the variable ℓ. Let E be a component from $\omega(t)$. Let C be a complex with dimension x such that $\mathit{stype}(C) \equiv \odot \cup E$. In complex C strands having weak type t are strictly longer than strands having weak type s, whence $\texttt{cleanup}(C)$ does not contain a strand having weak type s. Complex C has type τ. Hence, $\texttt{cleanup}(C) : \tau'$. Thus, $s \not\sqsubseteq \odot'$.

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

□

9 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.

Denote the set of free variables of a DNAQL expression e (i.e., those variables not bound by `for` or `let` constructs) by $FV(e)$. 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. Fig. 17). 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, $dom(\Gamma)$, to types. Let e be a DNAQL expression. If $dom(\Gamma) \supseteq FV(e)$, then we say that Γ is a type assignment *on* e.

The *typing relation* for DNAQL is defined in Fig. 17. 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 and say that e is *well typed* under Γ.

The domain of Γ is extended from variables to expressions as specified in Fig. 17. For the basic operators, we first typecheck subexpressions and then applies the operator on the type level as defined and investigated in Section 8. The `let`- and `for`-constructs are typed in the standard manner. We first typecheck the initializer expression, yielding a result type τ_1; then we typecheck the body expression with variable x declared to be of type τ_1.

Our approach to typing `if`-expressions leverages the least upper bound type defined and investigated in Section 7.3. Consider the four final typing rules in Figure 17. The first two of these can predict the outcome of the emptiness test, in those cases where the type declared for x is empty (so the content of x must be empty), or the type has mandatatory components (so the content of x cannot be empty). The third rule further inspects the type x and applies when the underlying weak type consists of a single component D (up to equivalence). In that case we can assign a sharper type to x in the `else`-branch: indeed, x is not empty there, so we can make D mandatory. The output type of the expression is the least upper bound of the result types of the `then`- and `else`-branches. The final rule is now as expected in the absence of any "smart" predictions.

The defined typing relation is clearly unambiguous, i.e., if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then τ_1 and τ_2 are equal up to isomorphism of their weak types and their mandatory types.

Recall the formal semantics of DNAQL (Section 6). When ℓ is not important, we refer to an ℓ-complex assignment simply as a complex assignment. Let Γ be a type assignment, and let ν be a complex assignment. We naturally say that ν *has type* Γ if $dom(\nu) = dom(\Gamma)$ and for all $x \in dom(\nu)$, 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 $[\![\Gamma]\!]$.

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \cup e_2 : \tau_1 \cup \tau_2}$$

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

$$\frac{\Gamma \vdash e : \tau \qquad \texttt{hybridize}(\tau) \text{ is well defined}}{\Gamma \vdash \texttt{hybridize}(e) : \texttt{hybridize}(\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ligate}(e) : \texttt{ligate}(\tau)}$$

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash x : (S, \odot, \mathfrak{h}) \qquad comp(\odot) = \emptyset \qquad comp(S) \neq \emptyset \\ \forall D_1, D_2 \in comp(S), D_1 \equiv D_2 \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x := (S, S, \mathfrak{h})] \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \texttt{if empty}(x) \texttt{ then } e_1 \texttt{ else } e_2 : \tau_1 \vee \tau_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : (S, \odot, \mathfrak{h}) \\ comp(\odot) = \emptyset \qquad \exists D_1, D_2 \in comp(S), D_1 \not\equiv D_2 \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \texttt{if empty}(x) \texttt{ then } e_1 \texttt{ else } e_2 : \tau_1 \vee \tau_2}$$

Figure 17: Typing relation of DNAQL.

9.1 Soundness

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 $\Gamma \models e : \tau$.

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 follows from the relational algebra simulation in DNAQL [8] 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.

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 9.1. *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 $e = x \in dom(\nu)$ be a variable. By Fig. 17, $\Gamma \vdash x : \Gamma(x)$. 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.

Operator Propositions 8.1, 8.4, 8.7, 8.9, 8.11, 8.12, 8.13, and 8.16 together prove the case where e is of the form of an operator applied to subexpressions.

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 $e = \texttt{if empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. 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 immediately.
4. The fourth inference rule is proven similar to the third rule.

□

Example 9.2. Recall the program from Example 6.1 in Section 6.

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 $e_1 = \texttt{hybridize}(x_1 \cup \texttt{immob}(\bar{a}))$ is typed as $(S_1^?, \emptyset, \mathit{true})$, where $S_1^?$ consists of the following components: (i) S_1 itself; (ii) $\texttt{immob}(?)$; and (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 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 9.3. For another example, consider the program

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

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 $\#_3\#_4$ 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 `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.

9.2 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 stands 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 stands 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 9.4. *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. The union, ligate, flush, split, and cleanup operations are always defined on the type level and so the result holds trivially for these operations. The if-directions in Propositions 8.4, 8.7, 8.11, 8.12, and 8.13 prove the result for the difference, hybridize, block, block-from, and the block-except operations, respectively. □

9.3 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 [26].

By Propositions 8.2, 8.5, 8.8, 8.10, 8.14, and 8.17 we have the following.

Theorem 9.5. *For every atomic expression, the DNAQL type relation is tight.*

10 No Maximality and Tightness for Non-Atomic Expressions

We introduced the notions of maximality and tightness on arbitrary DNAQL expressions. However, Theorems 9.4 and 9.5 apply to atomic expressions only.

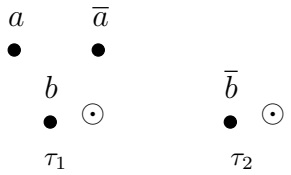

Figure 18: Types τ_1 and τ_2. The types consist of one-node components. Both types have their $\mathfrak{h}$-bit, $\mathfrak{h}_1$ resp. $\mathfrak{h}_2$, set to *true*.

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. In a companion paper [8], it is shown that expression e can be translated to an equivalent DNAQL expression e^{DNA}. 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 Fig. 18. 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.

11 Discussion and Further Work

In a companion paper [8] we show that the relational algebra can be simulated by DNAQL programs. Indeed, we like to think of DNAQL as an analogue to the relational algebra, when working in the sticker complex data model instead of the relational data model.

An interesting problem is to understand the precise expressive power of well-typed DNAQL programs. We conjecture that every well-typed DNAQL program can be simulated in the relational algebra (on relational structures representing the typed input complexes). Confirming this conjecture would firmly establish DNAQL as the DNA-computing equivalent of the relational algebra.

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

Acknowledgment

Jan Van den Bussche is partially supported by the National Natural Science Foundation of China (61972455).

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] R. Appuswamy, K. Le Brigand, P. Barbry, M. Antonini, O. Madderson, P. Freemont, J. McDonald, and T. Heinis. OligoArchive: Using DNA in the DBMS storage hierarchy. In *Proceedings 9th Conference on Innovative Data Systems Research (CIDR 2019)*, 2019.

[5] M. Arita, M. Hagiya, and A. Suyama. Joining and rotating data with molecules. In *Proceedings 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pages 243–248, 1997.

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

[7] J. Bornholt, R. Lopez, D. Carmean, L. Ceze, G. Seelig, and K. Strauss. A DNA-based archival storage system. In T. Conte and Y. Zhou, editors, *Proceedings 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 637–649. ACM, 2016.

[8] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. The relational completeness of the DNA query language DNAQL. In preparation, 2019.

[9] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. A type system for DNAQL. In D. Stefanovic and A. Turberfield, editors, *Proceedings 18th International Conference on DNA Computing and Molecular Programming (DNA18)*, volume 7433, pages 12–24. Springer, 2012.

[10] R. Brijder, J.J.M. Gillis, and J. Van den Bussche. The DNA query language DNAQL. In *Proceedings 16th International Conference on Database Theory*. ACM Press, 2013.

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

[12] L. Cardelli. Abstract machines in systems biology. In *Transactions on Computational Systems Biology III*, volume 3737 of *Lecture Notes in Computer Science*, pages 145–178. Springer, 2005.

[13] L. Cardelli. Strand algebras for DNA computing. In Deaton and Suyama [17], pages 12–24.

[14] 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.

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

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

[17] R.J. Deaton and A. Suyama, editors. *Proceedings 15th International Meeting on DNA Computing and Molecular Programming*, volume 5877 of *Lecture Notes in Computer Science*. Springer, 2009.

[18] 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.

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

[20] 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 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2010.

[21] Nick Goldman, Paul Bertone, Siyuan Chen, Christophe Dessimoz, Emily M. LeProust, Botond Sipos, and Ewan Birney. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Science*, 494:77–80, 2013.

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

[23] 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.

[24] U. Majumder and J.H. Reif. Design of a biomolecular device that executes process algebra. In Deaton and Suyama [17], pages 97–105.

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

[26] 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.

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

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

[29] 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.

[30] J.H. Reif. Parallel biomolecular computation: Models and simulations. *Algorithmica*, 25:142–175, 1999.

[31] 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.

[32] S. Roweis, E. Winfree, R. Burgoyne, N.V. Chelyapov, M.F. Goodman, P.W.K. Rothemund, and L.M. Adleman. A sticker-based model for DNA computation. *Journal of Computational Biology*, 5(4):615–629, 1998.

[33] 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.

[34] N. Schiefer and E. Winfree. Universal computation and optimal construction in the chemical reaction network-controlled tile assembly model. In A. Phillips and P. Yin, editors, *Proceedings 21st International Conference on DNA Computing and Molecular Programming (DNA 21)*, volume 9211 of *Lecture Notes in Computer Science*, pages 34–54. Springer, 2015.

[35] 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.

[36] D. Soloveichik, G. Seelig, and E. Winfree. DNA as a universal substrate for chemical kinetics. *PNAS*, 107(12):5393–5398, 2010.

[37] S. M. H. Tabatabaei Yazdi, Y. Yuan, J. Ma, H. Zhao, and O. Milenkovic. A rewritable, random-access DNA-based storage system. *Scientific Reports*, 5(14138), 2015.

[38] 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.

[39] E. Winfree, X. Yang, and N.C. Seeman. Universal computation via self-assembly of DNA: some theory and experiments. In L.F. Landweber and E.B. Baum, editors, *DNA Based Computers II: DIMACS Workshop*, pages 191–213. American Mathematical Society, 1998.

[40] D. Woods, D. Doty, C. Myhrvold, J. Hui, F. Zhou, P. Yin, and E. Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567:366–372, 2019.

[41] M. Yamamoto, Y. Kita, S. Kashiwamura, A. Kameda, and A. Ohuchi. 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.

[42] C.-W. Yeh, K.-R. Wu, and W. Meng. Development of a database model based on parallel biomolecular computation. *Simulation Modelling Practice and Theory*, 21(1):39–51, 2012.