2013 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

DOCTORAATSPROEFSCHRIFT

DNA and Databases: The Sticker Complex Model

Proefschrift voorgelegd tot het behalen van de graad van doctor in wetenschappen, informatica, te verdedigen door:

Joris Gillis

Promotor: prof. dr. Jan Van den Bussche

D/2013/2451/10

Maastricht University
universiteit hasselt
KNOWLEDGE IN ACTION

Acknowledgments

At first, being a PhD student is a daunting experience. All those smart people that used to teach me all kinds of clever things are suddenly colleagues, capable of small talk during lunch breaks. Instead of asking questions to a TA, I am on the receiving end of an endless stream of questions. Instead of studying for an exam, I sweat on formulating clear, funny, and representative questions and swear while grading poor exams instead of pondering whether the answer to question three was formulated accurately and eloquently enough. Nonetheless, I have very much enjoyed doing research, writing papers, visiting conferences, giving talks, guiding thesis students, discussing various topics at lunch time, and joking around with my fellow PhD students. I thank all my colleagues, friends and family for their advice, chats, cheers, support and meals during the preparation of my PhD.

I especially thank my supervisor Jan Van den Bussche, whom I admire for his elegant, concise and to-the-point style of writing. Although we have our disagreements, I have learned many things from him, most importantly, an ability to abstract.

I thank Jonny Daenen, my roomie, for many fruitful discussions on our research topics, the latest Apple products and teaching in general.

I thank my brother San Gillis for proofreading my dissertation and asking challenging questions about computer science. Those questions gave me a sense of purpose during the dark days of researcher's block, a variant of writer's block. I also enjoyed sharing "war stories" on being a PhD student.

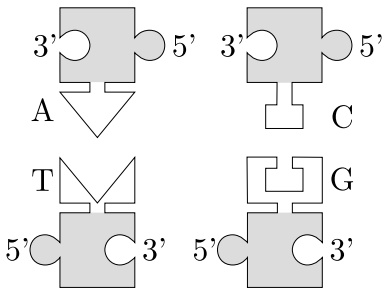
I thank my parents who always supported my in the best possible way, namely, in their own way. They are always there when I need comfort, advice, feedback on a text or an idea, or just someone to listen to me, or a hug.

I thank Maya Budo for being there, loving me, caring for me, advising me on how to handle students, and listening to things that must have sounded as if I worked at "Hogwarts School of Witchcraft and Wizardry" instead of Hasselt University.

Samenvatting

Desoxyribonucleïnezuur (DNA) is de drager van genetische informatie in grote en kleine organismen. Een DNA-streng is opgebouwd uit vier verschillende basisdeeltjes, nucleotiden of basen genoemd: `A`, `C`, `T`, en `G` (zie Figuur 1). Bindingen kunnen ontstaan tussen complementaire basen, `A-T` en `C-G`. Twee DNA-strengen die volledig gebonden zijn, vormen de beroemde dubbele helix. Het vormen van bindingen tussen complementaire basen wordt *hybridizatie* genoemd. Afhankelijk van de lengte en de omgevingsfactoren, kunnen twee niet perfect complementaire DNA-strengen toch binden. Bijvoorbeeld, DNA-strengen `CCCACCC` en `GGGGGGG` zijn niet perfect complementair, want basen `A` en `G` zijn niet complementair. Toch is het mogelijk dat de `C-G`-bindingen sterk genoeg zijn om één imperfectie te tolereren. DNA-moleculen worden bewaard in een waterige oplossing in een proefbuis. Door de oplossing op te warmen worden bindingen tussen basen gebroken, dit proces heet *denaturatie*. Hybridizatie en denaturatie zijn de twee belangrijkste operaties op DNA-moleculen.

Recente biotechnoligsche ontwikkelingen maken het mogelijk naar believen snel, accuraat en goedkoop synthetische DNA-moleculen te creëren en de base-sequenties van DNA-moleculen in een proefbuis "uit te lezen". Deze vooruitgang maakt het mogelijk om ook andere informatie, bv. een boek, een foto of een audiofragment, op te slaan met DNA-moleculen. Als informatiedrager



Figuur 1: Een abstracte voorstelling van de basisblokken van een DNA-streng: de nucleotiden ook wel basen genoemd.

heeft DNA meerdere voordelen t.o.v. klassieke informatiedragers zoals dvd's en harde schijven. De molecule is *zeer robuust* tegen externe "gevaren" zoals water, hitte en schokken. Om DNA uit fossielen te extraheren moet al het andere biologisch materiaal vernietigd worden, tot er alleen DNA overblijft. Ten tweede, heeft DNA een heel *hoge informatiedichtheid*. Bijvoorbeeld, momenteel is er een kleine kamer nodig om het archief van het CERN, goed voor 90 petabyte aan data, op te slaan. Datzelfde archief kan opgeslagen in een proefbuisje met 41 gram DNA.

Informatie opslaan in DNA-moleculen is relatief gemakkelijk. Moeilijker en interessanter is het manipuleren en ondervragen van opgeslagen data. De laatste twintig jaar is er het bloeiend onderzoeksgebied *"DNA-computing"* ontstaan rond de vraag: hoe berekeningen uitvoeren met DNA-moleculen als basis? Dit onderzoeksgebied brengt informatici, biologen, chemici en fysici samen met als doel DNA-computers te ontwerpen. Sinds de publicatie van de eerste DNA-computer door Leonard Adleman in 1994, zijn vele verschillende types DNA-computers gebouwd. Sommige types zijn enkel gebaseerd op hybridizatie. Bijvoorbeeld, de bouwblokken van hedendaagse chips, logische poorten, kunnen gesimuleerd worden met enkele hybridizatie reacties. Andere types baseren zich op een verzameling van operaties op DNA-moleculen geabstraheerd uit de werking van enzymes, de werkpaarden van levende cellen. Adlemans DNA-computer hoort thuis in deze tweede categorie. Een DNA-computer wordt steeds geabstraheerd in een theoretisch model. Een theoretisch model voorziet in een abstractie van DNA-moleculen en definieert transformaties op deze abstractie die labo-operaties nabootsen.

De huidige DNA-computers doelen erop alle taken van digitale computers aan te kunnen. Dit betekent dat de basisoperaties van dergelijke DNA-computers expressief moeten zijn, om complexe berekeningen uit te kunnen drukken. Een databank manipuleren en ondervragen vergt echter niet zulke expressieve operaties. Integendeel, hoe expressiever de operaties van een computer, hoe complexer programma's worden. Een complex programma optimaliseren is moeilijk of zelfs onmogelijk. Omwille van deze reden worden database-talen, om data te manipuleren en te ondervragen, opzettelijk zo simpel als mogelijk gehouden, zonder triviaal te zijn. Dit biedt een voordeel, want expressieve operaties ontwerpen op DNA-moleculen vergt ingewikkelde (tot het randje van onmogelijke) acrobatieën van de DNA-moleculen. Ofwel wordt de expressiviteit ontleend aan het samenstellen van vele kleine operaties, zoals in het geval van de lerende DNA-computer, waardoor het programmeren van zo'n DNA-computer een tijdrovend en uitermate ingewikkeld proces is. Dus is er een opening voor een type DNA-computer gericht op data manipulatie en ondervraging, programmeerbaarheid en implementeerbaarheid met DNA-moleculen. Deze dissertatie introduceert zo een model.

Het model, genaamd *sticker complex model*, is ontworpen met praktische

ramificaties steeds in het achterhoofd. De basisbouwblokken van het model zijn relatief korte (20 basen) DNA-strengen die een *DNA-code* vormen. Een DNA-streng in een DNA-code kan alleen binden met zijn perfecte complement en niet met de andere DNA-strengen in de DNA-code. Een element van een DNA-code noemen we een codewoord. Een codewoord noemen we *positief*. Zijn complement noemen we *negatief*. Een streng is een sequentie van codewoorden. Eén streng in het model, representeert duizenden DNA-strengen. Verschillende restricties zijn van toepassingen op strengen. Ten eerste, moet een streng homogeen zijn, in de zin dat ofwel alleen positieve ofwel alleen negatieve codewoorden worden gebruikt. Zo vermijden we dat een streng met zichzelf kan binden, wat kan leiden tot fysiek onmogelijke constructies. Ten tweede, de negatieve strengen zijn beperkt tot twee codewoorden. Positieve strengen modelleren data, terwijl de negatieve strengen, *stickers* genoemd, gebruikt worden voor de manipulatie van de data. Een verzameling van strengen die direct of indirect verbonden zijn via bindingen tussen complementaire codewoorden, noemen we een *component*. Ten derde, een negatief codewoord kan "*verankerd*" worden. In een proefbuis betekent dit dat de overeenkomstige DNA-strengen chemisch op een oppervlakte gebonden zijn. Een component is verankerd als er een verankerd codewoord in aanwezig is. De bindingssites zijn ruimtelijk verspreid zodat twee verankerde componenten niet kunnen binden. Een component in het sticker complex model kan dus maximaal één verankerd codewoord bevatten. Een verzameling van strengen en stickers, al dan niet gebonden, noemen we een *sticker complex*.

Een sticker complex is een abstractie van DNA-moleculen die slechts een beperkte klasse van DNA-moleculen modelleert om fysieke implementeerbaarheid te garanderen. Op sticker complexen is een verzameling van operaties gedefinieerd, gebaseerd op standaard biotechnologische operaties. Er is een operatie om strengen te knippen, om strengen aan elkaar te lijmen, om niet-verankerde componenten weg te spoelen, en om de langste strengen in een complex te isoleren. Een programmeertaal genaamd *DNA Query Language* (DNAQL), bundelt deze operaties tesamen met een test-operatie en een herhaal-operatie. De test-operatie test of een complex leeg is, zodat een programma keuzes kan maken. De herhaal-operatie schuift een teller over een interval $[1, \ell]$, want data-elementen worden opgeslagen als een sequentie van ℓ elementaire data-elementen. Vergelijkbaar met digitale computers die data-elementen opslaan als sequenties van `0`'en en `1`'en. De herhaal-operatie maakt het mogelijk om elementaire data-elementen individueel aan te spreken.

Hybridizatie is een zeer krachtige operatie en moet dus goed in het oog gehouden worden. Inderdaad, het is mogelijk dat de componenten van een sticker complex aanleiding geven tot een oneindig aantal mogelijke combinaties. DNA in een proefbuis is echter altijd eindig, dus slechts een klein deel van de mogelijke combinaties kunnen gevormd worden. Welke combinaties dat

zijn, is onmogelijk te voorspellen. De andere DNAQL operaties kunnen ook ongewenste effecten vertonen in bepaalde situaties. Daarom is er een *type systeem* ingevoerd op de DNAQL programmeertaal die "goede" programma's kan onderscheiden van "slechte" programma's. Verder, wordt een simulatie van de *relationele algebra* (RA) voorgesteld. De relationele algebra is de basis van alle grote database systemen. Er wordt bewezen dat de simulatie zich altijd goed gedraagt. Dit toont dus aan dat sticker complexen en DNAQL geschikt zijn voor het manipuleren en ondervragen van data opgeslagen in DNA.

Contents

1

Introduction

The *deoxyribonucleic acid molecule*, or DNA, is best known as the bearer of hereditary information and solver of many (television) crimes. The basic building blocks of a DNA molecule are the nucleotides. A *nucleotide* consists roughly speaking of two parts: a *"backbone"* part and a *"base"* part, as shown in Figure 1.1. The backbone part is the same in all nucleotides, the base distinguishes nucleotides. A nucleotide can be fitted with one of four different bases: Adenine, Cytosine, Thymine, or Guanine. Customarily a nucleotide is identified by the first letter of its base, i.e., A, C, T, or G. Five carbon atoms form the backbone part of a nucleotide. These carbon atoms are numbered 1' to 5'. The 5' carbon atom of one nucleotide can bind with the 3' carbon atom of another nucleotide. Hence, nucleotides can chain together in long sequences. A sequence of nucleotides is called a *DNA strand*. Because a 5' carbon atom always binds with a 3' carbon atom, a DNA strand has a direction. The nucleotide with the free, i.e., unbonded, 3' carbon atom indicates the 3' end of the strand. By construction, the other end of the strand ends with a nucleotide with a free 5' atom, hence, it is called the 5' end. Commonly, a DNA strand is denoted by its base sequence and the direction of the strand. For example, 5'-AATCCG-3' represents a DNA strand with six nucleotides with bases A, A, T, C, C, and G, such that the first nucleotide has a free 5' carbon atom. Bases can also form bonds, yet only between complementary bases: A and T are complementary and C and G are complementary. This is called *Watson-Crick complementarity* after the discoverers of the structure of the DNA molecule. Occasionally, non-complementary bases can bond, although they form a less stable connection and are thus likely to occur. The famous

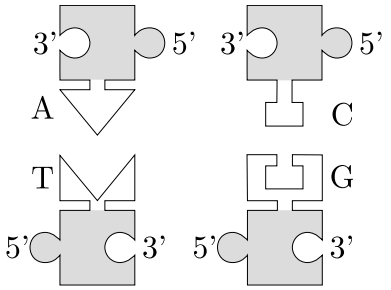


Figure 1.1: An abstraction of the four DNA nucleotides. The gray part is the backbone. The white parts are the bases: A and T bond and C and G bond.

double helix structure of the DNA molecule consists of two DNA strands that are complementary and *anti-parallel*, i.e., reading one strand in 5'-3' direction spells the complement of the other strand in 3'-5' direction. For example, 5'-AATCCG-3' forms a double helix with 3'-TTAGGC-5'. Constructing the DNA strand that forms a double helix with a DNA strand s is thus simply reversing the base sequence of s and replacing each base by its complement. The constructed DNA strand is called the reverse-complement of DNA strand s. A DNA strand with a bases bonded is called *double-stranded*. If no bases are bonded a DNA strand is said to be *single-stranded*. A DNA strand can also be partly double-stranded and partly single-stranded. This process of forming bonds between bases is called *hybridization* or *annealing*. The inverse process, breaking bonds between bases, is called *denaturing*.

Rumor has it that the field of DNA computing was invented by Leonard Adleman, while reading a book on molecular biology in his bed, saying to his wife: "Yeaz, these things can compute". Adleman developed an algorithm to solve the Hamiltonian path problem on directed graphs, a Hamiltonian path visits every node of the graph exactly once, based on standard biotechnology operations on DNA molecules. This experiment kicked off DNA computing [2]. Figure 1.2 shows the graph of which Adleman computed a Hamiltonian path. The Hamiltonian path is indicated with dashed lines. The Hamiltonian path problem is a NP-complete problem. Nonetheless, Adleman's algorithm uses only a polynomial number of wet-lab operations to compute the Hamiltonian path (if present). Central to the algorithm is the parallel generate-and-test strategy, i.e., first the exponential number of potential paths through the graph are constructed in parallel, with a single wet-lab operation, next all generated paths are subjected, once again in parallel, to a series of tests during which non-Hamiltonian paths are discarded.

How can one construct an exponential number of paths with just a single operation? Adleman encoded the graph by assigning to each node a sequence

Table 1.1: An example encoding of the nodes in the 7-node graph shown in Figure 1.2

Node	Base Sequence
0	5'-TCTCCGAGTTTCTACGCTGT-3'
1	5'-CGCAAGCGAGTGGGAGACGG-3'
2	5'-CGTTTCCTCGACACCCGAAA-3'
3	5'-CTATGTAATAAAGCTATAAT-3'
4	5'-TGGTTAAAATAGTGGTGACT-3'
5	5'-GTCCTCAATAGGCCCGGGAC-3'
6	5'-GGCAGAGGCACAGCGTTCCA-3'

of 20 DNA bases. Table 1.1 (a) shows an example of such an assignment. A Hamiltonian path is a sequence of nodes, thus a path is encoded as a concatenation of the DNA strands representing the nodes. Hence, an edge from node u to node v is formed by the reverse-complement of the concatenation of the last 10 bases of the strand representing u and the first 10 bases of the strand representing v. Consequently, an edge binds two consecutive nodes. Note that the first node can still receive an incoming edge because its first 10 bases are still single-stranded and the second node is still available for an outgoing edge. Also, the direction of edges is preserved in this encoding. All the paths in a graph are now constructed by collecting surplus quantities of all node and edge encodings in a test tube. Gradually cooling down the solution in the test tube initiates the hybridization process. Hybridization will now do the heavy lifting for us, as illustrated in Figure 1.3 where a path starting with node 0 and followed by node 1 is being formed.

Can we be absolutely certain that all paths are formed? Put simply, no, we cannot. Aggravating matters, because the graph is cyclic an infinite number of paths exists, while there is only a finite amount of DNA to construct paths. Nonetheless, the length of a path has an inverse correlation to its likelihood of being formed. Indeed, a test tube is a three dimensional space and strands need to meet in order to hybridize. As a result, longer strands need more time to form and are less likely to form than shorter paths. Thus, as Adleman showed, if enough resources are available, where "enough" depends on the application and on experimental conditions, all paths will be formed *with high probability*. With high probability is a key notion in DNA computing. Chemical processes are inherently stochastic and error-prone, thus error-correction and redundancy are essential parts of DNA computers.

Adleman's experiment took seven days to complete. Compared to modern digital computers this is slow. Nevertheless this proof-of-concept experiment

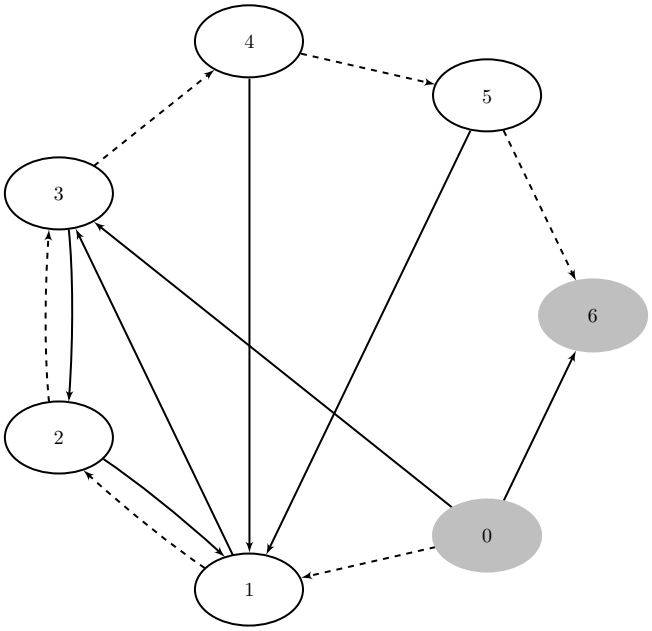


Figure 1.2: The graph for which Adleman computed the Hamiltonian path.

Table 1.2: Example encoding of the edges in the 7-node graph shown in Figure 1.2.

Edge	**Base Sequence**
$0 \rightarrow 1$	5'-CTCGCTTGCGACAGCGTAGA-3'
$0 \rightarrow 3$	5'-TATTACATAGACAGCGTAGA-3'
$0 \rightarrow 6$	5'-TGCCTCTGCCACAGCGTAGA-3'
$1 \rightarrow 2$	5'-CGAGGAAACGCCGTCTCCCA-3'
$1 \rightarrow 3$	5'-TATTACATAGCCGTCTCCCA-3'
$2 \rightarrow 1$	5'-CTCGCTTGCGTTTCGGGTGT-3'
$2 \rightarrow 3$	5'-TATTACATAGTTTCGGGTGT-3'
$3 \rightarrow 2$	5'-CGAGGAAACGATTATAGCTT-3'
$3 \rightarrow 4$	5'-ATTTTAACCAATTATAGCTT-3'
$4 \rightarrow 1$	5'-CTCGCTTGCGAGTCACCACT-3'
$4 \rightarrow 5$	5'-TATTGAGGACAGTCACCACT-3'
$5 \rightarrow 1$	5'-CTCGCTTGCGGTCCCGGGCC-3'
$5 \rightarrow 6$	5'-TGCCTCTGCCGTCCCGGGCC-3'

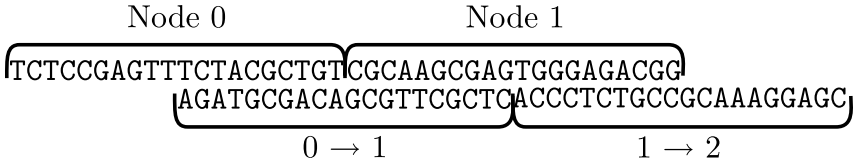


Figure 1.3: The formation of a path consisting of nodes 0 and 1 with edges 0 → 1 and 1 → 2

raised a lot of interest into the field of DNA computing. Shortly afterwards other experiments were devised to solve other NP-complete problems, e.g., SAT and Minimal Set Cover [5, 27, 43]. In the wake of the experiments came theoretical models, e.g., the parallel filtering model based on Adleman's experiment. Early on, two excellent books were written on the theory of DNA computing [3, 33]. A major problem with the generate-and-test-style DNA computing is the need to construct an exponential library of potential solutions at the start of an algorithm. Although this step is "quick", i.e., just one wet-lab operation, it trades time for space. A lot of space it appears. A DNA computer using Adleman's algorithm to compute the Hamiltonian paths in a 60-node graph uses an amount of DNA weighing roughly the same as planet Earth [23]. As a result, the focus of the DNA computing community shifted towards other computing paradigms such as tiling through self-assembly and strand displacement cascades, which will be discussed in Chapter 2. At the same time, DNA computing has also high potential for database applications [4, 11, 55, 41, 12, 21]. Computational models in the DNA computing field aim to be Turing-complete. As a result, models are either hard to implement in the wet-lab or they offer such basic primitives that writing a program is vexing.

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

Sticker complexes represent the structural content of a test tube. We assume that each component of a sticker complex is redundantly present in a tube. If a DNA complex can hybridize to itself, it can hybridize as well to an identical copy. Often, the copy can hybridize with yet another copy and so forth. We identify this undesirable behavior as *non-terminating hybridization*.

Non-terminating hybridization leads to infinite sticker complexes. In practice, when we have termination of hybridization, a test tube prepared with sufficient quantities of each component of the complex holds, in principle, sufficient material to produce all molecular species that can be the result of hybridization. If sufficient quantities are present, adding even more material will not yield new results. Of course, in practice, a test tube is always finite and the hybridization reaction will, under normal conditions, always "terminate" (reach equilibrium). But the point is that, when hybridization does not terminate 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 indeed infinite. Fortunately, it is efficiently decidable for a sticker complex whether it has terminating hybridization.

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

A difficulty with DNAQL, and with DNA computing in general, however, is that various manipulations of DNA must make certain assumptions on their input so as to be effectively implementable and produce a well-defined output. Even when these assumptions are well understood for each operation in isolation, the problem is exacerbated in an applicative programming language like DNAQL, where the output of one operation serves as input for another. Indeed the problem of deciding whether a given program will have well-defined behavior on all possible intended inputs is typically undecidable. While this undecidability is well known for Turing-complete programming languages, it remains so for database languages that are typically not Turing-complete [50].

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 [35, 22]. We propose a type system for DNAQL and establish a soundness theorem. In addition, the type system is maximal

and tight. That is, if an operation is defined on all complexes of a certain type, the operation's counterpart on types is defined on the considered type. In other words, the type system only forbids the application of an operation if there is a reason to. Secondly, the type system might output types that are too loose, in the sense that the type outputted by an operation can be slimmed down without jeopardizing the soundness of the type system. We prove that the types produced by the type system are not too loose, i.e., the type system is tight. Moreover, we show that the type system is flexible enough so that arbitrary relational databases can be represented as typed DNA complexes, and so that arbitrary relational algebra expressions on these data can be expressed by well-typed DNAQL programs. The relational algebra is the applicative language at the core of standard database query languages such as SQL [15, 19, 1].

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

2

State Of The Art

Since Adleman's experiment DNA computing has evolved significantly. This chapter discusses the theoretical models that form the state of the art of DNA computing, complemented by experimental results if applicable. The DNA molecule supports many types of reactions. Many of these reactions can be utilized to perform some computation. Hybridization is the most important reaction and lays the foundation for many methods of computing. The next two sections describe two such methods. The first method employs the hybridization reaction to construct intricate patterns such that the result of the computation can be derived from the final pattern. In the second method, computation is performed by DNA strands "fighting" over binding sites.

2.1 Self-Assembly

One of the first successors to Adleman's filtering model is the Abstract Tile Assembly Model (aTAM) introduced by Erik Winfree [53] and based on the concept of Wang Tiles [52]. Doty wrote an excellent overview of theoretical work on self-assembly with tiles [17]. A tile is conceptually a square. The sides of the square are denoted by the cardinal directions. Each side of a tile is labeled with a glue. One tile is designated as the *seed* tile, i.e., the tile from which the self-assembling of tiles starts. The seed tile has glues only on the west and north sides, hence, only the south and east glues of a tile matter for joining an assembly. Each glue is identified by a symbol and is assigned a strength. Sides of two tiles are said to match if they are labeled with the same glue. A tile can join an assembly only if it glues with strength at least

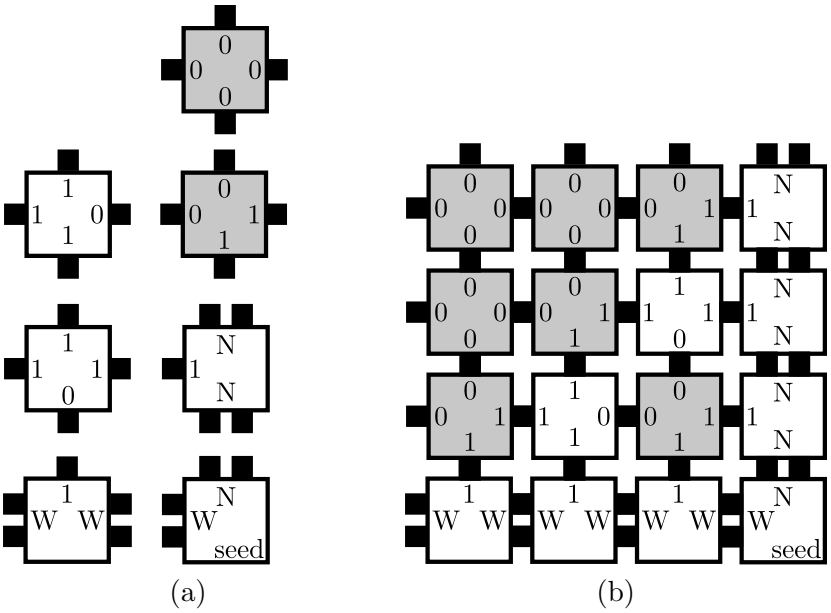


Figure 2.1: (a) A set of tiles forming the discrete Sierpinski triangle pattern. The number of squares on a side denotes the strength of a glue. (b) An assembly of the tiles in (a). Based on [17].

two. Consequently, a tile can be glued to the west or north side of another tile if they share a glue of strength two. Diagonally, a tile can already join an assembly if it shares a glue of strength one with the tile on its south and a glue of strength one with the tile of the east. Figure 2.1 shows (a) a set of tiles forming the discrete Sierpinski triangle pattern and (b) an assembly of these tiles. The number of squares on a side denotes the strength of a glue. The N and W glues both have strength 2, whereas the other glues have strength 1. If we consider the south and east sides of a tile as its input and the north and west sides as its output, then these tiles compute the XOR function.

Winfree showed in his PhD thesis that the aTAM is computationally universal in the sense that for each single-tape Turing Machine M and each input string x, there is a set of tiles to simulate the operation of M and an initial assembly (a seed tile possibly extended with other tiles), such that there is one unique terminal assembly representing the run of M on x.

Central to self-assembly is the notion of *tile complexity*, the number of tile types need to form a given pattern. One could also call it "shape complexity", where a complex shape needs many tiles. The most rudimentary pattern is the single dot. Obviously, a seed tile is sufficient to construct this pattern. Constructing a line of n dots in the aTAM requires n tiles. If less than n

tiles are used, at least one tile must be repeated. The assembly between the repetition can be reiterated infinitely. Things start to get interesting in the two dimensional case. Rothemund and Winfree considered the $n \times n$ square which became the de facto benchmark of tile complexity [42]. They showed that in the aTAM a $n \times n$ square has a tile complexity of $\Omega(\frac{\log(n)}{\log(\log(n))})$, by using binary counters.

The Abstract Tile Assembly Model does not provide a realistic view of how DNA tiles behave in a test tube. The model makes two assumptions known not to hold. Firstly, tiles can attach to an assembly but never detach. However, a tile attaching to an assembly is essentially a hybridization reaction which is a reversible chemical reaction. In other words, in a test tube tiles can detach from an assembly. Secondly, tiles only attach if their gluing strength exceeds a threshold. However, a tile can briefly attach to an assembly with a glue strength of one, even is the threshold is set to two. During this brief moment, the tile can get locked in by other tiles with sufficient binding strength, keeping the incorrect tile in place. In other words, the aTAM disregards errors inherent to DNA computing. The Kinetic Tile Assembly Model (kTAM) was introduced to eliminate these two unrealistic assumptions. The kTAM considers the concentration of each tile and expects that the rate of attaching and detaching is set for each glue. The bases C and G bind with greater strength, thus a glue with many G's and C's will bind easier and faster than a glue with a low GC-concentration. Furthermore, the temperature and salt concentration have a profound effect on the speed of hybridization reactions. From the errors modeled in the kTAM arose the need for error-correcting. Proofreading and self-healing schemes were devised to correct errors in the self-assembly process [54, 10, 47].

Several extensions of the original aTAM and kTAM were made. The *stepwise assembly* introduced by Reif proceeds in stages [39]. Each stage has its own set of tiles. At the end of a stage all loose tiles are removed before new tiles are added. For some patterns this significantly lowers the tile complexity. In the aTAM the glue strength needed to attach to an assembly is fixed. The glue strength threshold is also called the *temperature*. Glues are implemented as DNA strands and a tile attaching is a hybridization reaction. The hybridization reaction is controlled by temperature in the wet-lab. As the temperature rises, the bond strength must rise accordingly in order to occur. A modification of the tile assembly model is called *temperature programming*, in which the temperature is raised and lowered in successive stages of the assembly. As a result, some tiles may be excluded from binding at some points during the computation. A $n \times n$ square can then be realized by a constant number of tiles and $O(\log(n))$ temperature changes. Chemical processes are inherently stochastic. A natural extension to the aTAM is

randomized self-assembly. Tiles compete for binding sites and the tile concentrations determine the probability that a certain tile type attaches at a certain site. In this model, a line of dots can be constructed with $O(\log(n))$ tiles and an expected length of n dots. *Concentration programming* focuses on the concentration of tiles to compute. Lastly, unequal glues can be allowed to match with nonzero strength, i.e., hybridizations may occur with mismatches. This is called the *flexible glue model* and allows the formation of an $n \times n$ square with $O(\sqrt{\log(n)})$ tiles. Clearly, many options are available to program self-assembly. Last but not least, effectively designing sets of tiles (and glue types) is also an active research topic [56, 29, 26].

The tiles discussed up till now are "rigid" tiles in the sense that glues of the same tile cannot come close enough to interact. As a result they can be conceptualized as squares. Jonoska et al. introduced the *flexible tile model* in which glues are attached to bendable antennae. Such tiles are conceptualized as multisets of glues and glues of the same tiles can interact [24]. Flexible tiles also have a natural correspondence to nodes in directed graphs. In this context, one may wonder what types of graphs can be constructed and how many tiles are required in the construction [25]. We have compared hybridization in the flexible tile model and the sticker complex model [6]. Although there are some connections, the models are significantly different, because the order of symbols is important in the sticker complex model, whereas symbols in flexible tiles are unordered. Jonoska et al. also consider different problems from the problems studied in this dissertation.

2.2 Strand Displacement

The strand displacement reaction involves one *base* strand and two *signal* strands. Both the base and signal strands can conceptually be subdivided into two types of functional domains: (1) a *toehold*, a short sequence allowing signals to "dock" on a base strand and (2) a *specificity* domain, a longer sequence over which signals "fight" to bind. Toehold domains are identified by $t_1, t_2, \ldots$ and specificity domains are identified by $A, B, C, \ldots$. The most rudimentary strand displacement reaction consists of a base strand: $\overline{t_1 A}$ and two signal strands $t_1 A$ and A. Overlining denotes Watson-Crick complement. The strand displacement starts as depicted in Figure 2.2 (a), with signal strand A bonded on the base strand and signal strand $t_1 A$ free. The complementary domains t_1 and $\overline{t_1}$ can bind. This toehold allows signal strand $t_1 A$ to temporarily dock on the base strand. At this moment the base of $\overline{A}$ closest to $\overline{t_1}$ is bonded to the second signal strand, however, by a process called *branch migration* this base can swap its bond with the second signal strand for a new bond with the first signal strand. If the bond is swapped, the signal strands

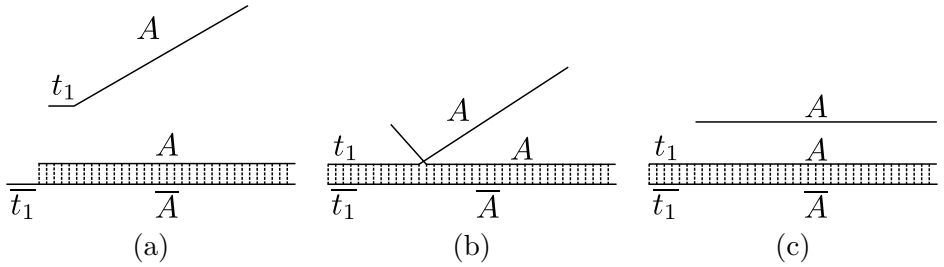


Figure 2.2: (a) Start: signal A bonded on the base strand, signal t_1A floating free. (b) Intermediate: signal t_1A and A "fighting" over $\overline{A}$ on the base strand. (c) End: signal t_1A has displaced signal A on the base strand.

can start "fighting" over the next base. Branch migration behaves as a random walk. The toehold domain is short hence only provides a weak binding between a signal strand and a base strand. If the first signal strand is thus able to gain a complete binding with $\overline{A}$ on the base strand, the second signal strand will detach from the base strand: a strand has been displaced as shown in Figure 2.2 (c). Likewise, it is possible that the first signal strand docks on the base strand, but is immediately kicked off again because the toehold binding is very weak. The displaced strand can be either an output signal or it can be input to a next strand displacement reaction, much like electric current runs through a set of connected logic gates in a digital computer. Because the output of one reaction is the input of the next reaction this style of computing is called strand displacement *cascades*.

Strand displacement cascades have been used to implement Boolean logic gates, chemical reaction networks and neural networks [45, 57, 48, 36, 37, 38]. Next, we describe the implementation of an AND gate. An AND gate takes as input two boolean values x and y and outputs *true* if and only if both values are *true*. Otherwise, the output is *false*. The fact that value x is *true* is represented by the presence of signal strand t_1A. The fact that value y is *true* is represented by the presence of signal strand t_2B. Furthermore, there are signal strands At_2 and B. The presence of a free signal strand B implies that the output of the logic gate is *true*. If this signal strand is not present, then the output is *false*. Signal strand At_2 is an intermediate signal, i.e., it is used during the computation but is not used as output signal. The base strand implementing the AND gate is $\overline{t_1At_2B}$, with signal strand At_2 and B bonded. If value x is *true*, the signal strand t_1A is present and can displace signal strand At_2, thereby freeing toehold t_2. At this point, if value y is *true*, signal strand t_2B can displace the output signal B from the base strand. This strand displacement cascade is shown in Figure 2.3.

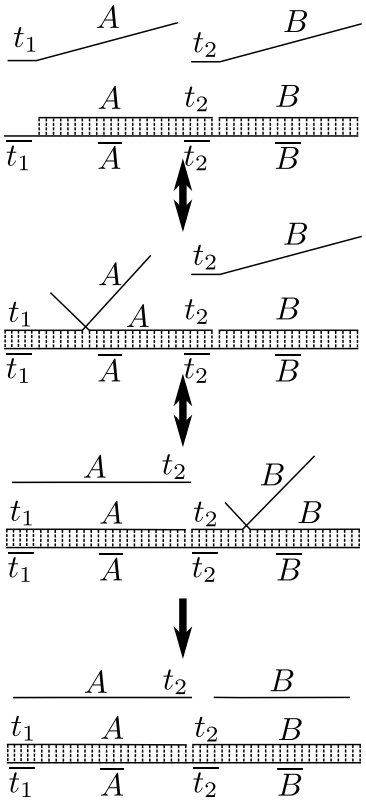


Figure 2.3: Implementation of an AND-gate with strand displacement. If both signals t_1A and t_2B are present, then the output signal B can be displaced or released from the base strand.

The speed of the strand displacement reaction depends on the length of the toehold and specificity domain. A short toehold, for example 3 bases, makes it difficult for the strand to displace another strand, because the toehold binding is so weak the strand has a very limited time before it is kicked off the base strand. The displacement reaction is modeled as a random walk, thus the length of the specificity domain has a clear effect on the time needed to displace a strand. Because a toehold is required to be short, the set of toehold domains is fairly small. Recycling domains is thus important in large-scale computations [13].

The strand displacement reaction is well understood, enabling the creation of the simulation tool *VisualDSD* [7, 34, 8, 9], in which strand displacement can be programmed and easily simulated. This simulation tool makes it easy for everyone to experiment with new types of strand displacement cascades.

3

Sticker Complexes

This chapter introduces the sticker complex model, a theoretical model of a restricted subclass of DNA molecules aiming to be the relational algebra of DNA computing, i.e., not Turing-complete yet highly efficient. The presented model differs slightly from the initial proposal [20]: the labels have been shifted from the edges to the nodes.

3.1 Alphabet

From the outset we assume a finite alphabet Σ. As customary in formal models of DNA computing [3, 33], each letter represents a DNA strand. Crucial in such a set of DNA strands is that each strand will only hybridize with its complement. Any hybridization between a strand and another strand, a strand and the complement of another strand, a strand and the concatenation of other strands, and a strand and the concatenation of complements of other strands is deemed undesirable and must be avoided at all times. Designing such sets of DNA codewords is a research topic on its own [30, 44, 46]. It should always be kept in mind that a symbol in the sticker complex model represents a DNA strand. The alphabet Σ is matched with its negative version $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$, disjoint from Σ. Thus there is a bijection between Σ and $\overline{\Sigma}$, which is called *complementarity* and is denoted by overlining. Obviously, $\overline{a}$ stands for the Watson-Crick complement of the DNA sequence represented by a. The elements of Σ are called *positive symbols* and the elements of $\overline{\Sigma}$ are called *negative symbols*.

For the purpose of data formatting we further assume that $\Sigma = \Lambda \cup \Omega \cup \Theta$

is composed of three disjoint parts: the set Λ of *atomic value symbols*; the set Ω of *attribute names*; and the set $\Theta = \{\#_1, \#_2, \#_3, \#_4, \#_5, \#_6, \#_7, \#_8, \#_9\}$ of *tags*.

3.2 Pre-Complex

We define pre-complexes to contain the overall structure of sticker complexes. A pre-complex is a finite, node-labeled, directed graph where the nodes represent bases in strands and edges indicate direction. Moreover, a pre-complex is equipped with a matching, representing base pairing, and two predicates. One predicate indicates which bases are "immobilized", i.e., do not float freely and can be separated from solution in a controlled manner; the other predicate indicates which bases are "blocked", i.e., cannot participate in base pairing. Formally, a pre-complex is a 6-tuple $(V, L, \lambda, \mu, \iota, \beta)$, where:

- V is a finite set of nodes;
- $L \subseteq V \times V$ is a set of directed edges without self-loops;
- $\lambda : V \to \Sigma \cup \overline{\Sigma}$ is a total function labeling the nodes with positive and negative alphabet symbols;
- $\mu \subseteq [V]^2 = \{\{u, v\} \mid u, v \in V \wedge u \neq v\}$ is a partial matching on the nodes, i.e., each node occurs in at most 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 $strands(S)$ we denote the set of positive strands of pre-complex C.

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

Components

Two strands s and s' are *bonded* if there is a node v in s and some node v' in s' with $\{v, v'\} \in \mu$. When two strands are connected (possibly indirectly) by this bonding relation, we say they belong to the same component. Thus a *component* of a pre-complex is a substructure formed by a maximal set of strands connected by the bonding relation. Note that a component of a pre-complex

(a) (b)

Figure 3.1: An example of two pre-complexes that are non-isomorphic but that are equivalent. These pre-complexes are not isomorphic, as (b) contains twice the number of nodes of (a).

is in itself a pre-complex. We use $comp(C)$ to denote the set of components of pre-complex C. Conversely, we can view a set of pre-complex components as a single pre-complex, basically by taking the union. For convenience, whenever it is clear from the context we write $D \in C$ for a component D and a pre-complex C to denote that D is a component of C, i.e., $D \in comp(C)$.

Subsumption and redundancy

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

A pre-complex C_1 *is subsumed* by pre-complex C_2, denoted by $C_1 \sqsubseteq C_2$, if for each component D_1 in C_1 there is an isomorphic component D_2 in C_2. Two pre-complexes are *equivalent* if they subsume each other, denoted $C_1 \equiv C_2$. A component D in pre-complex C is *redundant* if there exists a component D' in C such that D and D' are isomorphic. Note that removing D from C yields an equivalent sticker complex. A pre-complex is *minimal* if there are no redundant components.

Note that the notions of isomorphism and equivalence are not equal. Indeed, some pre-complexes can be simultaneously non-isomorphic and equivalent, as shown in Figure 3.1. If two pre-complexes are minimal the notions of isomorphism and equivalence collapse.

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. Naturally, a strand with positive (negative) symbols is called a positive (negative) strand.

3. Negative strands are severely restricted: every negative strand must be a chain of one or two nodes.

4. Matchings by μ only occur between nodes with complementary labels.

5. Nodes in β do not occur in μ.

6. A node can be immobilized only if it is the sole node of a negative strand.

7. Each component can contain at most one immobilized node.

Each requirement is introduced with a specific reason in mind:

1. The back bone of a nucleotide has two binding sites, the 3' and 5' carbon atom. Hence, a DNA strand is always a linear sequence of nucleotides without junctions. Therefore, we restrict the strands in a sticker complex to chains and cycles.

2. Previous theoretical models allowed strands to hybridized with themselves. This can easily lead to intricate conformations. It is difficult to predict which conformations are physically possible. We aim to avoid this issue by separating the positive and negative symbols.

3. Restricting the length of stickers is second restriction aimed keeping the model simple in order to avoid intricate conformations.

4. The partial matching μ is one of the abstraction of hybridization bonds. The alphabet is designed such that a strand can only hybridize with its complement, hence matching only occur between complementary symbols as the represent a strand and its complement.

5. Predicate β indicates longer stretches of double-strandedness. A base can only bond to one other base. Hence, if a symbol is double-stranded it cannot form yet another bond.

6. This restriction also aims at simplifying the model.

7. An immobilized node is the abstraction of a DNA strand chemically attached to a surface. We assume that attachment sites on such a surface are sufficiently separated to bar hybridizations between two immobilized DNA molecules.

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

Isomorphism of sticker complexes can be decided in polynomial time by depth-first search. Indeed, if complexes C and C' both consist of a single component, v is a node of C, and v' is a node of C', then there is at most one isomorphism from C to C' mapping v to v', and this isomorphism can be traced out by depth-first search, following the chain or cycle shape of strands, and the partial matching μ. Depth-first search is in linear time, which yields an isomorphism check for single components in cubic time (try all combinations of v and v'). This algorithm then easily extends to complexes C and C' with multiple components, by matching the components of C to the components of C'. This efficient isomorphism check is in contrast to the problem of general graph isomorphism, which is not known to be decidable in polynomial time. We thus see that sticker complexes form a restricted family of graphs. As a consequence of the efficient isomorphism checking algorithm, the algorithm for minimizing a sticker complex also has polynomial time complexity.

Dimension

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

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

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

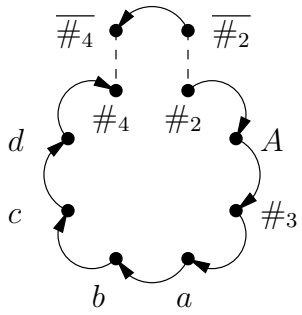


Figure 3.2: A sticker complex with one component. The long positive strand is being circularized by the short sticker $\overline{\#_4\#_2}$.

Example 3.1. Figure 3.2 shows a sticker complex with one component. The arrowed edges represent L. The dashed edges represent matchings in μ. The long positive strand is being circularized by the short sticker complex $\overline{\#_4\#_2}$. This is a 4-complex, as there are four atomic value symbols in the 4-vector $\#_3abcd\#_4$. □

3.4 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 the difference. But what is interesting, however, is that we have defined sticker complexes in such a way that each operation always results in a sticker complex when applied to sticker complexes. Moreover, several operations impose additional restrictions on the input, so as to guarantee effective implementability in real DNA.

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

3.4.1 Union

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

3.4.2 Difference

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

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

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

Example 3.2. Consider a 3-complex C_1 with strands $\#_3abc\#_4$, $\#_3cba\#_4$, and $\#_3bac\#_4$; consider complex C_2 with strands $\#_3abc\#_4$, $\#_3cab\#_4$, and $\#_3bca\#_4$. The difference of complexes C_1 and C_2, i.e., $C_1 - C_2$, is equivalent to the complex with the two strands $\#_3cba\#_4$ and $\#_3bac\#_4$. □

3.4.3 Hybridize

Let $C = (V, L, \lambda, \mu, \iota, \beta)$ and $C' = (V', L', \lambda', \mu', \iota', \beta')$ be two complexes. We say that C' is a *hybridization extension* of C if $V = V'$, $L = L'$, $\lambda = \lambda'$, $\iota = \iota'$, $\beta = \beta'$ and μ' is an extension of μ. Beware that a hybridization extension must satisfy all conditions from the definition of sticker complex. A complex C' is said to be *saturated* if the only hybridization extension of C' is C' itself. This notion captures the hypothesis that everything that can stick in a test tube will stick, i.e., if two substrands can hybridize then they will become hybridized.

The notion of hybridization extension is not sufficient, however, since we want to allow duplicate copies of components in C to participate in hybridization. (This important issue is glossed over in Reif's formalization [40].)

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

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

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

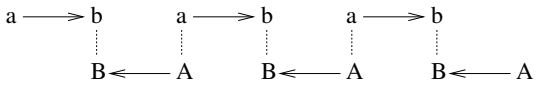


Figure 3.3: Illustration for Example 3.4.

Proposition 3.3. *Let C_1 and C_2 be two equivalent complexes.*

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

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

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

Example 3.4. Consider the simple complex consisting of two strands ab and $\overline{ba}$ and no matchings. For any number n, using n copies of ab and n copies of $\overline{ba}$, we can produce the MHE component shown in Fig. 3.3 for $n = 3$. This component could also be finished, by matching the remaining a shown on the left with the remaining $\overline{a}$ on the right, effectively creating a ring structure. (As always, in the figure, $\overline{a}$ and $\overline{b}$ are shown as A and B.) Different numbers n yield nonequivalent (non-isomorphic) MHE components, thus the number of potential MHE components is infinite. □

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

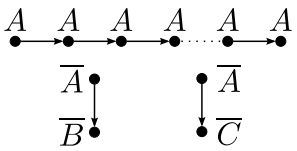


Figure 3.4: The smaller "choice components" can attach to any of the n nodes labeled A of the long "base strand". Hence, they form a binary code with 2^n possible MHE components.

"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 $\texttt{hybridize}(C)$ is defined as the disjoint union of all finished MHE components. Otherwise, the hybridization of C, i.e., $\texttt{hybridize}(C)$, is undefined.

Example 3.5. Excluding non-terminating behavior may seem to restrict the construction of complexes. Nonetheless, large complexes can be constructed without non-termination. Consider the parameterized complex C_n, with n a natural number, composed of the components depicted in Figure 3.4. The smaller "choice components" can attach to any of the n nodes labeled A of the long "base strand". Hence, they form a binary code with 2^n possible MHE components. □

3.4.4 Ligate

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

Example 3.6. Not every sticker introduces a gap. Figure 3.5 shows two situations. The left component has a gap, because the last node of a strand and the first node of another strand are brought together by a sticker. The right component on the other hand involves a node that already has an incoming and outgoing edge, i.e., is not at an end of its backbone, hence no edge can be added between the node labeled C and the node labeled D. Otherwise, the node labeled C would have to outgoing edges. □

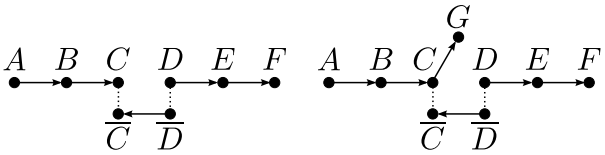


Figure 3.5: On the left a component with a gap. On the right a construction that may be confused for a gap, but is not a gap.

Table 3.1: The allowed split points.

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

3.4.5 Flush

Quite simply $\texttt{flush}(C)$ equals the complex obtained from C by removing all components that do not contain an immobilized node.

3.4.6 Split

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

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

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

3.4.7 Block

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

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

3.4.9 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 $\texttt{blockexcept}(C, n)$ equals the complex obtained from C by blocking, within each ℓ-vector $(e_0, e_1, \ldots, e_\ell, e_{\ell+1})$ that is not yet blocked, all nodes except e_n. If (C, n) does not satisfy the conditions above, then the operation $\texttt{blockexcept}(C, n)$ is undefined.

3.4.10 Cleanup

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

3.5 Implementation in DNA

In this section, we argue that the abstract sticker complexes and the operations on them presented above can be implemented by real DNA complexes. The discussion remains theoretical as we have not performed laboratory experiments. On the one hand, the main purpose is to make the abstract model

plausible as a theoretical framework to explore the possibilities and limitations of DNA computing as a database model; on the other hand, we use only rather standard biotechnological techniques.

Each component of an abstract complex is represented by a large surplus of duplicate copies in DNA. Each positive alphabet symbol from Σ is implemented by a strand of (single-stranded) DNA, such that the resulting set of DNA strands forms a set of DNA codewords. Designing a set of DNA codewords is ongoing research in the DNA computing community, with clear ties to constraint satisfaction problems [30, 44, 46]. If the DNA strand for symbol $a \in \Sigma$ is w, then the DNA strand for the complementary symbol $\overline{a}$, 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 [28] 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 [16]. 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 $\overline{\#_5}$. Polymerase is an enzyme that constructs a DNA strand based on a template strand. More specifically, consider two DNA strands, one is longer than the other. The longer strand is called the *template strand*, the shorter strand is called the *primer*. The primer is situated at the $5'$-end of the template. As shown in Figure 3.6, the template strand extends over the primer strand in the $5'$-$3'$ direction of the primer. The polymerase enzyme attaches to the $3'$-end of the primer and adds to the primer the complements of the template strand until it reaches the end of the template (or it reaches another DNA strand bonded to the template strand), effectively creating a copy of the complement of the template strand. Typically, the primer strand is only tens of bases long, whereas the template strand can be thousands of bases. This enzyme is, literally, of vital importance in the living cells, because it plays a key role in the transformation of a gene into its operational form, the protein. 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 compo-

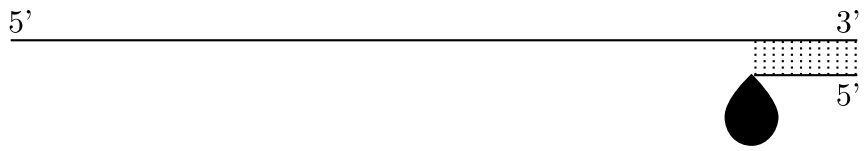


Figure 3.6: The polymerase enzyme (the black droplet) slides over the template strand and extends the primer strand with the complementary bases of the template.

nent 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 restriction enzyme cuts the backbone of a DNA strand whenever it detects a certain base sequence. The base sequence is specific to each enzyme. Furthermore, some enzymes only cut single-stranded DNA molecules, whereas others only cut double-stranded DNA molecules. A feature of the abstract model is that we require only five recognition sites (Table 3.1). Of course, these recognition sites will have to be integrated in the DNA codeword design.

Joining two strands held together by a sticker is implemented by the ligase enzyme. The ligase enzyme mends broken backbones if the ends are close enough.

The flush operation removes all the free floating components. The immobilized components are either attached to a surface, or they are marked by magnetic beads. A surface is large enough to extract mechanically, after which it is washed to remove any free floating components whilst keeping the immobilized components on the surface. Immobilized components marked by magnetic beads can be separated easily by applying a magnetic field on the test tube. For example, an electro-magnet inserted into the test tube will collect all marked components.

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 hybridized 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 the polymerase enzyme.

For the `blockexcept` operation to work, we need to adapt the implemen-

tation of ℓ-vector strands $\#_3 v_1 \dots v_\ell \#_4$, with $v_i \in \Lambda$ for $i = 1, \dots, \ell$, by introducing additional markers ϕ_i, so that we get $\#_3 \phi_1 v_1 \dots \phi_\ell v_\ell \#_4$. These ℓ additional markers must be part of the set of codewords. We can then implement `blockexcept`$(., n)$ by the composition `block`$(., \#_3)$; `blockfrom`$(., \phi_{n-1})$; `block`$(., \phi_{n+1})$; `blockfrom`$(., \#_4)$. In other words, both the left-hand side of the ℓ-vector and the marker following the ith atomic value are blocked to stop the polymerase enzyme which starts from the right-hand side of the ℓ-vector and from the marker proceeding the ith atomic value.

The cleanup operation starts by denaturing (warming up) the tube. Immobilized strands are removed from the tube, either by removing the surface to which strands are attached or by means of a magnetic field. Next, a gel electrophoresis is carried out to separate the longest DNA molecules from the other molecules. Gel electrophoresis is based on the fact that DNA molecules have a negative electrical charge. A special type of gel is spread over a surface. Such a gel is a porous substance, i.e., one can think of such a gel as a tight maze through which smaller objects move quicker than larger objects. DNA molecules are placed at one end of the gel and an electromagnetic field attracts the DNA molecules on the other end of the gel. After some time has passed, DNA molecules will have separated based on length, i.e., bands of DNA molecules have been formed, such that each band contains DNA molecules of roughly same size. The resolution of the separation increases with the length of the gel and time. 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 application of DNAQL, namely the simulation of the relational algebra, presented in Chapter 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.

4

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?

4.1 Deciding termination

When designing a programming language of DNA complexes, it is of course highly desirable to recognize easily whether or not a given complex is terminating. Our main result is the following.

Theorem 4.1. *A complex is terminating if and only if its hybridization graph does not contain a free alternating cycle.*

Corollary 4.2. *Termination of hybridization is decidable in polynomial time.*

Before a formal proof of our theorem can be given, we will formalize the hybridization process with the following notions: hybridization graph, free or immobilized alternating path and cycle, semi-strong homomorphism, and hybridization template. The Corollary will follow since the hybridization graph has the same number of nodes as the given complex, and checking for the presence of an alternating cycle can be done in polynomial time.

The hybridization graph of a complex is an instance of a "partitioned graph". A *partitioned graph* in general is a triple (V, π, E) where (V, E) is

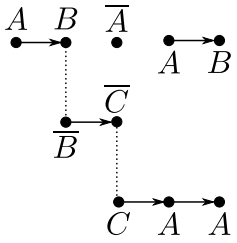


Figure 4.1: Example of a sticker complex. The dotted lines denote the matching μ.

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

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

- V equals the set of nodes of C;
- π contains, for each component D of C, the set of nodes belonging to D as a block;
- Let $F \subseteq V$ be the set of "free" nodes of C; recall a node is called *free* if it is not matched nor blocked. Then E equals $\{\{v, w\} \mid v, w \in F$ and $\lambda(w) = \overline{\lambda(v)}$ and v and w do not belong to different immobilized components $\}$.

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

Example 4.3. The hybridization graph for the complex of Fig. 4.1 is shown in Fig. 4.2. The blocks are depicted as hyperedges (closed curves enclosing the nodes belonging to the same block). The undirected edges are shown as dashed lines. □

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

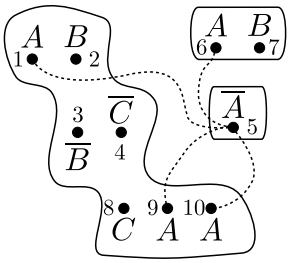


Figure 4.2: The hybridization graph of the complex shown in Figure 4.1. The nodes have been labeled to easily identify paths.

block move: $v_i \neq v_{i+1}$ and they belong to a common block.

The path is said to be *alternating* if edge moves happen for each odd i, and block moves happen for each even i (always for $1 \leq i < n$). An alternating path is called a *free alternating path* if none of the traversed blocks is immobilized. If an alternating path traverses an immobilized block, it is called an *immobilized alternating path*. When the path is alternating, it is said to be an *alternating cycle* when n is odd and at least 3, and $v_n = v_1$. An alternating cycle is also either free or immobilized.

Example 4.4. Consider the hybridization graph for the complex of Fig. 4.1, as shown in Fig. 4.2. Two examples of free alternating paths are $p_1 = 3, 9, 1, 3$ and $p_2 = 3, 1, 10, 3, 6, 7$. Note that p_1 is not an alternating cycle; although it satisfies $v_n = v_1$, its length, 4, is not odd. Indeed, this hybridization graph does not admit an alternating cycle, since the only free node with a negative label, $\overline{a}$, is in a component by itself. □

Example 4.5. Consider the complex discussed in Example 3.4. Its hybridization graph, shown in Figure 4.3, has four nodes partitioned in two blocks. One block, corresponding to the component ab, consists of two nodes 1 and 2 labeled a and b, respectively; the second block, corresponding to the component $\overline{b}\overline{a}$, consists of two nodes 3 and 4 labeled $\overline{b}$ and $\overline{a}$, respectively. There are two undirected edges, namely, $\{1, 4\}$ and $\{2, 3\}$. This hybridization graph admits a free alternating cycle in the form of $1, 4, 3, 2, 1$. □

Example 4.6. Figure 4.4 shows a complex with an immobilized node (immobilized nodes are decorated with the symbol ◓) consists of a sticker and an immobilized component. Figure 4.5 shows the hybridization graph of this complex. Clearly, there is an immobilized alternating cycle in the hybridization graph, i.e., the nodes $\#_4, \overline{\#_4}, \overline{\#_2}, \#_2, \#_4$. Despite the cycles in the hybridization graph, this complex has terminating hybridization, because all cycles run

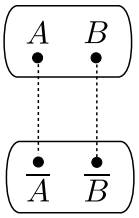


Figure 4.3: The hybridization graph of the complex discussed in Example 3.4.

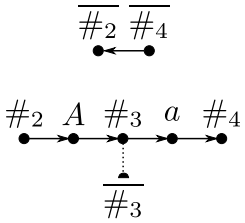


Figure 4.4: A sticker complex with two components. The larger component contains an immobilized node, thus, the component is immobilized. The sticker can make the strand circular.

through the bigger, immobilized component. Two copies of an immobilized component cannot be bonded together, as the resulting component would have two immobilized nodes. Figure 4.6 shows the result of hybridization on this complex. One component forms a cycle, the other has two "arms" spreading from the immobilized component. □

The above three examples are in line with Theorem 4.1. Indeed, the complex of Fig. 4.1 is terminating, and indeed its hybridization graph does not have an alternating cycle; the complex of Example 3.4 is non-terminating,

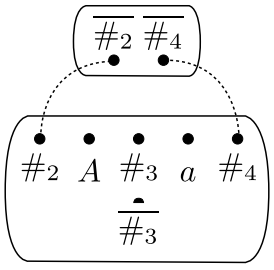


Figure 4.5: The hybridization graph of the complex shown in Figure 4.4.

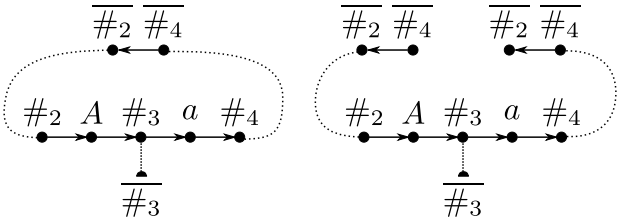


Figure 4.6: The hybridization of the complex shown in Figure 4.4 is terminating despite the alternating cycle in its hybridization graph and consists of two components.

and indeed its hybridization graph has a free alternating cycle; the complex of Example 4.6 is terminating, and indeed although it has an alternating cycle, it is an immobilized alternating cycle.

Next, we present a constructive characterization of MHE components in the form of hybridization templates. But first we need the auxiliary notion of semi-strong homomorphism. Let $G = (V, E)$ and $G' = (V', E')$ be two undirected graphs, and let $f : V \to V'$ be a mapping. Then f is called a *semi-strong homomorphism from G to G'* if, for all $u, v \in V$, we have the following:

- if $\{u, v\} \in E$ then $\{f(u), f(v)\} \in E'$; and
- if $\{f(u), f(v)\} \in E'$ then $\{u, w\} \in E$ for some $w \in V$, or $\{v, w\} \in E$ for some $w \in V$.

The first condition is the standard requirement for homomorphisms; the converse of that condition would state the standard requirement for what is known in universal algebra as a "strong" homomorphism. The second condition, however, states only a weak converse (hence the name "semi-strong"), in the sense that if there is an edge between $f(u)$ and $f(v)$, then either u or v have to be involved in an edge, but not necessarily with each other.

Now let $C = (V, L, \lambda, \mu, \iota, \beta)$ be a complex with hybridization graph $H = (V, \pi, E)$. A *hybridization template for C* is a pair $T = (t, f)$ where $t = (V^t, \pi^t, E^t)$ is a partitioned graph and f is a semi-strong homomorphism from (V^t, E^t) to (V, E), such that:

1. t is connected, i.e., there is a path between any two distinct nodes (using the notion of path in partitioned graphs as defined earlier);
2. E^t is a partial matching, i.e., each node of V^t occurs in at most one edge in E^t; and

3. for each block q of π^t there is a block q' of π such that the restriction $f|_q$ of f to q is a bijection from q to q', i.e., $f|_q$ is injective and the image of $f|_q$ equals q'.

From a hybridization template $T = (t, f)$ for C, and $C = (V, L, \lambda, \mu, \iota, \beta)$ itself, we can construct a sticker complex $comp(T) = (V^T, L^T, \lambda^T, \mu^T, \iota^T, \beta^T)$ as follows:

- $V^T = V^t$;
- $L^T = \{(x, y) \mid x$ and y belong to a common block and $(f(x), f(y)) \in L\}$, note that $(f(x), f(y)) \in L$ is not a sufficient condition as multiple blocks of a hybridization template may map on the same component of C, yet we do not wish to connect all these copies through L^T;
- $\lambda^T(x) = \lambda(f(x))$;
- $\mu^T = E^t \cup \{\{x, y\} \mid x$ and y belong to a common block and $\{f(x), f(y)\} \in \mu\}$, similar to the case of L^T, $\{f(x), f(y)\} \in \mu$ is not a sufficient condition;
- $\iota^T = \{x \in V^T \mid f(x) \in \iota\}$;
- $\beta^T = \{x \in V^T \mid f(x) \in \beta\}$.

Example 4.7. Recall the three example complexes shown in Figures 4.1, 3.3, and 4.4. Example hybridization templates are shown in Figures 4.7, 4.8, and 4.9. The dashed-dotted line indicates the semi-strong homomorphism f. To keep the exposition clear, blocks are connected instead of nodes. It is clear that f is actually the bijection between both blocks. □

Proof of Theorem 4.1

The only-if implication of Theorem 4.1 is relatively easy to prove:

Lemma 4.8. *If the hybridization graph of C has a free alternating cycle, then C is non-terminating.*

Proof. From any alternating cycle $p = v_1, \ldots, v_n$ we can construct an MHE component C_p as follows. For each even i with $1 \leq i < n$, we have a block move in the path: let D_i be the common component of C to which v_i and v_{i+1} both belong. Take distinct copies D'_i of all components D_i; there are $\lfloor n/2 \rfloor$ of them in total. We use D'_0 as a synonym for D'_{n-1}. Then C_p consists of all the copies D'_i, to which we perform the following hybridization extension in two phases. In the first, connection phase, we match, for each edge move $\{v_i, v_{i+1}\}$

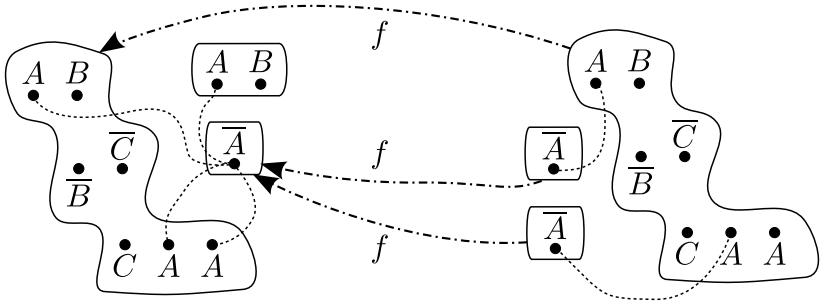


Figure 4.7: An example hybridization template of the complex in Figure 4.1. On the left is the hybridization graph of the complex, on the right is the template. Note that the $\overline{A}$ component has been duplicated whereas the AB strand is dropped. Two pairs of nodes are connected in the template.

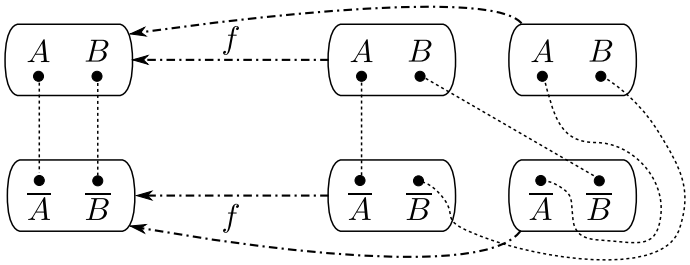


Figure 4.8: An example hybridization template of the complex in Figure 3.3. On the left is the hybridization graph of the complex, on the right is the template. This template forms a circle consisting of two strands and two stickers.

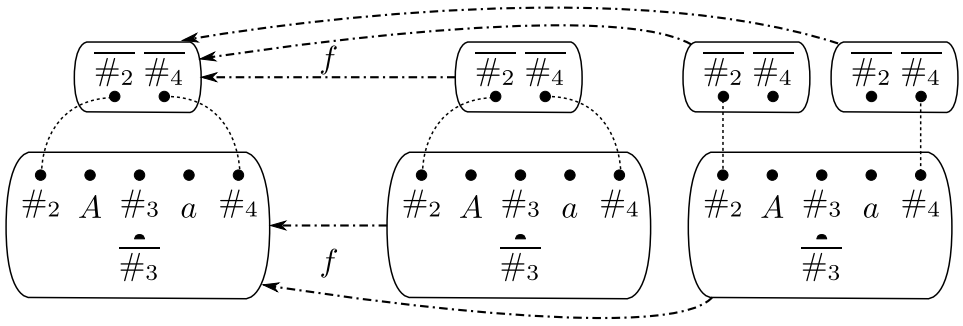


Figure 4.9: An example hybridization template of the complex in Figure 4.4. On the left is the hybridization graph of the complex, on the right is the template. This is a maximal hybridization template.

in the path, the corresponding nodes: the node corresponding to v_i belongs to D'_{i-1} and the node corresponding to v_{i+1} belongs to D'_i. In this way the separate components are connected into a single component. In the second, completion phase, we perform additional hybridization extension arbitrarily so as to obtain maximal matching. The result is an MHE component C_p.

Now for any natural number k, we can form the alternating cycle p^k obtained by repeating p, k times. Formally, p^1 is just p, and if p^k is the sequence $x_1, \ldots, x_N$, then p^{k+1} is defined as the sequence $x_1, \ldots, x_{N-1}, v_1, \ldots, v_n$. Now as above we can construct, for any natural number k, the MHE component C_{p^k}. These components grow strictly larger for increasing values of k and are thus non-isomorphic.

MHE component C_{p^k} may not be finished. Some nodes of p may even be part of other alternating cycles. Nonetheless, the chain of components corresponding to an alternating path can always be closed. Component C_{p^k} can thus always become finished. Hence, hybridization does not terminate. □

The construction explained in the proof of Lemma 4.8 does not work if C has an immobilized alternating cycle, because two immobilized components cannot be connected as there may be at most one immobilized node in any component of a sticker complex.

Proposition 4.9. *The MHE components are exactly the complexes of the form $comp(T)$ with T a hybridization template.*

Proof. Let $T = (t, f)$ be a hybridization template with $t = (V^t, \pi^t, E^t)$. We show that $comp(T)$ is an MHE component. Each block q of t represents a component D_q of C, as determined by f. In $comp(T)$, all directed edges from L, all labels, all matchings, all immobilizations and all blockings are inherited from D_q. Additional matchings are present in $comp(T)$ in the form of the set E^t. Since t is connected, $comp(T)$ consists of a single component.

To show that $comp(T)$ is an MHE component it remains to show that $comp(T)$ has maximal matching. Thereto, let x and y be nodes of $comp(T)$ with complementary labels; we must show that x and y cannot both be free. So, assume x is free; we will show that y is matched in μ^T.

First, note that $f(x)$ is free in C. Indeed, suppose $\{f(x), v\} \in \mu$ for some $v \in V$. Then $f(x)$ and v belong to the same block of π. Let z be the node in the same block as x such that $f(z) = v$. Then $\{x, z\} \in \mu^T$, which is impossible because x is free in $comp(T)$. Now there are two possibilities:

- $f(y)$ is also free in C. Then $\{f(x), f(y)\} \in E$. Hence, since f is a semi-strong homomorphism from (V^t, E^t) to (V, E), at least one of x or y must be matched in E^t. This must be y, since x is free in $comp(T)$. Hence y is matched in $E^t \subseteq \mu^T$ as desired.

– $f(y)$ is matched in μ, so $\{f(y), v\} \in \mu$ for some $v \in V$. Analogously to the reasoning used above for $f(x)$, this implies that y is matched in μ^T as desired.

Conversely, let D be an MHE component. We show that D equals $comp(T)$ for some hybridization template T. By definition, $D = (V', L', \lambda', \mu'', \iota', \beta')$ is a hybridization extension with maximal matching of some redundant variation $C' = (V', L', \lambda', \mu', \iota', \beta')$ of C. So we can form the partitioned graph $t = (V^t, \pi^t, E^t)$ where V^t equals V'; π^t is formed by the components of C'; and E^t equals $\mu'' \setminus \mu'$. Since D forms a single component, t is connected. Since each component of C' is isomorphic to some component of C, we can define $f : V' \to V$ such that, for every block q of t, the restriction $f|_q$ is equal to the corresponding isomorphism. Since D has maximal matching, f is a semi-strong homomorphism. Now clearly $comp((t, f))$ equals D. □

A hybridization template (t, f) is called *maximal* if there is no other hybridization template (t', f'), other than (t, f) itself, such that $V^t \subseteq V^{t'}$; $\pi^t \subseteq \pi^{t'}$; $E^t \subseteq E^{t'}$; and $f \subseteq f'$. From the previous proposition we obtain:

Corollary 4.10. *The finished MHE components are exactly the complexes of the form $comp(T)$ with T a maximal hybridization template.*

Remark 4.11. One can characterize the maximal hybridization templates as follows. They are exactly the hybridization templates that satisfy the stronger definition obtained by replacing, in the definition of semi-strong homomorphism, the second condition by the following:

– if $\{f(u), v'\} \in E'$ for some $v' \in V'$, then $\{u, w\} \in E$ for some $w \in V$. □

The proof of Theorem 4.1 also invokes the following lemma which may be interesting in its own right:

Lemma 4.12. *Let H be a partitioned graph with c distinct blocks. If H admits no alternating cycle, then the length of any alternating path in H is at most $4c + 2$.*

Proof. Let $p = v_1, \ldots, v_n$ be an alternating path in $H = (V, \pi, E)$ and let q be a block of π. For even i with $1 \leq i < n$, we say that q occurs in p at i if v_i and v_{i+1} belong to q (block move). Now assume the same block q would occur at three different i, say, $i_1 < i_2 < i_3$. If $v_{i_2+1} = v_{i_1+1}$, then the subpath of p starting at $i_1 + 1$ and ending in $i_2 + 1$ is an alternating cycle, which is impossible. The construction of this alternating cycle is illustrated in Figure 4.10. Hence $v_{i_2+1} \neq v_{i_1+1}$. Because v_{i_3} is a single node, it cannot be equal to both v_{i_1} and v_{i_2}, thus either $v_{i_3} \neq v_{i_1+1}$ or $v_{i_3} \neq v_{i_2+1}$. In the first case, $\{v_{i_3}, v_{i_1+1}\}$ is a legal block move and by replacing v_{i_3+1} by v_{i_1+1}

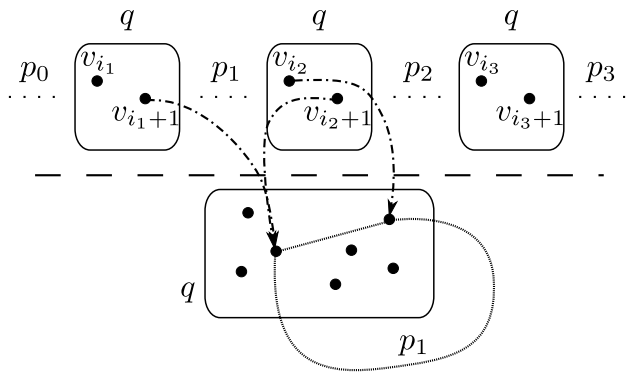


Figure 4.10: Illustration of how an alternating cycle can be formed if $v_{i_1+1} = v_{i_3+1}$ in Lemma 4.12. Path p is split into four parts by the three simplified occurrences of block q. At the bottom, block q is depicted again, with arrows indicating the position of the nodes in the path.

in p, we obtain an alternating cycle starting at $i_1 + 1$ and ending at $i_3 + 1$. The construction of this alternating cycle is illustrated in Figure 4.11, where $v_{i_3} = v_{i_2+1}$. In the second case, we similarly obtain an alternating cycle. We conclude that no block can occur more than twice in p. In other words, the number of block moves in an alternating path is at most $2c$. The number of block moves in an alternating path of length n is $\lfloor (n-1)/2 \rfloor$. Hence, we have $\lfloor (n-1)/2 \rfloor \leq 2c$ which yields $n \leq 4c + 2$. □

We are finally ready to prove the remaining direction of Theorem 4.1:

Lemma 4.13. *If the hybridization graph of C has no free alternating cycle, then C is terminating.*

Proof. Assume there are also no immobilized alternating cycles. To prove that there are only a finite number of non-isomorphic MHE components, we use Proposition 4.9 and prove that there are only a finite number of non-isomorphic hybridization templates. Here, we define an isomorphism between two hybridization templates (t, f) and (t', f') as an isomorphism φ from t to t' such that $f'(\varphi(x)) = f(x)$.

Let $H = (V, \pi, E)$ be the hybridization graph of C. For any hybridization template $T = (t, f)$ we can consider the *blocks tree* of t. The nodes of this tree are the blocks of π^t; the undirected edges are the pairs $\{q, q'\}$ such that $\{v, v'\} \in E^t$ for some $v \in q$ and some $v' \in q'$. Note that it is impossible for some $\{v, v'\}$ to be in E^t with v and v' belonging to the same block q, as this would imply the alternating cycle v, v', v in H. This "blocks tree" is really a

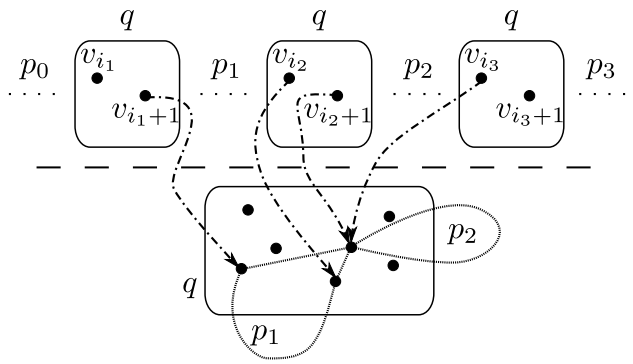


Figure 4.11: Illustration of how an alternating cycle can be formed if $v_{i_3} = v_{i_2+1} \neq v_{i_1+1}$ in Lemma 4.12. Path p is split into four parts by the three simplified occurrences of block q. At the bottom, block q is depicted again, with arrows indicating the position of the nodes in the path. The cycle pass through v_{i_1+1}, takes p_1 to v_{i_2}, does a block move to v_{i_2+1}, traverses p_2 to arrive in v_{i_3+}, and finishes with a block move back to v_{i_1+1}.

tree (undirected graph without cycles); since E^t is a partial matching, a cycle in the blocks tree would imply an alternating cycle in H, which does not exist.

If we know f, then we can reconstruct t from its blocks tree. Also, for a given t, there are only a finite number of possible hybridization templates (t, f); the number of possibilities for f is finite since H is finite. Hence, we are done if we can show that there are only finitely many non-isomorphic blocks trees. This is ensured by the following two properties:

1. The diameter of any blocks tree is at most $4c + 2$. Indeed, since E^t is a partial matching, any simple path in the blocks tree implies an alternating path in H, of the same length. Hence, by Lemma 4.12, the length of any simple path in the blocks tree is at most $4c + 2$.

2. The fan-out of any node in any blocks tree is at most n, where n is the number of nodes of C. Indeed, let q be a block of t. Then q has at most n nodes; by the definition of the edges of the blocks tree, taking into account that E_t is a partial matching, this gives a maximum of n neighbors of q in the blocks tree.

An immobilized alternating cycle will never make a complex non-terminating, as two immobilized components can never be bonded. Hence, an immobilized alternating cycle results in two types of components: (1) the cycle is closed after one iteration or (2) the immobilized component has two arms for each

cycle, each arm is a traversal of the cycle without reentering the immobilized component. □

4.2 Complexity issues

Assume hybridization terminates for a given sticker complex C. Then two follow-up questions come up related to the complexity of the result of hybridization. How many finished MHE components can there be? And, how large can a single finished MHE component become?

Example 4.14. In Example 3.5 we saw that even without alternating cycles exponentially many non-isomorphic MHE components can be formed. When constructing large MHE components, compared to the size of the input complex, the alphabet becomes a limiting factor (as will be formally shown in Lemma 4.16). However, if we assume that the alphabet may grow with the size of the complex, exponentially large MHE components can be formed. A simple example constructs a binary tree. For each level of the tree one alphabet symbol is needed, while the size of the tree grows exponentially with the number of levels. Figure 4.12 shows complex C_n, for $n = 3$, and the resulting MHE component.

Clearly, one can combine the exponential number of MHE components of Example 3.5 with the exponential size MHE component of this example. Assume three alphabet symbols A, B, and C, that are not used in constructing the tree. Instead of a single node component to terminate as leaves of the tree, a strand of n nodes labeled A can be used as leaves. As a result, 2^n nodes labeled A are present at the bottom level of the tree. Adding the stickers $\overline{AB}$ and $\overline{AC}$, results in double-exponentially (2^{2^n}) many MHE components of exponential size, albeit with a growing alphabet. □

As we have already seen in Example 3.5, the *number* of finished MHE components may well grow exponentially in the size of the complex. Also the *size* of MHE components can grow exponentially, as seen in Example 4.14. Unlike Example 3.5, however, the latter can only happen when the alphabet is allowed to grow with the size of the complex. Usually, however, the alphabet is fixed by the application setting. Indeed we show:

Proposition 4.15. *Over the class of terminating complexes over any fixed alphabet, the size of the largest MHE component for a complex C grows only polynomially in the size of C.*

Proof. We reason as in the proof of Lemma 4.13. A rooted tree with fan-out n and depth d has at most $\sum_{i=0}^{d} n^i = (n^{d+1} - 1)/(d - 1)$ nodes. The blocks tree of a hybridization template (where an arbitrary block is chosen as root)

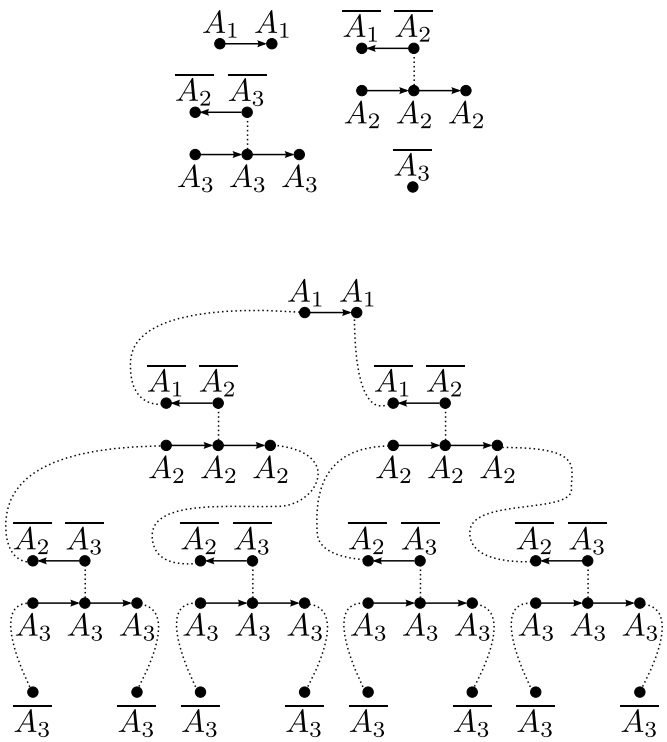


Figure 4.12: Top: Complex C_3 has an MHE component of exponential size. Bottom: The tree constructed from the components of C_3.

has fan-out at most n, and has depth at most $d = 8s + 2$, by Lemma 4.16. Since s is fixed, we obtain a number of blocks that is polynomial in n. Since each block itself has size at most n, the result follows. □

Interestingly, the proof of this proposition relies on the following counterpart to Lemma 4.12. The two lemmas are complementary as Lemma 4.12 does not assume anything about the alphabet, whereas Lemma 4.16 does not assume anything about the complex.

Lemma 4.16. *Let H be the hybridization graph of a complex over positive alphabet Σ. Let s be the number of symbols in Σ. If H admits no alternating cycle, then the length of any alternating path in H is at most $8s + 2$.*

Proof. Let $p = v_1 \dots v_m$ be an alternating path in H and let $a \in \Sigma \cup \overline{\Sigma}$. For even i with $1 \leq i < m$ (block move), we say that a occurs in p at i if $\lambda(v_i) = a$ or $\lambda(v_{i+1}) = a$; in the first case we say that a occurs in first place, in the second case we say that a occurs in second place. It is well possible that a occurs at some i both in first and second place. Now assume a would occur

at three different i (always even). Then it must either occur at least twice in first place, or twice in second place:

- a occurs in first place at some i and at some $j > i$. Note that $\lambda(v_{j-1}) = \overline{a}$. Then $v_{j-1}, v_i, \ldots, v_{j-1}$ is an alternating cycle; a contradiction.
- a occurs in second place at some i and some $j > i$. Note that $\lambda(v_{i+2}) = \overline{a}$. Then $v_{j+1}, v_{i+2}, \ldots, v_{j+1}$ is an alternating cycle; a contradiction.

We conclude that no symbol from $\Sigma \cup \overline{\Sigma}$ can occur in more than two block moves of p. Hence, the number of block moves in an alternating path cannot be greater than $4s$. The number of block moves in an alternating path of length m equals $\lfloor (m-1)/2 \rfloor$, which yields $m \leq 8s + 2$. □

Remark 4.17. Since the number of possible graphs on a polynomial number of nodes is singly-exponential, as a corollary to Proposition 4.15, we obtain that over the class of terminating complexes over a fixed alphabet, the number of MHE components for a complex C is bounded from above by $2^{n^{O(1)}}$, where n is the size of C. Hence, Example 3.5 essentially illustrates the worst that can happen, i.e., double-exponential or worse is impossible, with a fixed alphabet. □

Our final result presents a restriction on classes of complexes, which we call "c-bounded choice" (for a natural number c), so that hybridization is polynomial on the class of c-bounded complexes. It remains to be investigated further how practicable this restriction is, i.e., how many applications can be modeled using sticker complexes that are c-bounded for some c. A positive indication is that only 4-bounded complexes are needed to simulate the relational algebra.

To define the notion of c-boundedness, we first need the notion of a "choice node" of a complex. This is a free node having at least two neighbors in the hybridization graph. Since the edges of the hybridization graph are solely defined in terms of free nodes and their labels being complementary, we see the following, for any label $a \in \Sigma \cup \overline{\Sigma}$: a node v labeled a is a choice node if and only if it is free and there exist at least two free nodes labeled $\overline{a}$. Consequently, if there are at least two free nodes labeled $\overline{a}$, then *all* free nodes labeled a are choice nodes; in the other case, *no* node labeled a is a choice node.

Now for any natural number c, we say that a complex C *has c-bounded choice*, or shorter, *is c-bounded*, if for each component D of C, the number of nodes in D that can reach a choice node by an odd alternating path is at most c. Here, naturally, we say that a node w is reachable by an odd alternating path from a node v, if there is an alternating path $v_1, \ldots, v_n$, with n odd, $v_1 = v$ and $v_n = w$. In particular, taking $n = 1$, any node is reachable from itself by an odd alternating path. As a consequence, in a c-bounded complex, every component has at most c choice nodes.

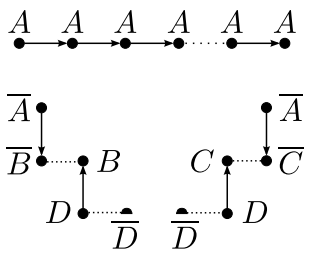


Figure 4.13: Modification C_n'' of the complex shown in Figure 3.4 in which the two bottom components are immobilized.

Example 4.18. Recall the complexes C_n discussed in Example 3.5. Recall that the number of finished MHE components for C_n is 2^n. Since there are two free $\overline{A}$-nodes, the n nodes labeled A are all choice nodes. As these n nodes all belong to a common component, the smallest c such that C_n is c-bounded is n. Hence, there is no fixed c such that all C_n are c-bounded.

Suppose now, we modify C_n to C_n' by removing the sticker $\overline{AC}$. Then the A-nodes are no longer choice nodes. The only remaining choice node C_n' is the $\overline{A}$-labeled node. Hence, each C_n' is 1-bounded. Now note that each C_n' has only one finished component, obtained by annealing each a-node to the $\overline{A}$-node of a fresh copy of the sticker $\overline{AB}$. In particular, hybridization is not exponential on the class of C_n' complexes for all n.

Consider another modification of C_n by immobilizing the smaller "choice" components, called C_n'', as shown in Figure 4.13. Although there are n choice nodes reachable through an alternating path from the two nodes labeled $\overline{A}$, there is no exponential blowup. Indeed, the choice components cannot be combined, as this would result in a component with two immobilized nodes. Hence, complexes that are not c-bounded but with some immobilizations can still have a polynomial-sized hybridization. □

The above example illustrates our result:

Theorem 4.19. *Let c be a natural number. Over the class of terminating, c-bounded complexes over a fixed alphabet, the hybridization of any complex C has size polynomial in the size of C.*

Proof. Since the size of each MHE component is polynomial by Proposition 4.15, we must only show that the number of non-isomorphic finished MHE components is polynomial. Using Corollary 4.10, we can focus on the number of non-isomorphic maximal hybridization templates.

We use blocks trees as introduced in the proof of Lemma 4.13. Let C be a c-bounded, terminating complex with n nodes, and let $H = (V, \pi, E)$ be

its hybridization graph. Up to isomorphism, a hybridization template (t, f) of C can be represented by the blocks tree of t, viewed as an abstract tree, augmented with a labeling *(i)* of each tree node (block q of π^t) with the component of C (block q' of π to which f maps q) it represents; and *(ii)* of each tree edge $\{q_1, q_2\}$ with $\{(q_1, f(v_1)), (q_2, f(v_2))\}$ where $v_1 \in q_1$ and $v_2 \in q_2$ such that $\{v_1, v_2\} \in E^t$ (this pair $\{v_1, v_2\}$ is unique, since a second such pair would imply an alternating cycle in the hybridization graph). If the hybridization template is maximal, each tree node labeled with a component D has an edge for each node of D that has an edge in the hybridization graph.

We must show that, over c-bounded complexes, there are only polynomially many such maximal augmented blocks trees. We can construct all possible augmented blocks trees using a recursive non-deterministic procedure which we describe next. The recursive step of the procedure takes as parameter a tree node q labeled by some component D. Initially, it is called on a newly created root node, labeled with a non-deterministically chosen D. There are at most n choices for D, where n is the number of nodes of C.

To describe the recursive step, we need the notion of "port". A "port" is a pair (q, u) where q is a tree node and u is a node in the component D that labels q, such that u occurs in E, i.e., has an edge in the hybridization graph. When q has an edge for u (formally, q has an edge such that the label contains (q, u)) we say that the port is "closed". Finally, note that if u is a choice node in D, then (q, u) is a port. If (q, u) is a port but u is not a choice node, then the port is called "one-way".

The recursive step is divided in two phases: the deterministic phase, followed by the choice phase. In the deterministic phase, we close all open one-way ports for q. For each such port (q, u), we take the unique node w in C such that $\{u, w\} \in E$, and let D_u be the component of C that contains this w. We create a child node r of q, label it with D_u, and label the child edge with $\{(q, u), (r, w)\}$. We say we have "closed" the open port. Note that r may have additional one-way open ports. We close those as well, and we iteratively close all open one-way ports in newly created nodes until there are no longer any open one-way ports. (This iteration must terminate since C is terminating.)

We now have a subtree rooted at q, in which there are no open one-way ports, but in which there can still be open choice ports. Since C is terminating, by Lemma 4.16, the subtree has depth at most $d = 8s + 2$. By the c-bounded restriction, the subtree has at most $\sum_{i=0}^{d} c^i$ nodes of the block tree that contain choice nodes. Hence, in total the number of open ports is at most $r = c \cdot \sum_{i=0}^{d} c^i$. Note that $r = c \cdot (c^{d+1} - 1)/(d - 1) = O(1)$. Each choice port is of the form (q', u), with q' equal to q or to a descendant of q in the tree, created during the deterministic phase. To close the choice port, there may be many possibilities. Each possibility consists of a node w in C such that $\{u, w\} \in E$;

we call w a "candidate" for u. There are at most n possible candidates. The procedure chooses a candidate w for each open choice port (q', u) and closes the port as described above for one-way ports. Then the procedure recurses on every newly created node.

Let us examine the recursion tree of this recursive procedure. Since C is terminating, by Lemma 4.16, the recursion is at most a constant $d = 8s + 2$ deep. The fan-out of the recursion tree is bounded by the constant r. Hence, each possible recursion tree arising from a non-deterministic execution of the algorithm embeds in the full tree of depth d and fan-out r, which has $\sum_{i=0}^{d} r^i = (r^{d+1} - 1)/(d - 1) = O(1)$ nodes. For each node of the recursion tree, there are at most n choices for the non-deterministic algorithm. Hence, there are $n^{O(1)}$ possible outcomes of the algorithm, which is polynomial as desired. □

Remark 4.20. Theorem 4.19 states that for c-bounded terminating complexes over a fixed alphabet, the result of hybridization has polynomial size. By the definition of hybridization, i.e., disjoint union of all finished MHE components, and Proposition 4.15, this is the same as saying that the number of finished MHE components is polynomial. Note that it is *not* true that the number of *unfinished* MHE components is polynomial. For example, for each number n, consider a complex U_n with two components: one is the strand $A \dots A$ (n times), and the other is the sticker $\overline{A}$. There are $2^n - 1$ unfinished MHE components, by choosing a strict subset of the n positive nodes, and hybridizing to each of them a copy of the sticker. There is, however, a unique finished MHE component, obtaining by hybridizing a copy of the sticker to *all* positive nodes. □

Remark 4.21. There is no converse to Theorem 4.19 in the sense that, if the result of hybridization has polynomial size over some class K of complexes over some fixed alphabet, then the complexes in K must be c-bounded for some fixed c. Take, for example, the class K consisting of all complexes L_n, for every number n, where L_n consists of four components: a strand $D \dots D$ of length 2^n; a strand $A \dots A$ of length n; and two stickers $\overline{AB}$ and $\overline{AC}$. The size of L_n is $2^n + n + 4$, and there are 2^n finished MHE components for L_n, which is a number polynomial in the size of L_n. Yet, the class K is not c-bounded for any fixed c, since L_n contains a strand with n choice nodes. □

5

DNAQL

DNAQL is an applicative programming language for expressing functions from ℓ-complexes to ℓ-complexes [20]. A crucial feature of DNAQL is that the same program can be applied uniformly to complexes of any dimension ℓ. DNAQL is not computationally complete, as it is meant as a query language and not a general-purpose programming language. The language is based on a basic set of operations on complexes, some distinguished constants, an emptiness test (if-then-else), let-variable binding, counters that can count from one up to the dimension of the complex, and a limited for-loop for iterating over a counter. The syntax of DNAQL is given in Figure 5.1. Note that expressions can contain two kinds of variables: variables standing for complexes, and counters, ranging from 1 to the dimension. Complex variables can be bound by let-constructs, and counters can be bound by for-constructs. The free (unbound) complex variables of a DNAQL expression stand for its inputs. A DNAQL *program* is a DNAQL expression without free counters. So, in a program, all counters are introduced by for-loops.

The constant expressions provide particular complexes as constants. A word $w \in (\Sigma - \Lambda)^+$ stands for a single, linear, positive strand that spells the word w. A two-letter word $\overline{AB}$, for $a, b \in \Sigma - \Lambda$, stands for a single, linear, negative strand of length two of the $1 \to 2$ with $\lambda(1) = \overline{B}$ and $\lambda(2) = \overline{A}$. The expression $\texttt{immob}(\overline{A})$, for $A \in \Sigma$, stands for a single, negative, immobilized node labeled $\overline{A}$. If $\overline{a} \in \overline{\Lambda}$ 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 as-*

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

Figure 5.1: Syntax of DNAQL.

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

The semantic rules that define this evaluation are shown in Figures 5.2 and 5.3. The superscript ℓ has been omitted in the figure to reduce clutter. The rules for `let` and `for` use the oft-used notation $f[x := u]$ to denote the mapping f updated so that x is mapped to u. Because the operations on complexes are not always defined, the evaluation may fail, so $[\![e]\!]^\ell(\nu, \gamma)$ may be undefined. When e is a program, we denote $[\![e]\!]^\ell(\nu, \emptyset)$ simply by $[\![e]\!]^\ell(\nu)$.

Example 5.1. We give an example of a DNAQL program, over the input variables x_1 and x_2, with a behavior similar to the selection operator and the cartesian product operator from the relational algebra. Below, a and b are assumed to be atomic value symbols.

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

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

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

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

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

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

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

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

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

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

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

Figure 5.2: DNAQL semantics Part 1

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

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

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

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

Figure 5.3: DNAQL Semantics Part 2

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

Example 5.2. Assume a DNAQL expression e resulting in a complex C consisting of strands representing tuples with a single attributes A. Concretely, each strand in C is of the general form $\#_2A\#_3t(A)\#_4$. We assume two atomic value symbols in this example, viz., 0 and 1, hence the values $t(A)$ on the strands are in a binary code. The following program computes whether there is a strand in complex C such that the value of attribute A is a sequence of 0's.

$$\texttt{for } x := e \texttt{ iter i do}$$
$$\texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(\texttt{blockexcept}(x, \texttt{i}) \cup \texttt{immob}(\overline{0}))))$$

Essential to this computation is the for-loop iterating counter variable `i` from 1 to ℓ. Indeed, eliminating the for-loop from this program, implies dropping the blockexcept operator, as the counter variable would no longer be bound. Consequently, the program has a significantly altered semantics. Without the blockexcept operator, the probe labeled $\overline{0}$ can attach to any atomic value symbol in the value of attribute A. Hence, the program would then compute whether there is a strand with a 0 in its value for attribute A.

The combination of a for-loop and the blockexcept operator ensure that only the $\texttt{i}^{th}$ atomic value symbol is free, whereas all other atomic value symbols are blocked. As a result, each atomic value symbol is probed in turn, from the

first to the last value. After blocking all atomic value symbols except the $\mathtt{i}^{th}$, a probe labeled $\overline{0}$ is added to the complex. Hybridization is performed so that the probe attaches to strands with 0 as $\mathtt{i}^{th}$ atomic value symbol. Next, the flush operator removes all non-immobilized components, i.e., the strands with a 1 as $\mathtt{i}^{th}$ atomic value symbol. Finally, the cleanup operator removes the probe and unblocks any remaining strands. The remaining strands forms the input to the next iteration. At the end of each iteration of the for-loop, the current complex contains zero or more strands. Only if there is a strand with all 0's, the last iteration will retain a single strand. Otherwise, the output is empty. □

6

Sticker Complex Types

Chapter 3 we introduced the sticker complexes and defined a number of operations on them. Some operations can be applied on any sticker complex, for example, flush, ligate, or union. Other operations place restrictions on the sticker complexes they act on, for example, hybridization is only defined on terminating complexes and the block operations are only defined on saturated complexes. Most of the conditions posed by operations can be checked easily and efficiently. Determining termination of hybridization is more complex, yet still efficient as we discussed in Chapter 4. An applicative programming language allows us to string multiple operations together. Hence, the question arises whether a given DNAQL program applied on given types of inputs results in a defined result? Determining whether a program is "safe" to execute is a problem encountered all but trivial programming languages. The standard approach to dealing with this issue is introducing a type system. A type system strips the values handled by a programming language to their barest form to distinguish defined from undefined without performing the computation expressed by a program. For example, the type system of the Java programming language will abstract any natural number to the type `int`. This is enough information to detect errors that make the output of programs undefined. We take a similar approach in defining types for sticker complexes, called sticker complex types. A good overview of programming languages and type systems can be found in [35, 22].

Intuitively, a sticker complex type is an ℓ-complex where all data entries have been replaced by wildcards. What remains is a structural description of the components that may appear in the complex, with attribute names and

tags explicit, but the dimension and actual values of data entries hidden. In order to obtain a powerful type-checking algorithm for DNAQL, these "weak" types S are augmented to "strong" types that have an indication $\odot$ of the mandatory components, which must occur, and a bit $\mathfrak{h}$ indicating that all the complexes of a strong type are saturated. The former is needed to type common DNAQL programs that use hybridization, and the latter is needed to type blocking operators in a DNAQL program.

6.1 Definition

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

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

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

The fourth new symbol, denoted by '?' will be used to represent a single negative atomic value symbol that has been immobilized, i.e., a probe. 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 a free or an open ℓ-core. As a result, the complementarity relation is no longer a bijection. Note that $\underline{*}$ has no complementary symbol.

A *sticker complex type* is very similar to a sticker complex; it is a structure $S = (V, L, \lambda, \mu, \iota, \beta)$ that satisfies the same definition as that of a sticker complex with the following exceptions:

- the range of the node labeling function λ is now $\Sigma_N \cup \overline{\Sigma_N}$ instead of $\Sigma \cup \overline{\Sigma}$;
- $\beta \subseteq V$ is not allowed to contain nodes labeled with a symbol from N;
- there are no redundant components (recall the definition of redundancy from Chapter 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

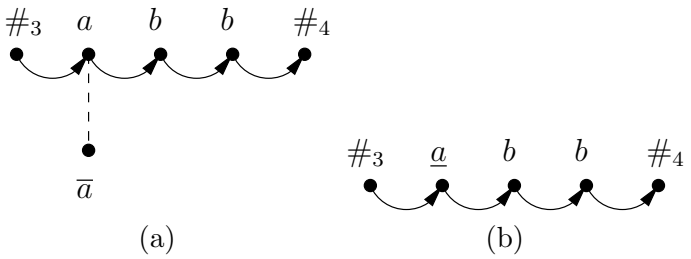


Figure 6.1: Two ill-typed complexes. Blocking is shown by underlining a symbol. The complex in (a) has a negative atomic value symbol that is not immobilized. This is prohibited by the definition of sticker complexes. The complex in (b) has one blocked and two non-blocked symbols, this corresponds neither to $*$, $\hat{*}$ nor $\underline{*}$.

- r is of type $*$ if no node of r belongs to β, and at most one node of r is involved in μ;
- r is of type $\underline{*}$ if all nodes of r belong to β;
- r is of type $\hat{*}$ if all nodes of r but one belong to β.

Now we say that C is *well typed* if

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

Example 6.1. Figure 6.1 shows two ill-typed complexes. The first complex is ill typed because it contains a negative atomic value symbol ($\overline{a}$) that is not immobilized. The is prohibited by the definition of sticker complexes. 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 sticker complex type obtained by:

- contracting every ℓ-core occurring in C to a single node labeled by the type of the ℓ-core ($*$, $\underline{*}$ or $\hat{*}$);
- replacing the label of a node labeled with an immobilized negative atomic value by ?;

– when a node from an ℓ-core r in C is matched by μ to a node u, then in *stype*(C) the single node representing r is matched to u. Note that, by the previous item, in *stype*(C) node u has label ?. Furthermore, the node representing r is labeled $*$ or $\hat{*}$.

Note that the subsumption relation among sticker complexes, defined in Chapter 3, can be adopted naturally to sticker complex types. We have the following lemma.

Lemma 6.2. *Let C_1 and C_2 be well-typed complexes. If $C_1 \sqsubseteq C_2$, then* *stype*$(C_1) \sqsubseteq$ *stype*(C_2). *If* *stype*$(C_1) \sqsubseteq$ *stype*(C_2) *and* *stype*(C_1) *does not contain nodes labeled by symbols of $N = \{*, \hat{*}, \underline{*}, ?\}$, then $C_1 \sqsubseteq C_2$.*

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

We will often use these properties without mention. For a well-typed complex C and a complex type S, we now say that C *has type* S, denoted by $C : S$, if *stype*(C) is subsumed by S. For complex C, *stype*(C) is the "smallest" type, in the sense that there is no complex type S' such that $C : S'$ and S' is strictly subsumed by S.

Example 6.3. Figure 6.2 shows a sticker complex C of dimension 2, and a sticker complex type S. Structurally, C and S are very alike. There are two differences: (i) 2-cores are contracted to one node labeled $*$, and (ii) as the second and third strand of C only differ in their respective 2-cores, only one strand (the bottom strand of S) is needed to represent both. Sticker complex type S is *stype*(C) and is thus the smallest type for C. □

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

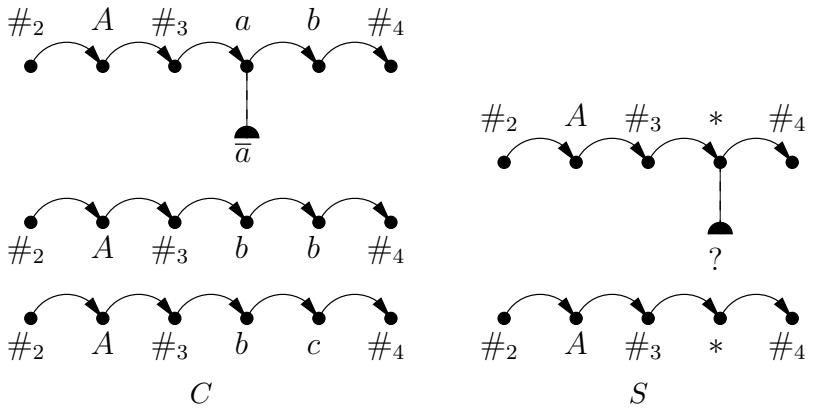


Figure 6.2: A sticker complex C and a sticker complex type S such that C has type S.

Figure 6.3: A sticker complex type with two single-node components.

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

Example 6.4. Consider the sticker complex type τ with the weak type shown in Figure 6.3. Assume that the component on the left is the only mandatory component and that $\mathfrak{h} = \mathit{true}$. Then the component on the right is "garbage", in the sense that any complex having type τ cannot contain a node labeled with $\overline{A}$, because such a complex will not be saturated. Indeed, because the component on the left is mandatory, each complex having type τ must contain a node labeled A. This is the raison d'être for the condition that for all $C \in \mathit{comp}(S)$, $C \cup \odot$ has to be saturated if $\mathfrak{h} = \mathit{true}$. □

Example 6.5. 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

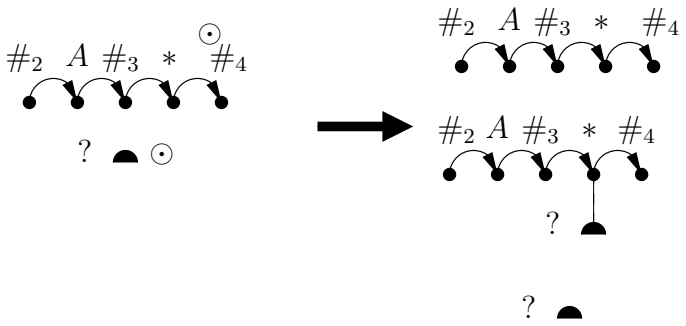


Figure 6.4: A type with two mandatory components on the left. On the right is the hybridization of the type on the left. Despite the fact that all components start as mandatory, the hybridization contains only non-mandatory components.

about types, the $\mathfrak{h}$-bit introduces some subtle modeling options. For example, recall the weak type S shown in Figure 6.3. Suppose a type τ, with weak type S, $\odot = \texttt{empty}$ and $\mathfrak{h} = \mathit{true}$. There are three complexes having type τ: the empty complex, the complex consisting of the component on the left and the complex consisting of the component on the right. The complex consisting of both components is not saturated and thus prohibited by the $\mathfrak{h}$-bit. □

Example 6.6. Consider the complex in Figure 6.4, on the left. Although both components are mandatory (indicated by the $\odot$), we will see that the hybridization of this type consists of three non-mandatory components (Figure 6.4 on the right). Let us call this type $\tau = (S, \odot, \mathfrak{h})$. The $\mathfrak{h}$-bit of the resulting type is *true*. This has important repercussions on the set of complexes having this type. Indeed, consider the complex in Figure 6.5. This complex does *not* have type τ, but it has type $\tau' = (S, \odot, \mathit{false})$. □

6.2 Saturated

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

Lemma 6.7. *Type $\tau = (S, \odot, \mathfrak{h})$ is saturated if and only if S is saturated or $\mathfrak{h} = \mathit{true}$.*

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

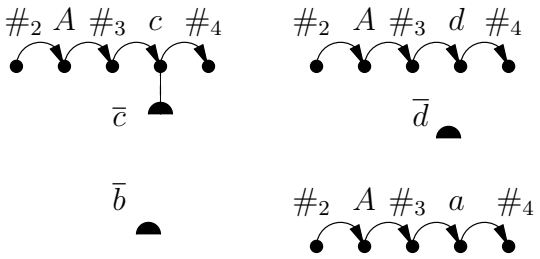


Figure 6.5: A complex with five components. This complex does not have the type on the right of Figure 6.4.

$C \in [\![\tau]\!]$ we know that $\mathit{stype}(C) \sqsubseteq S$. If a complex subsumed in S is not saturated, S itself cannot be saturated.

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

1. $\lambda(u) \notin N$: node u is not labeled with $*$, $\underline{*}$, or $\hat{*}$. Replacing the ℓ-cores in $\mathit{stype}(C)$ by any sequence of atomic value symbols, and replacing all the probes by an arbitrary negative atomic value symbols, results in a complex C that is not saturated.

2. $\lambda(u) \in N$: node u is labeled either with $*$ or with $\hat{*}$. Consequently, v is labeled with ?. Fix an atomic value symbol, say $a \in \Lambda$. We let complex C be the complex in which all ℓ-cores are replaced by a^ℓ and all probes, i.e., ?, are replaced by $\overline{a}$.

□

Note that, as a consequence of Lemma 6.7, saturatedness of a type is decidable in polynomial time. Indeed, simply iterating over all pairs of nodes is sufficient to find out whether a complex is saturated or not.

6.3 Subtypes

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

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

Example 6.8. Recall the type $\tau = (S, \odot, \mathit{true})$ on the right of Figure 6.4. Let type $\tau' = (S, \odot, \mathit{false})$. Notwithstanding the fact that both τ and τ' have the same weak type and the same set of mandatory components, we have that $\tau \preceq \tau'$ but not $\tau' \preceq \tau$, because the complex shown in Figure 6.5 has type τ' but does not have type τ. □

The notion of subtyping is defined semantically. However, a type can have an infinite number of complexes. An efficiently decidable syntactic characterization of subtyping is thus called for. Proposition 6.9 provides such a characterization.

Proposition 6.9. *Let $\tau = (S, \odot, \mathfrak{h})$ and $\tau' = (S', \odot', \mathfrak{h}')$ be types. Type τ is a subtype of τ' if and only if (i) $S \sqsubseteq S'$; (ii) $\odot' \sqsubseteq \odot$; and (iii) if $\mathfrak{h}' = \mathit{true}$ then τ is saturated.*

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

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

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

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

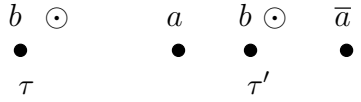


Figure 6.6: Types $\tau = (S, \odot, \mathit{false})$ and $\tau' = (S', \odot', \mathit{true})$ with $\tau \preceq \tau'$.

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

Lemma 6.7 implies that the notion of saturated for types is decidable in polynomial time, and therefore the notion of subtype is decidable in polynomial time.

We have the following corollary to Proposition 6.9.

Corollary 6.10. *Let $\tau = (S, \odot, \mathfrak{h})$ and $\tau' = (S', \odot', \mathfrak{h}')$ be types. Types τ and τ' are equivalent if and only if (i) $S \equiv S'$; (ii) $\odot \equiv \odot'$; and (iii) if $S \equiv S'$ is not saturated, then $\mathfrak{h} = \mathfrak{h}'$.*

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

By Lemma 6.7, $\tau = (S, \odot, \mathfrak{h})$ is saturated iff S is saturated or $\mathfrak{h} = \mathit{true}$ (and similarly for τ'). Hence, if $S \equiv S'$ is saturated, then conditions (iii) and (iv) hold trivially. If $S \equiv S'$ is not saturated, then condition (iii) says if $\mathfrak{h}' = \mathit{true}$, then $\mathfrak{h} = \mathit{true}$, and condition (iv) says if $\mathfrak{h} = \mathit{true}$, then $\mathfrak{h}' = \mathit{true}$. Consequently, $\mathfrak{h} = \mathfrak{h}'$ in this case. □

Obviously, type $\tau = (S, \odot, \mathit{true})$ is a subtype of the type $\tau' = (S, \odot, \mathit{false})$. On the other hand, by Corollary 6.10, $\tau' = (S, \odot, \mathit{false})$ may also be a subtype of τ when S is saturated.

Example 6.11. Figure 6.6 shows types $\tau = (S, \odot, \mathit{false})$ and $\tau' = (S', \odot', \mathit{true})$ with $\tau \preceq \tau'$. Since S is saturated, setting $\mathfrak{h} = \mathit{true}$ in τ yields a type equivalent to τ. □

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

Lemma 6.12. *Let C be a complex and τ a type. Then $C : \tau$ iff $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C) \preceq \tau$ with $\mathfrak{h}_C = \mathit{true}$ iff C is saturated.*

Proof. Let $\tau = (S, \odot, \mathfrak{h})$. Suppose $C : \tau$, then we show that $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C) \preceq \tau$. By $C : \tau$ we know that (i) $\mathit{stype}(C) \sqsubseteq S$, (ii) $\odot \sqsubseteq \mathit{stype}(C)$,

and (iii) if $\mathfrak{h} = \mathit{true}$, then C is saturated. Because, $\mathfrak{h}_C$ is defined as *true* if C is saturated, these are also the conditions for $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C) \preceq \tau$ as shown in Proposition 6.9.

Next, we show that if $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C) \preceq \tau$, then $C : \tau$. By Proposition 6.9 we know that $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C) \preceq \tau$ implies that (i) $S \sqsubseteq \mathit{stype}(C)$, (ii) $\odot \sqsubseteq \mathit{stype}(C)$, and (iii) if $\mathfrak{h} = \mathit{true}$, then ($\mathit{stype}(C)$, $\mathit{stype}(C), \mathfrak{h}_C$) is saturated. Items (i) and (ii) are the first condition for $C : \tau$. From Lemma 6.7 we know that if $(\mathit{stype}(C), \mathit{stype}(C), \mathfrak{h}_C)$ is saturated, then $\mathit{stype}(C)$ is saturated or $\mathfrak{h}_C = \mathit{true}$. If $\mathit{stype}(C)$ is saturated, then C is saturated. If $\mathfrak{h}_C = \mathit{true}$, then by definition C is saturated. Hence, we may conclude $C : \tau$. □

6.4 Least upper bound

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

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

Proposition 6.13. *Let* $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ *and* $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ *be types. The least upper bound of* τ_1 *and* τ_2 *exists and is equivalent to* $(S_1 \cup S_2, \odot_1 \cap \odot_2, \tau_1$ *saturated* $\wedge\, \tau_2$ *saturated*).

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

Now we must show that $\tau = (S_1 \cup S_2, \odot_1 \cap \odot_2, \tau_1$ saturated $\wedge \tau_2$ saturated) is the *least* upper bound. Let $\tau' = (S', \odot', \mathfrak{h}')$ be a type. Type τ' is an upper bound of τ_1 and τ_2 if $\tau_i \preceq \tau'$ for $i \in \{1, 2\}$. By Proposition 6.9, $\tau_i \preceq \tau'$ 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 6.9, τ' is an upper bound of τ_1 and τ_2 iff $\tau \preceq \tau'$. □

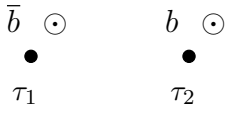


Figure 6.7: Types $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$, having no mandatory components in common. As a result, $\tau_1 \vee \tau_2 = (S_1 \cup S_2, \texttt{empty}, \mathit{true})$ allows the empty complex, whereas the empty complex is not part of $[\![\tau_1]\!]$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

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

6.5 Greatest lower bound

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

Proposition 6.14. *Let $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ be types. Then a lower bound of τ_1 and τ_2 exists iff both $\odot_1 \cup \odot_2 \sqsubseteq S_1 \cap S_2$ and if either τ_1 or τ_2 is saturated, then $\odot_1 \cup \odot_2$ is saturated. If a lower bound of τ_1 and τ_2 exists, then there is a greatest lower bound τ, and τ is equivalent to $\tau_g = (S_1 \cap S_2 - Z, \odot_1 \cup \odot_2, \tau_1$ saturated $\vee\, \tau_2$ saturated$)$, where $Z = \{C \in comp(S_1 \cap S_2) \mid C \cup \odot_1 \cup \odot_2$ is saturated$\}$ if either τ_1 or τ_2 is saturated, and $Z = \emptyset$ otherwise.*

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

Let $\tau = (S, \odot, \mathfrak{h})$ be a type. Then, by Proposition 6.9, $\tau \preceq \tau_i$ iff $S \sqsubseteq S_i$, $\odot_i \sqsubseteq \odot$, and if $\mathfrak{h}_i = \mathit{true}$, then τ is saturated. Hence, τ is a lower bound of τ_1 and τ_2 iff $S \sqsubseteq S_1 \cap S_2$, $(\odot_1 \cup \odot_2) \sqsubseteq \odot$, and if $\mathfrak{h}_1 = \mathit{true}$ or $\mathfrak{h}_2 = \mathit{true}$, then τ is saturated. By Corollary 6.10, if τ_i is saturated and $\mathfrak{h}_i = \mathit{false}$, then the type τ_i' obtained from τ_i by setting $\mathfrak{h}_i$ to *true* is equivalent to τ_i. Thus, τ is a lower bound of τ_1 and τ_2 iff $S \sqsubseteq S_1 \cap S_2$, $(\odot_1 \cup \odot_2) \sqsubseteq \odot$, and if τ_1 or τ_2 is saturated, then τ is saturated. Hence, τ is a lower bound of τ_1 and τ_2 iff both (1) $\odot_1 \cup \odot_2 \sqsubseteq S_1 \cap S_2$, (2) if either τ_1 or τ_2 is saturated, then $\odot_1 \cup \odot_2$ is saturated, and (3) $\tau \preceq \tau_g$, where the $\mathfrak{h}$-bit τ_1 *saturated* $\vee\, \tau_2$ *saturated* of τ_g follows from Proposition 6.9. □

Example 6.15. Types τ_1 and τ_2 from Figure 6.7 do not have a greatest lower bound. Indeed, $S_1 \cap S_2$ is the empty complex, while the weak type $\odot_1 \cup \odot_2$ contains two components. □

6.6 Operations on Sticker Complex Types

We have defined a set of operations on complexes. The type system will mimic the structural changes, effected by the operations on complexes, on types. Therefore we define the set of operations, introduced in Section 3.4, on types.

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

Union

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

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

Difference

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

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, non-circular. Furthermore, all strands have the same length and the same number of $*$-labeled nodes.

3. Each strand of S_2 ends with $\#_4$ and does not contain $\#_5$.

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

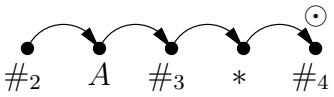


Figure 6.8: A hybridized, strong type with a single mandatory component.

Table 6.1: Two complexes having the type depicted in Figure 6.8.

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

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

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

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

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

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

Example 6.16. Figure 6.8 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 6.1. On the type-level, the cases $C_1 - C_1$ and $C_1 - C_2$ are indistinguishable, however, the resulting complexes are definitely different. The output of $C_1 - C_1$ is the empty complex, whereas the output of $C_1 - C_2$ is the complex containing the strand $\#_2A\#_3a\#_4$. In other words, the values of data strands (strands with a node labeled $*$) are unknown on the type-level, consequently, they are preserved in the output type. □

Hybridize

The hybridize operator on sticker complexes can naturally be adapted to weak types by incorporating the extended complementarity relation, i.e., with $\overline{*} =?$

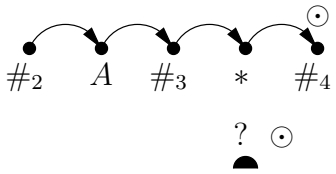


Figure 6.9: A type τ with two mandatory components.

and $\bar{\hat{*}} =?$ as legal matchings. Denote this adjusted version by $\texttt{hybridize}_t$.

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

Assume $\mathfrak{h} = \mathit{false}$. If hybridization does not terminate for S, then the hybridization of τ is not defined.

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 set of necessary components of τ. The hybridization of τ, denoted $\texttt{hybridize}(\tau)$, equals $(Cs, \odot_h, \mathit{true})$, where

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

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

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

Example 6.17. Consider type τ displayed in Figure 6.9. Type τ' equals $\texttt{hybridize}(\tau)$ and is shown in Figure 6.10 (except for the $\mathfrak{h}$-bit which is always *true*). Note that the weak type of τ' consists of three components, all of which are not mandatory. □

Ligate & Flush

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

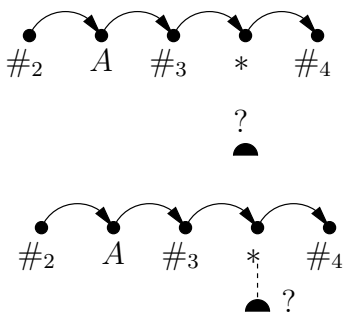


Figure 6.10: Type $\texttt{hybridize}(\tau)$, where τ is from Figure 6.9.

Split

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

Block

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

Block-From

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

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

Block-Except

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

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

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

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

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

Cleanup

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

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

which all blockings have been cleared and $\hat{*}$- and $\underline{*}$-labeled nodes are relabeled to $*$.

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

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

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

Lemma 6.18. *Let $\tau = (S, \odot, \mathfrak{h})$ be a type; let ℓ be a natural number; let C be an ℓ-complex of type τ; let d be a strand belonging to $\texttt{cleanup}(C)$; and let $s = stype(d)$. Then s qualifies for S_{clean}.*

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

7

A Type System for DNAQL

In this chapter we introduce a type system for DNAQL and we show that it enjoys the desirable properties of soundness, maximality, and tightness. Intuitively, a safe type system ensures that if a program type checks, then the program will not exhibit predefined erroneous behavior. A maximal type system only refuses programs if there is a reason to, i.e., there a situation in which the program crashes. A tight type system returns a type that cannot be made more precise, i.e., represent less values.

7.1 Type System

A DNAQL expression e has a set of free variables, denoted $FV(e)$. If a type is fixed for each free variable, all the *complexvar*-subexpressions of e are *well typed* and their types are known. The *constant*-subexpressions of e are always well typed, and their types are known (cf. Figures 7.1 and 7.2). In the previous chapter we defined for each DNAQL operator, its counterpart operating on types. In this chapter 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 Figures 7.1 and 7.2. Here we write $\Gamma \vdash e : \tau$ to indicate that expression e *is assigned* type τ under type

assignment Γ on e. If $\Gamma \vdash e : \tau$, then we call (Γ, τ) a *typing* of e. The domain of Γ is extended from variables to expressions as specified in Figures 7.1 and 7.2. The typing relation given in Figure 7.1 and 7.2 is clearly unambiguous, i.e., if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$. Note that conditions of the typing-rules of the if statement are mutual exclusive.

Recall the formal semantics of DNAQL in Chapter 5. When ℓ is not important, we refer to an ℓ-complex assignment simply as a complex assignment. Let Γ a type assignment, and let ν be a complex assignment. We naturally say that ν *has type* Γ if $dom(\nu) = dom(\Gamma)$ 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 $[\![\Gamma]\!]$.

7.2 Sound

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

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

Since types do not restrict the dimension of complexes, if a type involves wildcards, there are infinitely many complexes of that type. Hence safety is not easy to guarantee, indeed safety is undecidable: this will follow from Theorem 8.2 and an easy reduction from satisfiability of well-typed relational algebra expressions, which is undecidable [1].

The best we can do is to come up with a type system that tries to infer output types from the given input types. The type-checking algorithm induced by Figures 7.1 and 7.2, given e and Γ as above, judges whether e is *well typed* under Γ, and, if so, infers its output type τ. This is denoted by $\Gamma \vdash e : \tau$.

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

Theorem 7.1. *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.

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{cleanup}(e) : \texttt{cleanup}(\tau)}$$

Figure 7.1: Typing relation of DNAQL part 1

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

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

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

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

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

Figure 7.2: Typing relation of DNAQL part 2.

Variable. Let $e = x \in dom(\nu)$ be a variable. By Figure 7.1, $\Gamma \vdash x : \Gamma(x)$. Hence $\nu(x) : \Gamma(x) = \tau$. Consequently, $[\![e]\!](\nu, \gamma) = \nu(x) : \tau$ as required.

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

Union. Let $e = e_1 \cup e_2$. By induction, we assume that $C_1 = [\![e_1]\!](\nu, \gamma)$ and $C_2 = [\![e_2]\!](\nu, \gamma)$ are defined and they are of types $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$. By definition the union of two sticker complexes is defined and so $C = C_1 \cup C_2 = [\![e]\!](\nu, \gamma)$ is defined. We have $\tau = \tau_1 \cup \tau_2 = (S_1 \cup S_2, \odot_1 \cup \odot_2, \mathfrak{h})$.

It suffices to show that C is of type τ. We verify the three conditions in the definition a complex having a particular type.

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

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

If $\mathfrak{h} = true$, then by definition of union on types, both (1) S_1 and S_2 are mutually non-interacting, and (2) S_i is saturated or $\mathfrak{h}_i = true$ for

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

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

First we prove that $C = [\![e]\!](\nu, \gamma)$ is defined. The definition of difference on sticker complexes imposes three restrictions on the sticker complexes C_1 and C_2. We prove that each restriction is met.

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

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

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

Let $s \in \mathit{comp}(\odot)$. By definition of $\odot$, $s \in \mathit{comp}(\odot_1)$ and is not subsumed by S_2. Moreover, since $s \cong \mathit{stype}(D)$ is not subsumed by S_2, but $\mathit{stype}(C_2)$ is subsumed by S_2, we have by Lemma 6.2 that D is not

subsumed by C_2. Thus $D \in \mathit{comp}(C)$ whence s is subsumed by $\mathit{stype}(C)$ as desired.

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

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

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

Assume now that $\mathfrak{h}' = \mathit{false}$. Since τ is defined, hybridization terminates for weak type S' (i.e., $\texttt{hybridize}_t(S')$ is defined). By Theorem 4.1 there is no alternating cycle in the hybridization graph of S' (the definition of hybridization graph is straightforwardly extended to sticker complex types by using the extended complementarity relation). Consequently, there is no alternating cycle in the hybridization graph of C', and therefore C' has terminating hybridization (ℓ-cores and ?-labeled probes can never engage in an alternating cycle). Hence $[\![e]\!](\nu, \gamma)$ is well defined.

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

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

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

1. $\mathcal{D}$ contains no probe. Note that $\mathcal{D}$ may contain nodes labeled from $\bar{\Lambda}$, but then these are already matched in C'. In this case $\mathit{stype}(D) \in \mathit{comp}(\texttt{hybridize}_t(\mathit{stype}(C') \setminus \{\texttt{immob}(?)\}))$. Since $\odot' \sqsubseteq \mathit{stype}(C') \sqsubseteq S'$, we can view $\mathit{stype}(C') \setminus \texttt{immob}(?)$ as an X such that $NC \sqsubseteq X \sqsubseteq S'$. Hence $\mathit{stype}(D)$ is clearly subsumed by S in this case.
2. $\mathcal{D}$ consists of a probe. In this case $\mathit{stype}(D) \equiv \texttt{immob}(?)$. In particular, $\texttt{immob}(?)$ occurs as a separate component in $\mathit{stype}(C')$ which is in turn subsumed by S'. Hence, again $\mathit{stype}(D)$ is subsumed by S in this case.
3. $\mathcal{D}$ contains a probe in addition to other components of C'. Since D is a component, the probe is involved in the matching that creates D. Note also that $\mathcal{D}$ contains exactly one probe, since probes are immobilized and components of sticker complexes can

contain at most one immobilized node. The *stype* of the probe is $\texttt{immob}(?)$, and the *stype* of the component containing the core r having the atomic value node that is matched to the probe has a node representing r that is labeled by $*$ or $\hat{*}$. Since both $(*, ?)$ and $(\hat{*}, ?)$ are complementary pairs of symbols, we conclude that $\mathit{stype}(D) \in \mathit{comp}(\texttt{hybridize}_t(\mathit{stype}(C')))$. As in Case 1, we can see $\mathit{stype}(C')$ as an X such that $NC \sqsubseteq X \sqsubseteq S'$. Hence $\mathit{stype}(D)$ is subsumed by S.

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

1. Assume case (1) holds. Since $s \in \mathit{comp}(\texttt{hybridize}_t(NC))$, 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 $\odot'$ is subsumed by $\mathit{stype}(C')$, 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 $\mathit{stype}(C)$.

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

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

The operation ligate is defined on all complexes, thus $[\![e]\!](\nu, \gamma)$ is defined.

On types, operation `ligate` is defined as performing ligate on the weak type S and on the mandatory weak type. Ligate does not change the state of $\mathfrak{h}'$. From this it is clear that $[\![e]\!](\nu, \gamma)$ is of type τ.

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

The flush operation is defined on any complex. As a result, $C = \texttt{flush}(C') = [\![e]\!](\nu, \gamma)$ is defined.

Let $D \in \mathit{comp}(C)$. Then $D \in \mathit{comp}(C')$ and $\iota_D \neq \emptyset$, where ι_D is the set of immobilized nodes of D. Since C' is of type τ', there is a $t \in \mathit{comp}(S')$ with $\iota_t \neq \emptyset$ such that $t \equiv \mathit{stype}(D)$. Hence $\mathit{stype}(D)$ is subsumed by S.

Let $s \in \mathit{comp}(\odot)$. Then $s \in \mathit{comp}(\odot')$ and $\iota_s \neq \emptyset$. Since $\odot'$ is subsumed by $\mathit{stype}(C')$, there is a $D \in \mathit{comp}(C')$ such that $s \cong \mathit{stype}(D)$. Since $\iota_s \neq \emptyset$, also $\iota_D \neq \emptyset$, whence $D \in \mathit{comp}(C)$ and thus s is subsumed by $\mathit{stype}(C)$ as desired.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cleanup. Let $e = \texttt{cleanup}(e')$. By induction, we assume that $C' = [\![e']\!](\nu, \gamma)$ is defined and has type $\tau' = (S', \odot', \mathit{true})$. Let $\tau = \texttt{cleanup}(\tau') = (S, \odot, \mathit{true})$.

The cleanup operation is defined on all complexes, so we must only verify that $C = \texttt{cleanup}(C')$ is of type τ.

Let D be a component of C. Then D is a strand of length m with m the length of the longest positive strand in C'. By Lemma 6.18 (applied to C') we obtain that $\mathit{stype}(D)$ qualifies for $S = S'_{\mathit{clean}}$. Hence $\mathit{stype}(D)$ belongs to S whence $\mathit{stype}(D)$ is subsumed by S as desired.

Let $s \in \mathit{comp}(\odot)$. Consequently, from the definition of `cleanup` on types, it is known that there is a component $D \in \omega(s)$ such that $D \in \mathit{comp}(\odot')$, s qualifies for S'_{clean} and s qualifies for mandatory. By $D \in \mathit{comp}(\odot')$ there is a component $E \in \mathit{comp}(C')$ such that $D \equiv \mathit{stype}(E)$. In particular, there is a strand $d \in \mathit{strands}(E)$ such that $s \equiv \mathit{stype}(d)$. It remains to be shown that $d \in \mathit{comp}(C)$. Thereto, we must show

that the length of d is greater than or equal to the length of d' for any $d' \in \mathit{strands}(C')$. Note that the length of d' is smaller than or equal to the length of $d_{\max}$ for any $d_{\max} \in \mathit{strands}(C)$. By Lemma 6.18, $\mathit{stype}(d_{\max})$ qualifies for S, whence $n(s)+(\ell-1)a(s) \geq n(\mathit{stype}(d_{\max}))+(\ell-1)a(\mathit{stype}(d_{\max}))$. Since the number on the left-hand side equals the length of d, and the number on the right-hand side equals the length of $d_{\max}$, we are done.

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

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

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

If. Let $e = \texttt{if empty}(x)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. 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 7.2. Recall the program from Example 5.1 in Chapter 5:

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

Consider the weak types $S_1 = \#_3{*}\#_4\#_5$ and $S_2 = \#_1\#_3{*}\#_4$. The program is well-typed under the types $\tau_1 = (S_1, S_1, \mathit{false})$ for x_1 and $\tau_2 = (S_2, \emptyset, \mathit{false})$ for x_2. Since S_1 is mandatory in τ_1, we know that input x_1 will be nonempty. Note also that the $\mathfrak{h}$-bit in τ_1 is *false*, although complexes of type S_1 are necessarily saturated; so we are making it hard on the type checker. The subexpression $e_1 = \texttt{hybridize}(x_1 \cup \texttt{immob}(\overline{a}))$ is typed as $(S_1^?, \emptyset, \mathit{true})$, where $S_1^?$ consists of the following components: (i) S_1 itself; (ii) $\texttt{immob}(?)$; and (iii) the complex formed by the union of (i) and (ii) and matching the node $*$ with the node ?. Note that there are no mandatory components, since on inputs without an a, only (i) and (ii) will occur, whereas on inputs where all strands have an a, only (iii) will occur. The $\mathfrak{h}$-bit is now *true* since a complex resulting from hybridization is always saturated.

Applying `flush` to e_1 yields output type $(S_1^{?\prime}, \emptyset, \mathit{true})$, where $S_1^{?\prime}$ consists of components (ii) and (iii) above. Finally the variable y_1 in the `let`-construct is assigned the type $(S_1, \emptyset, \mathit{true})$. Similarly, y_2 gets the type $(S_2, \emptyset, \mathit{true})$. Yet, by the design of the if-then-else typing rules, the subexpression on the last line of the program will be typed under the strong types $(S_1, S_1, \mathit{true})$ for y_1 and $(S_2, S_2, \mathit{true})$ for y_2. Because all components are now mandatory, the type inferred for subexpression $\texttt{hybridize}(y_1 \cup y_2 \cup \overline{\#_5\#_1})$ will be $(S_{12}, S_{12}, \mathit{true})$, where S_{12} is the weak type obtained from the union of S_1, S_2 and $\overline{\#_5\#_1}$ by matching the $\#_5$ and $\overline{\#}_5$ and the $\#_1$ and $\overline{\#}_1$ nodes, respectively. After ligate and cleanup the output type is (S, S, true) where S consists of the single strand $\#_3{*}\#_4\#_5\#_1\#_3{*}\#_4$. The final output type of the entire program, combining the then- and else-branches, is $(S, \emptyset, \mathit{true})$. □

Example 7.3. For another example, consider the program

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

This program is ill-typed under the type $\tau = (S, S, \mathit{true})$ for x with $S = \#_3{*}\#_4$. Indeed, the nested hybridize subexpression is still well-typed, yielding the output type $(S^?, \emptyset, \mathit{true})$ without any mandatory components. Adding

the component $\overline{\#_3\#_4}$ to $S^?$, however, yields a complex with non-terminating 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 one immobilized node.

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

7.3 Maximal

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

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

Theorem 7.4. *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. We will show that if atomic expression e is ill-typed under Γ, then e is not safe under Γ, i.e., there exists a specific input complex assignment of type Γ on which the evaluation of e is undefined.

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

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

1. Suppose there is a node x that is matched, immobilized, blocked or labeled with $\underline{*}$ or $\hat{*}$ in τ_1 or τ_2. Recall that a node labeled with $\underline{*}$

or $\hat{*}$ represents a (possibly partially) blocked ℓ-core, and so every complex having such a type will have a blocked node (as $\ell \geq 2$ by definition, there will also be blocked node in case of $\hat{*}$). Suppose x is present in τ_1 (the proof is similar if x is in τ_2). Let D be the component of S_1 containing x. Let C be a complex such that $\mathit{stype}(C) = D \cup \odot_1$. Complex C has type τ_1 and therefore has a matching, blocking, or immobilization. So difference is not defined.

2. This case will be split into two sub cases: (a) there is a negative or circular strand in $\mathit{strands}(S_1)$ or $\mathit{strands}(S_2)$, and (b) assuming that there are only positive non-circular strands, two strands s_1 and s_2 in $\mathit{strands}(S_1) \cup \mathit{strands}(S_2)$ are of different length or have a different number of $*$-labeled nodes.
   (a) Let d be a strand in S_1 (S_2) that is negative or circular, and let D be the component in which d occurs. Let C_1 (C_2) be a complex with $\mathit{stype}(C_1) \equiv \odot_1 \cup D$ ($\mathit{stype}(C_2) \equiv \odot_2 \cup D$). Hence, complex C_i is of type τ_i, for $i \in \{1, 2\}$. Moreover, C_1 (C_2) has a negative or circular strand, whence the difference $C_1 - C_2$ is undefined.
   (b) Let s_1 and s_2 in $\mathit{strands}(S_1) \cup \mathit{strands}(S_2)$, having a different length or a different number of $*$-labeled nodes. Denote with $n(s_1)$ resp. $n(s_2)$ the length of s_1 resp. s_2 and denote with $a(s_1)$ resp. $a(s_2)$ the number of $*$-labeled nodes in s_1 resp. s_2. The length of any strand of weak type s_1 resp. s_2 is expressed by $n(s_1)+(\ell-1)a(s_1)$ resp. $n(s_2)+(\ell-1)a(s_2)$ (ℓ is the dimension). For the difference operation to be u-maximal, we must show that any two strands having respective types s_1 and s_2 should be of equal length for all values of ℓ. We distinguish three cases, and prove for each case that all strands having respective types s_1 and s_2 have a different length:
      i. Suppose that $n(s_1) = n(s_2)$ and $a(s_1) \neq a(s_2)$. Now, $n(s_1) + (\ell - 1)a(s_1) = n(s_2) + (\ell - 1)a(s_2)$ implies $(\ell - 1)(a(s_1) - a(s_2)) = 0$ — a contradiction as $\ell - 1$ and $a(s_1) - a(s_2)$ are both nonzero.
      ii. Suppose that $n(s_1) \neq n(s_2)$ and $a(s_1) = a(s_2)$. Now, $n(s_1) + (\ell - 1)a(s_1) = n(s_2) + (\ell - 1)a(s_2)$ implies $n(s_1) = n(s_2)$ — a contradiction.
      iii. $n(s_1) \neq n(s_2)$ and $a(s_1) \neq a(s_2)$: s_1 and s_2 are of equal length if $\ell - 1 = (n(s_2) - n(s_1))/(a(s_1) - a(s_2))$. Without loss of generality we may assume that $a(s_1) > a(s_2)$. If $\ell-1$ has a value strictly larger than $\max\{n(s_2)-n(s_1)\}$, then the above condition cannot be satisfied, i.e., two strands having

respective types s_1 and s_2 will have different lengths.

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

Hybridize Let $\tau = (S, \odot, \mathfrak{h})$. Assume the hybridize operation is undefined on type τ. Hence both $\mathfrak{h} = \mathit{false}$ 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, `hybridize` is not defined for C, and C is of type τ because $\mathfrak{h} = \mathit{false}$.

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

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

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

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

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

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

Cleanup This operation is always defined on the type level.

□

7.4 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 [32].

Theorem 7.5. *For every atomic expression, the DNAQL type relation is tight.*

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

Atomic expression e is of one of the following forms.

Union We have $e = x_1 \cup x_2$. Let $\Gamma(x_1) = \tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\Gamma(x_1) = \tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$. Then $\tau = \tau_1 \cup \tau_2$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.9.

1. Proof of $S \sqsubseteq S''$. Let $D \in \mathit{comp}(S)$. If $D \in \mathit{comp}(S_1)$, let C_1 be such that $\mathit{stype}(C_1) \equiv (\odot_1 \cup D)$, and let C_2 be such that $\mathit{stype}(C_2) \equiv \odot_2$. Complex C_1 has type τ_1 (recall that by the definition of type, C_1 is saturated if $\mathfrak{h}_1 = \mathit{true}$), and complex C_2 has type τ_2. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 \cup C_2 : \tau''$, and so $D \in \mathit{comp}(S'')$.
2. Proof of $\odot'' \sqsubseteq \odot$. We show that if $D \notin \mathit{comp}(\odot)$, then $D \notin \mathit{comp}(\odot'')$. Let $D \in \mathit{comp}(S)$ and $D \notin \mathit{comp}(\odot)$. Let complex C_1 be such that $\mathit{stype}(C_1) \equiv \odot_1$ and let complex C_2 be such that $\mathit{stype}(C_2) \equiv \odot_2$. Complexes C_1 and C_2 are of types τ_1 and τ_2,

respectively. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 \cup C_2 : \tau''$. Because $D \notin \mathit{comp}(\odot)$, $D \notin \mathit{comp}(\odot_1)$ and $D \notin \mathit{comp}(\odot_2)$. Hence complex $C_1 \cup C_2$ does not contain a component of type D. Thus, $D \notin \mathit{comp}(\odot'')$.

3. Proof of $\mathfrak{h}'' = \mathit{true}$ implies that τ is saturated. Assume that τ is not saturated. We show that $\mathfrak{h}'' = \mathit{false}$. Since τ is not saturated, $\mathfrak{h} = \mathit{false}$. Thus, by definition of $\tau = \tau_1 \cup \tau_2$, (a) S_1 and S_2 are mutually interacting, or (b) S_i is not saturated and $\mathfrak{h}_i = \mathit{false}$ for some $i \in \{1, 2\}$.
   (a) Suppose that S_1 and S_2 are mutually interacting, i.e., there is a component D_1 in S_1 with a node u and a component D_2 in S_2 with a node v, such that u and v are free and complementary labeled and D_1 and D_2 are not both immobilized. Let C_1 and C_2 be such that $\mathit{stype}(C_1) \equiv (\odot_1 \cup D_1)$ and $\mathit{stype}(C_2) \equiv (\odot_2 \cup D_2)$, respectively. Thus $C_1 \cup C_2$ is not saturated. Complexes C_1 and C_2 are of types τ_1 and τ_2, respectively. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 \cup C_2 : \tau''$. Since $C_1 \cup C_2$ is not saturated, $\mathfrak{h}'' = \mathit{false}$
   (b) Suppose that S_i is not saturated and $\mathfrak{h}_i = \mathit{false}$ for some $i \in \{1, 2\}$. According to Lemma 6.7, τ_i is not saturated. Hence there is a unsaturated complex C of type τ_i. Without loss of generality we assume $i = 1$. Let C' be a complex of type τ_2. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C$ and $\nu(x_2) = C'$. As $\Gamma \models e : \tau''$, $C \cup C' : \tau''$. Since $C \cup C'$ is not saturated, $\mathfrak{h}'' = \mathit{false}$.

Difference We have $e = x_1 - x_2$. Let $\Gamma(x_1) = \tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$ and $\Gamma(x_2) = \tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$. Then $\tau = \tau_1 - \tau_2$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.9.

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. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 - C_2 : \tau''$. Since $D \in \mathit{comp}(S)$ we have $D \in \mathit{data}(S_1)$, or $D \notin \mathit{data}(S_1)$ and D does not have an isomorphic copy in $\odot_2$. In the first case, D itself appears as a component in $C_1 - C_2$, because C_1 and C_2 have

different ℓ-cores. In the second case, there is, by definition of C_2, a component C in $C_1 - C_2$ with $\mathit{stype}(C) \equiv D$. Since, $C_1 - C_2 : \tau''$, we conclude in both cases that $D \in \mathit{comp}(S'')$.

2. Proof of $\odot'' \sqsubseteq \odot$. Let $D \in \mathit{comp}(S)$ and $D \notin \mathit{comp}(\odot)$. We show that $D \notin \mathit{comp}(\odot'')$. Let complex C_1 be obtained from $\odot_1$ by replacing all $*$-labeled nodes by a^ℓ. Complex C_1 has type τ_1. Let complex C_2 be obtained from $\odot_2 \cup (\odot_1 - T_1)$ by replacing all $*$-labeled nodes by a^ℓ. Recall that T_1 consists of all components in S_1 that have *no* isomorphic copy in S_2. Consequently, $\odot_1 - T_1$ consists of the components in $\odot_1$ having an isomorphic copy in S_2. As a result, C_2 has type τ_2 (note that by the nature of S_1 and S_2 (since $\tau_1 - \tau_2$ is defined), C_2 is always saturated). Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x_1) = C_1$ and $\nu(x_2) = C_2$. As $\Gamma \models e : \tau''$, $C_1 - C_2 : \tau''$. Complex $C_1 - C_2$ consists solely of components of type $\odot_1 \cap T_1$, whence $C_1 - C_2$ does not contain a component of type D, because $D \notin \mathit{comp}(\odot)$ and $D \notin \odot_1 \cap T_1$. Thus, $D \notin \mathit{comp}(\odot'')$.
3. Proof of $\mathfrak{h}'' = \mathit{true}$ implies that τ is saturated. As $\mathfrak{h} = \mathit{true}$, τ is trivially saturated.

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

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

We now assume $\mathfrak{h}' = \mathit{false}$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.9.

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

nodes are labeled with $\overline{b}$, there is a free probe in $\texttt{hybridize}(C)$. Thus, $D \in \mathit{comp}(S'')$.

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

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

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

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

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

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

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

$\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{hybridize}(C) : \tau''$. Thus, $D \notin \mathit{comp}(\odot'')$.

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

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

Ligate We have $e = \texttt{ligate}(x)$. Let $\Gamma(x) = \tau' = (S', \odot', \mathfrak{h}')$. Then $\tau = \texttt{ligate}(\tau')$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.9.

1. Proof of $S \sqsubseteq S''$. Let $D \in \mathit{comp}(S)$. Let E be a component of S' such that $\texttt{ligate}(E) \equiv D$. Let C be a complex such that $\mathit{stype}(C) \equiv \odot' \cup E$. Complex C has type τ'. Hence, we may consider input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. By definition, $\texttt{ligate}(C)$ contains a component of type D. As $\Gamma \models e : \tau''$, $\texttt{ligate}(C) : \tau''$, and so $D \in \mathit{comp}(S'')$.
2. Proof of $\odot'' \sqsubseteq \odot$. Let $D \in \mathit{comp}(S)$ and $D \notin \mathit{comp}(\odot)$. We show that $D \notin \mathit{comp}(\odot'')$. Let E be the set of components of S' such that for every component $F \in E$ we have $\texttt{ligate}(F) \equiv D$. By definition of $\odot$ and $D \notin \mathit{comp}(\odot)$, we know that $\forall F \in E : F \notin \mathit{comp}(\odot')$. Let C be a complex such that $\mathit{stype}(C) \equiv \odot'$. Complex C has type τ'. Hence, we may consider the assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. By construction of C, there is no component of type D in $\texttt{ligate}(C)$. As $\Gamma \models e : \tau''$, $\texttt{ligate}(C) : \tau''$. Thus, $D \notin \mathit{comp}(\odot'')$.
3. Proof of $\mathfrak{h}'' = \mathit{true}$ implies τ is saturated. Assume that $\mathfrak{h} = \mathit{false}$, otherwise the proof is trivial. If $\mathfrak{h} = \mathit{false}$, so is $\mathfrak{h}'$. The fact $\mathfrak{h}'' = \mathit{true}$ implies that τ'' is saturated. Let C be a complex such that $\mathit{stype}(C) \equiv S'$ with all ℓ-cores equal to a^ℓ and all probes labeled $\overline{a}$. Since $\mathfrak{h}' = \mathit{false}$, complex C has type τ'. As $\Gamma \models e : \tau''$, $\texttt{ligate}(C) : \tau''$. Because $\mathfrak{h}'' = \mathit{true}$, $\texttt{ligate}(C)$ must be saturated. The ligate operator only introduces new edges between nodes, in particular, no new nodes are introduced and no closed nodes are made open. Thus, $\texttt{ligate}(C)$ is saturated, implies C is saturated. Hence, S' is saturated, because all probes and ℓ-cores are labeled

complementary. The ligate operator on types also does not introduce new nodes and it does not make closed nodes free. As a result, S is saturated, whence τ is saturated — a contradiction.

Split, Flush Similar to the case of Ligate.

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

Cleanup We have $e = \texttt{cleanup}(x)$. Let $\Gamma(x) = \tau' = (S', \odot', \mathfrak{h}')$. Then $\tau = \texttt{cleanup}(\tau')$. To show $\tau \preceq \tau''$ we verify the three conditions of Proposition 6.9.

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

$\texttt{cleanup}(C)$ does not contain a strand having type s. Complex C has type τ'. Hence, we may consider the input assignment $\nu \in [\![\Gamma]\!]$ with $\nu(x) = C$. As $\Gamma \models e : \tau''$, $\texttt{cleanup}(C) : \tau''$. Thus, $s \notin comp(\odot'')$.

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

□

8

Relational Algebra Simulation

In this section we show that relational algebra expressions can be simulated by DNAQL programs: we show that the simulation is already possible by *well-typed* programs. This illustrates the power of the type checking algorithm.

8.1 Relational Algebra

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

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

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

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

Let us recall the typing rules for the relational algebra [51, 50]. Let $relvars(e)$ be the set of relation variables in a relational algebra expression e. Let $\mathcal{D}$ be a database schema such that $relvars(e) \subseteq dom(\mathcal{D})$, i.e., every relation variable in e is assigned a type. Let R be a relation schema. The rules for when e has type R given $\mathcal{D}$, denoted $\mathcal{D} \vdash e : R$, are the following:

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

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

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

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

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

8.2 Simulation

We want now to represent relations by complexes. We will store data elements as vectors of atomic value symbols. So formally, we use Λ^*, the set of string over Λ, as universe U. Then a tuple t (relation r, instance I) is said to be of dimension ℓ if all data elements appearing in t (r, I) are strings of length ℓ. Let t be a tuple of dimension ℓ over relation schema R. We may assume a fixed order on the attributes of R, say, $A, \ldots, B$. We denote the order by $\oplus$. If the order is clear from the context, it is left implicit. We then represent t by the following ℓ-complex: (using the constant notation of DNAQL)

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

Example 8.1. Let $R = \{A, B, C\}$ be a relation schema with three attributes, and order $A \oplus B \oplus C$. Let $\ell = 3$ and let $\Lambda = \{0, 1\}$. Consider the tuple t over R with $t(A) = 000$, $t(B) = 010$, and $t(C) = 111$. Then

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

□

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

$$\bigcup \{\mathit{complex}(t) \mid t \in r\}$$

which we denote by $\mathit{complex}(r)$. Under order $\oplus$, this complex has type:

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

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

Theorem 8.2. *Let e be an arbitrary well-typed relational algebra expression over database schema $\mathcal{D}$, with output relation schema R, i.e., $\mathcal{D} : e \vdash R$. Then e can be translated into a DNAQL expression e^{DNA}, such that the following holds:*

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

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

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

8.2.1 Abbreviations

For the proof we introduce a few useful abbreviations.

Blockfromto

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

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

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

Connect

A frequently reoccurring pattern in DNAQL programs is adding several complexes together, by means of unions, and then applying hybridize, ligate and cleanup consecutively. We abbreviate the last part of this pattern as *connect*(x) with x a complex variable:

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

Circularization

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

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

Schemas & Types

Let R be a relation schema, then we call S_R, the weak type of τ_R, a *relation-schema-type*. A *pseudo-relation-schema-type* of a relation schema R, resembles the relation-schema-type of R, except that some additional tags outside $\{\#_2, \#_3, \#_4\}$ may be present between attribute-value blocks, furthermore, no additional tags are present at the beginning and end of the strand. More formally, a pseudo-relation-schema-type is of the form $\#_2A_1\#_3 * \#_4S_1\#_2A_2\#_3 * \#_4S_2 \ldots S_{k-1}\#_2A_k\#_3 * \#_4$, where $A_1, \ldots, A_k$ are attributes and $S_1, \ldots, S_{k-1}$ are (possibly empty) sequences of tags, different from $\#_2$, $\#_3$, and $\#_4$. Hence, a relation-schema-type is a special case of a pseudo-relation-schema-type: the sequences of additional nodes are all empty.

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

Lemma 8.3. *Let S be a pseudo-relation-schema-type with k attributes denoted A_1 to A_k. In the following we write A_1 as A and A_k as B. Let S_c be the circularization of S. Let $\tau = (S, \odot, \mathfrak{h})$, and let τ_c be a strong type $(S_c, \odot_c, true)$. Let $\odot_c \equiv$ `empty` if $\odot \equiv$ `empty`, otherwise $\odot_c \equiv S_c$. Let Γ be a type assignment such that $\Gamma(x) = \tau$. Then $\Gamma : circularize(x, A, B) \vdash \tau_c$.*

Thus, $circularize(x, A, B)$ will equal the complex obtained from x by circularizing every strand [40, 4].

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

1. $\Gamma : blockfromto(x, B, A) \vdash \tau_1$. The weak type of τ is a pseudo-relation-schema-type, hence, type τ is saturated regardless of the value of $\mathfrak{h}$. Type $\tau_1 = ((V_1, L_1, \lambda_1, \mu_1, \iota_1, \beta_1), \odot_1, \mathfrak{h}_1)$ resembles type τ, in the sense that it has a strand t_1 that is isomorphic to strand s, except that all nodes from the node labeled A up to and including the node labeled B are blocked, i.e., are members of β_1. The bit $\mathfrak{h}_1$ is set to *true*, and $\odot_1$ is empty. Note that the node labeled $\#_3$ directly following the node labeled B is the only free $\#_3$-labeled node in this type. Also note that there is only one free node labeled $\#_2$ resp. $\#_4$, at the beginning resp. end of the strand.

2. $\Gamma : blockfromto(x, B, A) \cup \texttt{immob}(\overline{\#_3}) \vdash \tau_2$. Type $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ resembles type τ_1. A new node n is introduced, labeled $\overline{\#_3}$. Node n is in $\odot_2$, and is immobilized in both S_2 and $\odot_2$. The strand in S_2, isomorphic to strand t_1, is called t_2. Strand t_2 is non-mandatory and largely blocked (with a single free $\#_3$), whereas probe n is mandatory and free. The $\mathfrak{h}$-bit of τ_2 is set to *false*, because n and t_2 have nodes that can interact.

3. $\Gamma : \texttt{hybridize}(blockfromto(x, B, A) \cup \texttt{immob}(\overline{\#_3})) \vdash \tau_3$. The set of necessary components consists of probe n, thus there are two sets of components on which we will apply $\texttt{hybridize}_t$, i.e., just n and the combination of strand t_2 and n. The first set results in an isomorphic copy of n. The second set results in a new component C_3 consisting of a probe isomorphic to n and an isomorphic copy of strand t_2, connected by a matching between the the free nodes labeled $\#_3$ and $\overline{\#_3}$. As neither of the components occurs both in $\texttt{hybridize}_t(NC)$ and $\texttt{hybridize}_t(S_2)$, neither of the components is mandatory. The $\mathfrak{h}$-bit of τ_3 is set to *true*.

   From this point on, we regard type assignment $\Gamma_2 = \Gamma \cup \{(f_2, \tau_3)\}$.

4. $\Gamma_2 : f_2 \cup \overline{\#_2\#_4} \vdash \tau_4$. Type τ_4 resembles type τ_3, except that a mandatory sticker labeled $\overline{\#_2\#_4}$ is present. Call this sticker u. The $\mathfrak{h}$-bit of τ_4 is set to *false*, as the the sticker and the strand are mutually interacting.

5. $\Gamma_2 : \mathit{connect}(f_2 \cup \overline{\#_2\#_4}) \vdash \tau_5$. The $\mathfrak{h}$-bit of τ_4 is set to *false*, thus hybridization takes place. The sticker u is the only mandatory component. Thus, there are four different sets to apply $\mathit{hybridize}_t$ on:

   (a) $X = NC = \{u\}$: as there is only one sticker, the result of hybridization is isomorphic to u;

   (b) $X = \{u, n\}$: the sticker u and the probe n have no complementary labels, thus the output is isomorphic to the input;

   (c) $X = \{u, C_3\}$: hybridizing the sticker u and the immobilized component C_3 results in two new components. The first component consists of an isomorphic copy of C_3 with a single isomorphic copy of sticker u. The sticker connects the end and beginning of the positive strand in the immobilized component. As a result, the strand is bent into a circle. However, there is still a gap between the beginning and end of the strand. The second component is based on an isomorphic copy of C_3 and two isomorphic copies of the sticker u. One copy of u will match to the beginning of the copy of C_3. The other copy will match to the end of the copy of C_3. This component has maximal matching because the only free nodes are labeled $*, \overline{\#_2}$, and $\overline{\#_4}$. Clearly, these free nodes have no complementary labeled nodes; and

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

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

   (a) Component C_1^h is isomorphic to sticker u;

   (b) Component C_2^h is isomorphic to probe n;

   (c) Component C_3^h is isomorphic to the component formed by one copy of C_3 and one copy of u; and

   (d) Component C_4^h is isomorphic to the component formed by one copy of C_3 and two copies of u.

   The mandatory weak type of type τ_h is equivalent to the empty complex, because none of the components in $\texttt{hybridize}_t(NC)$ is isomorphic to a component in $\texttt{hybridize}_t(S_4)$, where S_4 is the weak type of type τ_4.

The $\mathfrak{h}$-bit of τ_5 is set to *true*, in accordance with the definition of the hybridization operation on complex types.

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

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

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

6. $\Gamma_1 : \texttt{blockfrom}(f_1, A) \vdash \tau_6$. Let us examine both strands of type τ_5 separately. In the linear strand s, there is one σ-blocking range, consisting of the first two nodes, because the second one is labeled with A and the first node is the beginning of the strand. In the circular strand c, there is one σ-blocking range, consisting of all nodes in c. Consequently, type τ_6 consists of two strands, denoted c' and s', which are isomorphic copies of respectively strand c and s, except that all nodes of c' are blocked and the first two nodes of s' are blocked. The $\mathfrak{h}$-bit of type τ_6 is set to *true*, because the $\mathfrak{h}$-bit of type τ_5 is *true*.

7. $\Gamma_1 : \texttt{split}(\texttt{blockfrom}(f_1, A), \#_3) \vdash \tau_7$. The split point identified by $\#_3$ splits only at free $\#_3$. The linear strand s' contains k free nodes labeled $\#_3$, because only the first two nodes are blocked and there are k attributes in s', with a $\#_3$ labeled node following each attribute. The circular strand c' is completely blocked and thus has no free nodes labeled $\#_3$.

   The weak type of type τ_7 thus consists of $k+1$ linear strands, obtained by splitting the linear strand s' at each of the k free $\#_3$-labeled nodes. Note that each of the linear strands consists of a maximum of three nodes plus the length of the longest S_i for $i \in \{1, \ldots, k-1\}$, of which at most one is labeled $*$. Furthermore, the weak type of type τ_7 contains circular strand c'.

   The mandatory weak type of type τ_7 is equivalent to the empty complex type. The $\mathfrak{h}$-bit of type τ_7 is set to *true* because the $\mathfrak{h}$-bit of type τ_6 is *true*.

8. $\Gamma_1 : \mathit{circularize}(x, A, B) \vdash \tau_8$. Let m be a linear strand in the weak type of τ_7. From the previous item, it is known that $n(m) \leq 5 +$

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

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

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

The only strand that qualifies is c'. The cleanup operation, furthermore, removes all matchings, blockings and immobilizations. As a result, the weak type of type τ_8 consists of strand c. The mandatory weak type of type τ_8 is equivalent to the empty complex type. The $\mathfrak{h}$-bit of type τ_8 is set to *true*.

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

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

1. $\Gamma : \mathit{blockfromto}(x, B, A) \vdash \tau_1$. Type $\tau_1 = (S_1, \odot_1, \mathfrak{h}_1)$, with $S_1 = (V_1, L_1, \lambda_1, \mu_1, \iota_1, \beta_1)$, resembles type τ, in the sense that it has a strand t_1 that is isomorphic to strand s, except that all nodes from the node labeled A up to and including the node labeled B are blocked, i.e., are members of β_1. The bit $\mathfrak{h}_1$ is set to *true*, $\odot_1$ is equivalent to weak type S_1. Note that the node labeled $\#_3$ directly following the node labeled B is the only free $\#_3$-labeled node in this type. Also note that there is only one free node labeled $\#_2$ resp. $\#_4$, at the beginning resp. end of the strand.

2. $\Gamma : \mathit{blockfromto}(x, B, A) \cup \texttt{immob}(\overline{\#_3}) \vdash \tau_2$. Type $\tau_2 = (S_2, \odot_2, \mathfrak{h}_2)$ resembles type τ_1. A new node n is introduced, labeled $\overline{\#_3}$. Node n is in $\odot_2$, and in ι of both S_2 and $\odot_2$. The $\mathfrak{h}$-bit of τ_2 is set to *false*. The strand in S_2, isomorphic to strand t_1, is called t_2. Strand t_2 is mandatory and largely blocked (with a single free $\#_3$). Probe n is mandatory and free.

3. $\Gamma : \texttt{hybridize}(\mathit{blockfromto}(x, B, A) \cup \texttt{immob}(\overline{\#_3})) \vdash \tau_3$. The set of necessary components consists of strand s and probe n, thus there is one set of components on which we will apply $\texttt{hybridize}_t$, i.e., strand s and probe n. This results in a new component C_3 consisting of a probe isomorphic to n and an isomorphic copy of strand t_2, connected by a matching between the free nodes labeled $\#_3$ and $\overline{\#_3}$. Component C_3 is mandatory. The $\mathfrak{h}$-bit of τ_3 is set to *true*.

   From this point on, we regard type assignment $\Gamma_2 = \Gamma \cup \{(f_2, \tau_3)\}$.

4. $\Gamma_2 : f_2 \cup \overline{\#_2\#_4} \vdash \tau_4$. Type τ_4 resembles type τ_3, except that a mandatory sticker labeled $\overline{\#_2\#_4}$ is present. Call this sticker u. The $\mathfrak{h}$-bit of τ_4 is set to *false*, as sticker u and the strand of component C_3 are mutually interacting.

5. $\Gamma_2 : \mathit{connect}(f_2 \cup \overline{\#_2\#_4}) \vdash \tau_5$. The $\mathfrak{h}$-bit of τ_4 is set to *false*, thus hybridization takes place. Both component C_3 and the sticker u are mandatory and necessary components. Thus, there is one set to apply $\mathit{hybridize}_t$ on, i.e., $\{u, C_3\}$. Hybridizing the sticker u and the immobilized component C_3 results in two new components. The first component, denoted with C_1^h consists of an isomorphic copy of C_3 with a single isomorphic copy of sticker u. The sticker connects the end and beginning of the positive strand in the immobilized component. As a result, the strand is bent into a circle. However, there is still a gap between the beginning and end of the strand. The second component, denoted with C_2^h, is based on an isomorphic copy of C_3 and two isomorphic copies of the sticker u. One copy of u will match to the beginning of the copy of C_3. The other copy will match to the end of the copy of C_3. This component has maximal matching because the only free nodes are labeled $*, \overline{\#_2}$, and $\overline{\#_4}$. Clearly, these free nodes have no complementary labeled nodes. Both components C_1^h and C_2^h are mandatory.

   Recall that component C_1^h is bent into a circle, but is not circular because it has a gap. The ligate operation fills this gap, creating a new component $C_1^{h'}$ which is circular, i.e., removing the isomorphic copy of the sticker u, would result in a circular strand.

   Finally, the weak type after the cleanup operation consists of the circular strand, denoted c, isomorphic to circularization of strand s.

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

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

6. $\Gamma_1 : \texttt{blockfrom}(f_1, A) \vdash \tau_6$. Let us examine both strands of type τ_5 separately. In the linear strand s, there is one σ-blocking range, consisting of the first two nodes, because the second one is labeled with A and the first node is the beginning of the strand. In the circular strand c, there is one σ-blocking range, consisting of all nodes in c. Consequently, type τ_6 consists of two strands, denoted c' and s', which are isomorphic copies of respectively strand c and s, except that all nodes of c' are blocked and the first two nodes of s' are blocked. The $\mathfrak{h}$-bit of type τ_6 is set to *true*,

because the $\mathfrak{h}$-bit of type τ_5 is *true*.

7. Γ_1 : $\texttt{split}(\texttt{blockfrom}(f_1, A), \#_3) \vdash \tau_7$. The split point identified by $\#_3$ splits only at free $\#_3$. The linear strand s' contains k free nodes labeled $\#_3$, because only the first two nodes are blocked and there are k attributes in s', with a $\#_3$ labeled node following each attribute. The circular strand c' is completely blocked and thus has no free nodes labeled $\#_3$.

   The weak type of type τ_7 thus consists of $k+1$ linear strands, obtained by splitting the linear strand s' at each of the k free $\#_3$-labeled nodes. Note that each of the linear strands consists of a maximum of three nodes plus the length of the longest S_i for $i \in \{1, \ldots, k-1\}$, of which at most one is labeled $*$. Furthermore, the weak type of type τ_7 contains circular strand c'.

   The mandatory weak type of type τ_7 is equivalent the weak type. The $\mathfrak{h}$-bit of type τ_7 is set to *true* because the $\mathfrak{h}$-bit of type τ_6 is *true*.

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

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

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

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

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

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

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

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

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

Inserting into a Circle

Let x be a complex variable and let A and B be attributes. We use the shortcut $\mathit{insertcirc}(x, A, B, s)$ to abbreviate

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

Lemma 8.4. *Let S be the circularization of a pseudo-relation-schema-type with k attributes denoted $A_1, \ldots, A_k$. Let $1 \leq i \leq k$. We denote attribute A_i with A and the attribute following attribute A on the circular strand of S is denoted B. Let S_i be the empty sequence of additional tags between attributes A and B. Let s be a linear strand of length at least two such that no node is labeled with $\#_2$ or $\#_4$. Denote with σ_1, respectively σ_2, the first, respectively last, symbol, of s. We assume that the symbols σ_1 and σ_2 are unique with respect to the circular strand and strand s. Let weak type S' resemble S, except that sequence $S_i = s$.*

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

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

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

2. $\Gamma : \texttt{split}(\mathit{blockfromto}(x, A, B), \#_4) \vdash \tau_{y_1}$. Type τ_{y_1} consists of a single, linear, mandatory strand t_1. Strand t_1 resembles strand t except it is linear. The first attribute of t_2 is B and A is the last attribute. All the nodes, beginning from the node labeled B up to and including the node labeled A, are blocked. The $\mathfrak{h}$-bit is set to *true*.

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

3. $\Gamma_1 : y \cup \texttt{immob}(\overline{\#_3}) \vdash \tau_2$. Type τ_2 consists of two mandatory components. The first component, denoted t_2, is a strand isomorphic to t_1. The

second component consists of a single node n, which is immobilized and is labeled $\overline{\#_3}$. The $\mathfrak{h}_2$-bit is set to *false*, because node n can match with the free node labeled $\#_3$ in strand t_2, following directly behind the node labeled A.

4. $\Gamma_1 : \texttt{hybridize}(y \cup \texttt{immob}(\overline{\#_3})) \vdash \tau_3$. Because $\mathfrak{h}_2 = \mathit{false}$, hybridization takes place. Type τ_2 consists of two mandatory components. One of the mandatory components is a probe, but the probe is not labeled with a negative atomic value symbol, thus both components are necessary components. Hence, $\texttt{hybridize}_t$ is applied on just one set X, consisting of the strand t_2 and the probe n. Strand t_2 has one free node labeled $\#_3$ and probe n is labeled $\overline{\#_3}$. The result is a component C_1 consisting of a strand isomorphic to t_2, a probe isomorphic to n and a matching between the probe and the only free node labeled $\#_3$ in the strand. Component C_1 is mandatory, as all components in X are necessary and mandatory in τ_2. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.

5. $\Gamma_1 : \texttt{hybridize}(y \cup \texttt{immob}(\overline{\#_3})) \cup \overline{\#_4\sigma_1} \cup s \vdash \tau_4$. Type τ_4 consists of three components:

   (a) component C_1;
   (b) sticker s_1, labeled $\overline{\#_4\sigma_1}$; and
   (c) strand s

   All three components are mandatory. The $\mathfrak{h}$-bit of type τ_4 is set to *false*, as the sticker can match with component C_1 and with strand s.

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

7. $\Gamma_1 : \texttt{hybridize}(\texttt{hybridize}(y \cup \texttt{immob}(\overline{\#_3})) \cup \overline{\#_4\sigma_1} \cup s) \cup \overline{\sigma_2\#_2} \vdash \tau_{y_2}$. Type τ_{y_2} consists of the mandatory component C_2 and a mandatory sticker s_2, labeled $\overline{\sigma_2\#_2}$. The $\mathfrak{h}$-bit of type τ_{y_2} is set to *false*, because sticker s_2 can interact with the first node of the linear strand in component C_2, isomorphic to t_2. Moreover, sticker s_2 can interact with the last node of strand isomorphic to s in component C_2.

From this point on, we use the type assignment $\Gamma_2 = \Gamma_1 \cup \{(y_2, \tau_{y_2})\}$

8. $\Gamma_2 : \mathit{connect}(y_2) \vdash \tau_6$. The connect abbreviation consists of (a) a hybridization, (b) a ligation and (c) a cleanup.

   (a) Type τ_{y_2} consists of the immobilized, mandatory component C_2 and mandatory sticker s_2, labeled $\overline{\sigma_2 \#_2}$. Component C_2 consists of an immobilized, largely blocked, linear strand isomorphic to t_2 and the strand s, attached to the former by means of sticker s_1. Component C_2 thus starts with the only free node labeled $\#_2$. Strand s ends with the only free node labeled σ_2. As $\mathfrak{h}_6$ is set to *false*, hybridization takes place. Because both components in τ_{y_2} are mandatory and do not contain a node labeled ?, both are necessary components. Consequently, there is one set X to hybridize, consisting of both components. The hybridization results in two components.

      i. The first component, call it D_1, consists of one isomorphic copy of C_2 and one isomorphic copy of s_2. The sticker binds to the front and end of the immobilized component, bending it into a circle, yet there are still gaps.

      ii. The second component, call it D_2, consists of one isomorphic copy of C_2 and two isomorphic copies of s_2. One of the sticker binds to the front of the isomorphic copy of C_2. The other sticker binds to the end of the isomorphic copy of C_2. More formally, the second sticker binds with the last node of the strand isomorphic to s.

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

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

   Component D_2 also has one gap between the end of the strand isomorphic to strand s and the beginning of the strand isomorphic to t_2. The ligate operation closes this gap, creating a new *linear* strand, denoted t_4, which is the concatenation of strands isomorphic to t_1 and s.

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

Type τ_6 consists of the strands t_3 and t_4, both are mandatory, and $\mathfrak{h}_6$, the $\mathfrak{h}$-bit of type τ_6, is set to *true*.

Strand t_3 equals the insertion of s between the attribute-value blocks of attributes A and B. It remains to get rid of the linear strand t_4.

9. $\Gamma_2 : \texttt{blockfrom}(\mathit{connect}(y_2), B) \vdash \tau_7$. Type τ_6 consists of two strands: t_3 is a circular strand and t_4 is a linear strand. In strand t_3 all nodes will be blocked when starting to block from the node labeled B. In strand t_4, the node labeled B is the second node of the strand. Hence, only two nodes will become blocked in the linear strand. Type τ_7 thus consists of two strands: a circular strand resembling t_3, except that all nodes are blocked, and a linear strand resembling t_4, except that the first two nodes are blocked. Both strands are mandatory and the $\mathfrak{h}$-bit of type τ_7 is set to *true*.

10. $\Gamma_2 : \texttt{split}(\texttt{blockfrom}(\mathit{connect}(y_2), B), \#_3) \vdash \tau_8$. Type τ_7 consists of two mandatory strands: one is circular, the other linear. The circular strand has no free nodes, in particular there is no free node labeled $\#_3$, whence no splitting can be performed on the circular strand. The linear strand, on the other hand, has at least two free nodes labeled $\#_3$ (there is one located directly after the node labeled A and another directly after the node labeled B). The linear strand is thus split in at least three parts. All components are mandatory and the $\mathfrak{h}$-bit of type τ_8 is set to *true*.

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

□

Removing from a Circle

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

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

Lemma 8.5. *Let S be a circularization of a pseudo-relation-schema-type with k attributes, denoted $A_1, \ldots, A_k$. We denote with c the circular strand in S. Let A and B be two attributes in S. Let $c_{A \to B}$ be the substrand of c situated between the end of the attribute-value block of A and the beginning of the attribute-value block of B. Let $c_{A \leftarrow B}$ be the substrand of c starting from the attribute-value block of B up to and including the attribute-value block of A. Let $\tau = (S, S, true)$ be a type. Let S_r be a weak type with a single component isomorphic to $c_{A \leftarrow B}$. Let $\tau_r = (S_r, S_r, true)$ be a type. Let Γ be a type assignment such that $\Gamma(x) = \tau$. If $n(c_{A \to B}) < n(c_{A \leftarrow B})$ and $a(c_{A \to B}) < a(c_{A \leftarrow B})$, then $\Gamma :$ removeBetweenCirc$(x, A, B) \vdash \tau_r$.*

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

1. $\Gamma :$ *blockfromto*$(x, A, B) \vdash \tau_1$. Type τ is saturated. Type τ_1 consists of a strand t_1 that is isomorphic to t, except that all nodes that are *not* between attribute A and B are blocked. As a result, at least one $\#_4$-labeled node is free (the last one of the attribute-value block of attribute A) and at least one $\#_2$-labeled node is free (the first one of the attribute-value block of attribute B). Strand t_1 is mandatory and the $\mathfrak{h}$-bit of type τ_1 is set to *true*.

2. $\Gamma : \texttt{split}(\texttt{split}(\textit{blockfromto}(x, A, B), \#_4), \#_2) \vdash \tau_2$. Circular strand t_1 is cut at least behind the attribute-value block of attribute A and before the attribute-value block of attribute B. Type τ_2 contains at least two strands. The first strand, t_3 is isomorphic to $c_{A \leftarrow B}$, except all nodes are blocked except for the first node (labeled $\#_2$, just in front of the node labeled B) and the last three nodes (labeled $\#_3$, $*$ and $\#_4$, just after the node labeled A). The other strands, resulting from the double split, bundled in a set called T, are at most as long as $c_{A \to B}$. All strands are mandatory. The $\mathfrak{h}$-bit of type τ_2 is set to *true*.

3. $\Gamma : \texttt{cleanup}(\texttt{split}(\texttt{split}(\textit{blockfromto}(x, A, B), \#_4), \#_2)) \vdash \tau_r$. Next, we show that strand $c_{A \leftarrow B}$ is longer than any strand in T, for any value of ℓ. Let u be a strand in set T, then the length of u is at most equal to the length of $c_{A \to B}$, which is $n(c_{A \to B}) + (\ell - 1)a(c_{A \to B})$. The length of $c_{A \leftarrow B}$ is $n(c_{A \leftarrow B}) + (\ell - 1)a(c_{A \leftarrow B})$. We need to show that $n(c_{A \to B}) + (\ell -$

$1)a(c_{A\to B}) < n(c_{A\leftarrow B}) + (\ell-1)a(c_{A\leftarrow B})$, or, $0 < n(c_{A\leftarrow B}) - n(c_{A\to B}) + (\ell-1)(a(c_{A\leftarrow B}) - a(c_{A\to B}))$. This is true, because $a(c_{A\to B}) < a(c_{A\leftarrow B})$ and $n(c_{A\to B}) < n(c_{A\leftarrow B})$. As a result, strand $c_{A\leftarrow B}$ is the only strand in the output type and the strand also qualifies for mandatory. The $\mathfrak{h}$-bit of the output type is set to *true*.

□

Block Selecting

Let x be a complex variable. Let A and B be attributes. Let a be an atomic value symbol and let i be a counter variable. We use $\mathit{blockselect}(x, A, B, a, i)$ to abbreviate

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

Lemma 8.6. *Let S be a circularization of a relation-schema type with k attributes, denoted $A_1, \ldots, A_k$. Let A and B be two consecutive attributes on the circular strand of S. Let $\tau = (S, S, \mathit{true})$ be a type. Let $\tau_s = (S, \texttt{empty}, \mathit{true})$ be a type. Let Γ be a type assignment such that $\Gamma(x) = \tau$. Then*

$$\Gamma : \mathit{blockselect}(x, A, B, a, i) \vdash \tau_s \ .$$

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

1. $\Gamma : \mathit{blockfromto}(x, A, B) \vdash \tau_1$. Type τ is saturated. Type τ_1 contains a single, circular, mandatory strand t isomorphic to c, except that the nodes between the two nodes labeled A and B are the only free nodes. In particular, the only node labeled $*$ is the ℓ-core of attribute A. The $\mathfrak{h}$-bit of type τ_1 is set to *true*.

2. $\Gamma : \texttt{blockexcept}(\mathit{blockfromto}(x, A, B), i) \vdash \tau_2$. Type τ_2 consists of a single, circular, mandatory strand u, isomorphic to strand t, except that the only node labeled $*$ in t is relabeled to $\hat{*}$ in u and the nodes labeled $\#_3$ and $\#_4$ (respectively before and after the ℓ-core of attribute A) are blocked. The $\mathfrak{h}$-bit of type τ_2 is set to *true*.

3. $\Gamma : \texttt{blockexcept}(\mathit{blockfromto}(x, A, B), i) \cup \texttt{immob}(\overline{a}) \vdash \tau_y$. Type τ_y consists of two mandatory complexes: strand u and a probe n labeled ?. As probe n and the $\hat{*}$ labeled node of strand u are free and can match, the $\mathfrak{h}$-bit of type τ_y is set to *false*.

   From this point on, we use the type assignment $\Gamma' = \Gamma \cup \{(y, \tau_y)\}$.

4. $\Gamma' : \texttt{hybridize}(y) \vdash \tau_3$. Type τ_y consists of two mandatory components, however, probe n is labeled with ?, hence the probe is not a necessary component. As a result, there are two sets X to consider in the hybridization process. The first set X consists solely of the strand u. Strand u is a positive strand, thus no matchings can be added. The second set X consists of strand u and probe n. The hybridization of X results in a single, immobilized component called C_1, built up from a strand isomorphic to u and a probe isomorphic to n, with a matching between the only $\hat{*}$-labeled node of the strand and the probe. Because neither u or C_1 is present in both invocations of $\texttt{hybridize}_t$, both are non-mandatory components. The $\mathfrak{h}$-bit of type τ_y is set to *true*.

5. $\Gamma' : \texttt{flush}(\texttt{hybridize}(y)) \vdash \tau_4$. Component C_1 is the only immobilized component in type τ_3. Type τ_4 thus consists solely of component C_1. The component is non-mandatory. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.

6. $\Gamma' : \texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(y))) \vdash \tau_s$. The cleanup operation removes blockings, matchings and probes. What is left, is a strand isomorphic to strand c. The strand is non-mandatory. The $\mathfrak{h}$-bit of the output type is set to *true*.

□

8.2.2 Relational Algebra Expressions

The proof of Theorem 8.2 now goes by induction on the structure of e. By induction, we know that subexpressions e_1 and e_2 of expression e are simulated by the respective well-typed (under type assignment $\Gamma_{\mathcal{D}}$) DNAQL expressions e_1^{DNA} and e_2^{DNA}. Subexpression e_1 (e_2) is over relation schema R (S). Type τ_R (τ_S) is a relation-schema-type of relation schema R (S). Type τ'_R (τ'_S) has a weak type isomorphic to τ_R (τ_S), but in contrast to τ_R (τ_S), which has the empty complex as mandatory weak type, the mandatory weak type of τ'_R (τ'_S) is isomorphic to the weak type of τ'_R (τ'_S).

Union

Lemma 8.7. *Let $e = e_1 \cup e_2$, with $\mathcal{D} : e \vdash R$. If expression e^{DNA} is defined as $e_1^{DNA} \cup e_2^{DNA}$, then $\Gamma_{\mathcal{D}} : e^{DNA} \vdash \tau_R$.*

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

Difference.

Lemma 8.8. *Let $e = e_1 - e_2$, with $\mathcal{D} : e \vdash R$. If expression e^{DNA} is defined as $e_1^{DNA} - e_2^{DNA}$, then $\Gamma_{\mathcal{D}} : e_1^{DNA} - e_2^{DNA} \vdash \tau_R$.*

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

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

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

□

Cartesian product.

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

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

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

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

Lemma 8.9. *Let $e = e_1 \times e_2$, with $\mathcal{D} : e \vdash T$, where $T = R \cup S$ and $R \cap S = \emptyset$. Let τ_R (τ_S) be the relation-schema-type of relation schema R (S) with k attributes, denoted $A_1, \ldots, A_k$ ($B_1, \ldots, B_m$). Let $A = A_1$ ($C = B_1$) be the first and $B = A_k$ ($D = B_m$) be the last attribute of relation schema R under order $\oplus_R$. Let $\oplus$ be the combined order of $\oplus_R$ and $\oplus_S$. Let $\tau_T^{\oplus}$ be a relation-*

schema-type with order $\oplus$*. If expression* e^{DNA} *is defined as*

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

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

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

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

1. We extend and augment the type assignment $\Gamma_{\mathcal{D}}$ with $\{(x, \tau_R'); (y, \tau_S')\}$. Type checking the first two let-statements, adds the types for complex variables x and y. The main body of the expression resides inside the else-part of two if-statements. Because τ_R and τ_S both consist of a single, non-mandatory strand, the main body of the expression is type checked with the augmented types τ_R' and τ_S' for x respectively y. We denote the extended and augmented type assignment by Γ'.

2. $\Gamma' : \overline{\#_4\#_5} \cup \#_5 \vdash \tau_0$. Type τ_0 consists of a single-node strand t_0 labeled $\#_5$ and a sticker s_0 labeled $\overline{\#_4\#_5}$. Both components are mandatory. The $\mathfrak{h}$-bit of type τ_0 is set to *false*, because the node of t_0 can match with the node labeled $\overline{\#_5}$ of sticker s_0.

3. $\Gamma' : \mathit{hybridize}(\overline{\#_4\#_5} \cup \#_5) \vdash \tau_1$. Type τ_0 consists of two mandatory components and none of the nodes is labeled ?, thus both components are necessary. Type τ_1 consists of a single, mandatory component, formed by binding strand t_0 on sticker s_0. The $\mathfrak{h}$-bit is set to *true*.

   From this point on, we use type assignment $\Gamma_1 = \Gamma' \cup \{(r, \tau_1)\}$.

4. $\Gamma_1 : \overline{\#_1\#_2} \cup \#_1 \vdash \tau_2'$. Type τ_2' consists of a single-node strand t_2 labeled $\#_1$ and a sticker s_2 labeled $\overline{\#_1\#_2}$. Both components are mandatory. The $\mathfrak{h}$-bit of type τ_2' is set to *false*, because the node of t_2 can match with the node labeled $\overline{\#_1}$ of sticker s_2.

5. $\Gamma_1 : \texttt{hybridize}(\overline{\#_1\#_2} \cup \#_1) \vdash \tau_2$. Type τ_2' consists of two mandatory components and none of the nodes is labeled ?, thus both components are necessary. Type τ_2 consists of a single, mandatory component, formed by binding strand s_2 on sticker s_2. The $\mathfrak{h}$-bit is set to *true*.

   From this point on, we use the type assignment $\Gamma_2 = \Gamma_1 \cup \{(l, \tau_2)\}$.

6. $\Gamma_2 : \mathit{connect}(x \cup r) \vdash \tau_3$. Firstly, τ_R' consists of a single component, denoted t_R. Type τ_1 consists of a single component, denoted C_R. Both components are mandatory within their respective types. Strand t_R has at least one free node labeled $\#_4$ and component C_R has one free node labeled $\overline{\#_4}$. As a result, the union results in a type called τ' with components t_R and C_R that are both mandatory. The $\mathfrak{h}$-bit is set to *false*.

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

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

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

   (c) There are essentially two strands in the C_R^3, the concatenation of t_R and t_0, called t_R', and t_0. The length of t_R' is $5k + 1 + (\ell - 1)k$. The length of isomorphic copies of t_0 is 1.

   Type τ_3 thus consists of strand t_R'. The strand is mandatory. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.

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

7. $\Gamma_3 : \mathit{connect}(y \cup l) \vdash \tau_4$. An analogous reasoning to the above point reveals that type τ_4 consists of a single strand, called t_S', which is a

concatenation of strand t_2 and t_S, i.e., t'_S is equivalent to t_S except that a node labeled $\#_1$ is attached to the front of it.

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

8. $\Gamma_4 : \mathit{connect}(e_2^a \cup e_2^b \cup \overline{\#_5\#_1}) \vdash \tau_5$. Firstly, we combine the mandatory strands t'_R and t'_S with sticker s_3, labeled $\overline{\#_5\#_1}$. The $\mathfrak{h}$-bit of this combination is set to *false*, because strand t'_R (t'_S) contains a free node labeled $\#_5$ ($\#_1$) which can match with sticker s_3.

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

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

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

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

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

9. $\Gamma_5 : \mathit{circularize}(e_2, A, D) \vdash \tau_6$. Type τ_5 consists of strand t_{RS}, which is a pseudo-relation-schema-type at this point, because of the two nodes labeled $\#_1$ and $\#_5$ between attributes B and C. Moreover, strand t_{RS} is also mandatory in τ_5. The $\mathfrak{h}$-bit of type τ_5 is set to *true*. The first (last) attribute of strand t_{RS} is A (D). Hence, by Lemma 8.3 we know that type τ_6 consists of the circular version of strand t_{RS}, called c_{RS}. Moreover, c_{RS} is mandatory in type τ_6 and the $\mathfrak{h}$-bit of type τ_6 is set to *true*.

   From this point on, we use the type assignment $\Gamma_6 = \Gamma_5 \cup \{(e_1, \tau_6)\}$.

10. $\Gamma_6 : \mathit{blockfromto}(e_1, B, C) \vdash \tau_7$. Type τ_7 consists of an isomorphic copy of c_{RS}, except that the nodes starting from the node labeled $\#_3$ directly

following the B-labeled node, up to but not including the C-labeled node are the only free nodes. All other nodes are blocked. The strand is mandatory and the $\mathfrak{h}$-bit of type τ_7 is set to *true*.

11. $\Gamma_6 : \texttt{cleanup}(\texttt{split}(\texttt{split}(\mathit{blockfromto}(e_1, B, C), \#_2), \#_4)) \vdash \tau_8$. In the strand of type τ_7 there is only one free $\#_2$-labeled node and only one free $\#_4$-labeled node, namely, the first node of the attribute-value block of C and the last node of the attribute-value block of B. Between these nodes, there are two nodes, labeled $\#_5$ and $\#_1$. The two split operations cut these two nodes from the strand, forming a new short strand called t_{15}. As a result, the circular strand becomes linear and attribute C is the first attribute. Call this strand t_{SR}. Note that strand t_{SR} is the concatenation of strands t_S and t_R.

    The length of strand t_{SR} equals $(4 + \ell)(k + m)$, whereas the length of strand t_{15} equals 2. Hence, the cleanup operation removes the blockings from strand t_{SR} and disposes of strand t_{15}. Strand t_{SR} is mandatory in type τ_8 and the $\mathfrak{h}$-bit is set to *true*.

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

    By the previous steps, we know that the else-part of the if-statement has type τ_8. By definition we know that $\Gamma'' : \texttt{empty} \vdash (\texttt{empty}, \texttt{empty}, \mathit{true})$. Hence, combining both types results in $\tau_T^{\oplus}$ as strand t_{SR} in type τ_8 becomes non-mandatory.

13. $\Gamma' : \texttt{if empty}(x) \texttt{ then empty else if empty}(y) \texttt{ then empty else } e_y \vdash \tau_T^{\oplus}$. The else-part of the if-test has type $\tau_T^{\oplus}$. The then-part of the if-test has the empty type. Hence, it does not add components to nor does it make components non-mandatory, thus the output type is $\tau_T^{\oplus}$.

□

Reordering is performed by repeated shuffling of attribute-value pairs. Shuffling attribute-value pairs in a tuple is done using a new technique we call *double bridging*. Instead of using a single sticky bridge, two sticky bridges

are hybridized onto one strand. A careful placement of the bridges allows us to cut twice in the strand whilst keeping the strand connected. Moreover, the two bridges guide the strand into its new conformation.

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

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

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

Lemma 8.10. *Let R be a relation schema with order $\oplus$ on the attributes. Let C be an attribute in R, and let $\otimes$ be the order derived from $\oplus$ by making attribute C the last attribute, i.e., for any attribute $X \in R$, if $X \neq C$ then $X \otimes C$ and for any pair of attributes (X, Y) in R, with $X \neq C$ and $Y \neq C$, $X \oplus Y \Leftrightarrow X \otimes Y$. Let $\tau_R^{\oplus}$ be a relation-schema-type for relation schema R with order $\oplus$. Let A be the first attribute, B be the attribute just in front of attribute C, D be the attribute appearing just after C and E be the last attribute of R under order $\oplus$. Let x be a complex variable. Let e^{DNA} be the following DNAQL expression:*

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

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

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

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

1. $\Gamma : \mathit{circularize}(x, A, E) \vdash \tau_1$. Strand t_R in type $\tau_R^{\oplus}$ is a positive, linear, and mandatory strand. It does not contain matchings, blockings, or immobilizations. The first attribute occurring on strand t_R is A and the last attribute is E. Hence, by Lemma 8.3 we know that type τ_1 consists of the circular version of strand t_R, called c_R. Strand c_R is mandatory in type τ_1 and the $\mathfrak{h}$-bit is set to *true*.

2. $\Gamma : \mathit{insertcirc}(\mathit{circularize}(x, A, E), B, C, \#_6\#_7) \vdash \tau_2$. Strand c_R is circular and it is mandatory in type τ_1. As strand c_R is derived from strand t_R and t_R is a relation-schema-type, there are no nodes between the attribute-value block of B and C. The labels $\#_6$ and $\#_7$ are unique with respect to the strand $\#_6\#_7$ and the c_R. Hence, by Lemma 8.4 we know that type τ_2 consist of a mandatory, circular strand, isomorphic to c_R except that two nodes, labeled $\#_6$ resp. $\#_7$, are added between the attribute-value blocks of attributes B and C. Call this strand c_R^1. The $\mathfrak{h}$-bit of type τ_2 is set to *true*.

   From this point on, we use the type assignment $\Gamma_{f_1} = \Gamma \cup \{(f_1, \tau_2)\}$.

3. $\Gamma_{f_1} : \mathit{insertcirc}(f_1, C, D, \#_8\#_9) \vdash \tau_3$. Strand c_R^1 is circular and it is mandatory in type τ_2. As strand c_R is derived from strand t_R and t_R is a relation-schema-type, there are no nodes between the attribute-value block of C and D. The labels $\#_8$ and $\#_9$ are unique with respect to the strand $\#_8\#_9$ and strand c_R^1. Hence, by Lemma 8.4, we know that type τ_3 consists of a mandatory, circular strand, isomorphic to c_R^1 except that two nodes, labeled $\#_8$ resp. $\#_9$, are inserted between the attribute-value blocks of attributes C and D. Call this strand c_R^2. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.

   From this point on, we use the type assignment $\Gamma_{f_2} = \Gamma_{f_1} \cup \{(f_2, \tau_3)\}$.

4. $\Gamma_{f_2} : \mathtt{split}(\mathit{blockfromto}(f_2, E, A), \#_4) \vdash \tau_4$. Type τ_4 consists of a strand, called c_R^3, isomorphic to strand c_R^2, except that all the nodes from the first attribute to the last attribute are blocked. Hence, the only free node labeled $\#_4$ is the last node of the attribute-value block of E. The split operation results in a linear version of c_R^3 in which A as the first attribute. Call this strand t_R^4. Strand t_R^4 is mandatory in type τ_4. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.

   From this point on, we use the type assignment $\Gamma_{f_3} = \Gamma_{f_2} \cup \{(f_3, \tau_4)\}$.

5. Γ_{f_3} : $\mathit{blockfromto}(f_3, E, A) \cup \#_9\#_1 \cup \overline{\#_4\#_9} \vdash \tau_5$. Type τ_5 consists of strand t_R^4, a strand t_1 labeled with $\#_9\#_1$, and a sticker s_1 labeled $\overline{\#_4\#_9}$. All components are mandatory. The $\mathfrak{h}$-bit of type τ_5 is set to *false*, because strand t_R^4 has a free $\#_4$-labeled node, strand t_1 has a free $\#_9$-labeled node and the sticker has a free node labeled $\overline{\#_4}$ and a free node labeled $\overline{\#_9}$.

6. Γ_{f_3} : $\mathit{connect}(\mathit{blockfromto}(f_3, E, A) \cup \#_9\#_1 \cup \overline{\#_4\#_9}) \vdash \tau_6$. The connect abbreviation consists of (a) a hybridization, (b) a ligation and (c) a cleanup.

   (a) All components are mandatory and no node is labeled with ?. Hence, all components are necessary. Strands t_R^4 and t_1 can be connected by means of sticker s_1. Thus, hybridization forms a new component C_1 consisting of one isomorphic copy of strand t_R^4, one isomorphic copy of t_1 and one isomorphic copy of s_1. There is a gap between the last node of the strand isomorphic to t_R^4 and the first node of the strand isomorphic to t_1. Component C_1 is mandatory. The $\mathfrak{h}$-bit is set to *true*.

   (b) The ligate operation will fill the gap between the two positive strands in component C_1. Let component C_1' be the result of the ligate operation on component C_1. Then there is a single positive strand in component C_1', namely the concatenation of strands isomorphic to t_R^4 respectively t_1.

   (c) There is only one positive strand in component C_1'. Hence, type τ_6 consists of a single strand (with blockings), call it t_R^5. Strand t_R^5 is mandatory. The $\mathfrak{h}$-bit of type τ_6 is set to *true*.

   From this point on, we use the type assignment $\Gamma_{f_4} = \Gamma_{f_3} \cup \{(f_4, \tau_6)\}$.

7. Γ_{f_4} : $\texttt{hybridize}(f_4 \cup \texttt{immob}(\overline{A})) \vdash \tau_7$. Firstly, a probe n labeled $\overline{A}$ and strand t_R^5 are combined. Both components are mandatory. The $\mathfrak{h}$-bit is set to *false*, because strand t_R^5 and probe n can match.

   Because both components are mandatory and no node is labeled with ?, there is only one set X to hybridize. Hybridization forms a new component C_2 consisting of one isomorphic copy of t_R^5 and one isomorphic copy of n. Node n and the node labeled A in the isomorphic copy of strand t_R^5 are matched. Component C_2 is mandatory and immobilized. The $\mathfrak{h}$-bit of type τ_7 is set to *true*.

8. Γ_{f_4} : $\texttt{hybridize}(f_4 \cup \texttt{immob}(\overline{A})) \cup \overline{\#_6\#_8} \cup \overline{\#_1\#_7} \vdash \tau_8$. Type τ_8 consists of component C_2, a sticker b_1 labeled $\overline{\#_6\#_8}$ and a sticker b_2 labeled $\overline{\#_1\#_7}$.

Stickers b_1 and b_2 are called "bridges". All components are mandatory. The $\mathfrak{h}$-bit of type τ_8 is set to *false*.

9. $\Gamma_{f_4} : \texttt{hybridize}(\texttt{hybridize}(f_4 \cup \texttt{immob}(\overline{A})) \cup \overline{\#_6\#_8} \cup \overline{\#_1\#_7}) \vdash \tau_9$. Because the $\mathfrak{h}$-bit of type τ_8 is *false*, hybridization takes place. All components are mandatory and there is no ?-labeled probe. Hence, all components are necessary components. Component C_2 is immobilized and has one free node labeled $\#_6$, one free node labeled $\#_8$, one free node labeled $\#_7$, and one free node labeled $\#_1$. Each node of the bridges b_1 and b_2 can thus match to exactly one node in one copy of component C_2.

   Hybridization is performed on components C_2, b_1, and b_2. Four new components are formed.

   (a) The first component, call it C_b^1 consists of one isomorphic copy of C_2, one isomorphic copy of b_1, and one isomorphic copy of b_2.

   (b) The second component, call it C_b^2, consists of one isomorphic copy of C_2, one isomorphic copy of b_1, and two isomorphic copies of b_2 (one binding with the free $\#_1$-labeled node in C_2, the other binding with the free $\#_7$-labeled node in C_2). In this case, only bridge b_1 is successfully placed.

   (c) The third component, call it C_b^3, consists of one isomorphic copy of C_2, two isomorphic copies of b_1 (one binding with the free $\#_6$-labeled node in C_2, the other binding with the free $\#_8$-labeled node in C_2), and one isomorphic copy of b_2. In this case, only bridge b_2 is successfully placed.

   (d) The fourth component, call it C_b^4, consists of one isomorphic copy of C_2, two isomorphic copies of b_1, and two isomorphic copies of b_2. Each bridge matches with one free node. In this case, no bridge is placed successfully.

   Each component C_b^i, for $1 \leq i \leq 4$, is mandatory. The $\mathfrak{h}$-bit of type τ_9 is set to *true*.

   From this point on, we use the type assignment $\Gamma_{f_5} = \Gamma_{f_4} \cup \{(f_5, \tau_9)\}$.

   Type τ_9 is depicted in Figure 8.1.

10. $\Gamma_{f_5} : \texttt{split}(\texttt{split}(f_5, \#_6), \#_8) \vdash \tau_{10}$. Type τ_9 consists of four mandatory, immobilized components. Each component has one closed node labeled $\#_6$ and one closed node labeled $\#_8$.

    (a) Component C_b^1 has two successfully placed bridges. Cutting at the nodes labeled $\#_6$ and $\#_8$ results in a component C_c^1 in which

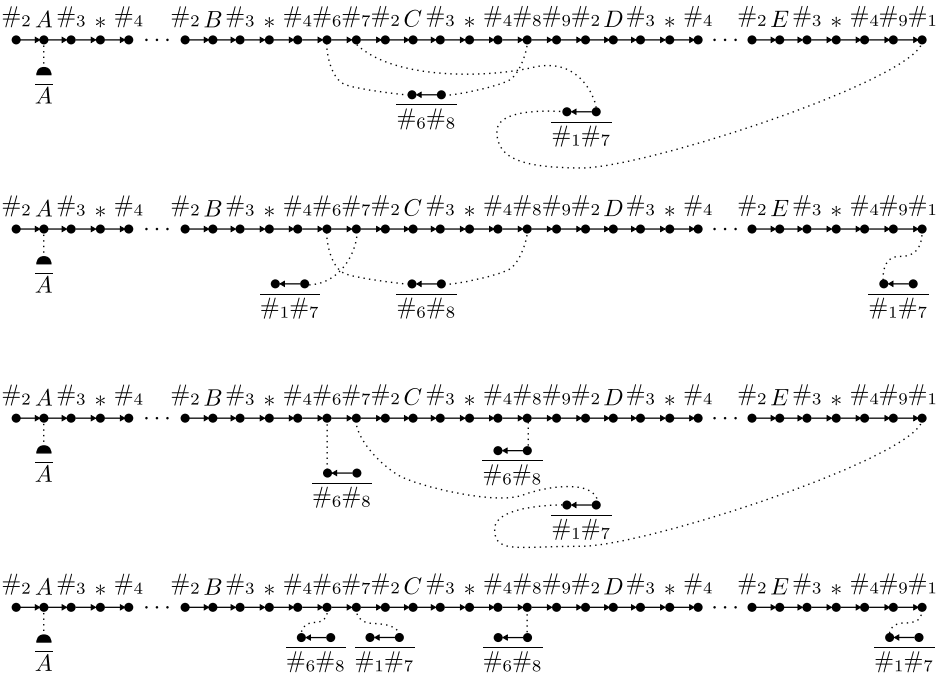


Figure 8.1: Type τ_9: two bridges attached to the data strand.

the attribute-value block of C is pulled to position just after the attribute-value block of E. The attribute-value block of D is pulled against the attribute-value block of B. This situation is depicted in Figure 8.2. The component consists of three positive strand with gaps between the first and second substrand and the second and third substrand. Note that some additional nodes are still present between some attribute-value blocks.

(b) In component C_b^2, two copies of b_2 are present. As a result, two new components C_c^2 and C_c^3 are created by the split operations. Component C_c^2 consists of two substrands held together by a copy of bridge b_1. A copy of bridge b_2 is also present in this component. The attribute-value block C has been cut from this component. Component C_c^3 consists of a strand labeled with $\#_7$ and the attribute-value block of C and a copy of bridge b_2.

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

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

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

11. $\Gamma_{f_5} : \texttt{cleanup}(\texttt{ligate}(\texttt{split}(\texttt{split}(f_5, \#_6), \#_8))) \vdash \tau_{11}$. There are four gaps in the components of type τ_{10}. In component C_c^1 the ligate operation glues the three linear strand together in their new conformation, creating a new component called D_1. Component D_1 is depicted in

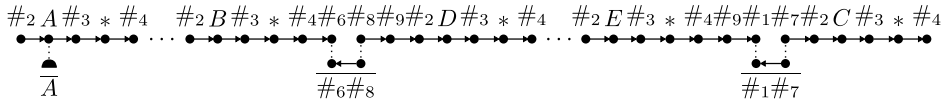


Figure 8.2: Type τ_{10}: the two bridges have guided the strand into its new conformation. Note there are still gaps in the data strand.

Figure 8.2. Component D_1 contains a one positive strand with the all attribute-value blocks. Call thus strand t_2. In component C_c^2 the ligate operation glues the two positive strands together, creating a new component called D_2. Component D_2 contains the attribute-value blocks of all attributes except attribute C. In component C_c^5 the ligate operation glues the two linear strands together, creating a new component called D_3. Component D_3 contains the attribute-value blocks of attributes D to E and C.

The cleanup operation is applied to eight mandatory components: D_1, D_2, C_c^3, C_c^4, D_3, C_c^5, C_c^6, C_c^7, and C_c^8. All of these components contain a single positive strand and one or more sticker. The lengths of the strands in the components are:

(a) D_1: $|R|(4+\ell)+6$
(b) D_2: $(|R|-1)(4+\ell)+5$
(c) C_c^3: $5+\ell$
(d) C_c^4: $|\{A,\ldots,B\}|(4+\ell)+1$
(e) D_3: $|\{D,\ldots,E,C\}|(4+\ell)+5$
(f) C_c^6: $|\{A,\ldots,B\}|(4+\ell)+1$
(g) C_c^7: $5+\ell$
(h) C_c^8: $|\{D,\ldots,E\}|(4+\ell)+4$

Clearly, strand t_2 in component D_1 is always the longest strand. Moreover, strand t_2 qualifies for mandatory. The $\mathfrak{h}$-bit of type τ_{11} is set to *true*.

From this point on, we use the type assignment $\Gamma_{f_6} = \Gamma_{f_5} \cup \{(f_6, \tau_{11})\}$.

12. $\Gamma_{f_6} : \mathit{removeBetweenCirc}(\mathit{circularize}(f_6, A, C), B, D) \vdash \tau_{12}$. Strand t_2 is a pseudo-relation-schema type that is mandatory in type τ_{11}, moreover it is the only component of type τ_{11}. Hence, by Lemma 8.3, we know that the output type of $\mathit{circularize}(f_6, A, C)$ contains the circular version of t_2. Call this strand c_2. Strand c_2 is mandatory, it is a pseudo-relation-schema type and between the attribute-value blocks of attributes B and

D there are only three nodes. Hence, by Lemma 8.5, we know that type τ_{12} consists of a linear strand t_3 starting with attribute D and ending with attribute B. Strand t_3 is mandatory in type τ_{12}. The $\mathfrak{h}$-bit of type τ_{12} is set to *true*.

From this point on, we use the type assignment $\Gamma_{f_7} = \Gamma_{f_6} \cup \{(f_6, \tau_{12})\}$.

13. $\Gamma_{f_7} : \mathit{removeBetweenCirc}(\mathit{circularize}(f_7, D, B), E, C) \vdash \tau_{13}$. In a reasoning similar to the previous item, we derive that τ_{13} consists of a strand, call it t_4, that is isomorphic to strand t_3, except that the three nodes labeled $\#_9\#_1\#_7$ between the attribute-value blocks of attributes E and C have been removed. Strand t_4 starts with attribute C and ends with attribute E. It is mandatory in type τ_{14}, and the $\mathfrak{h}$-bit of τ_{13} is set to *true*.

    From this point on, we use the type assignment $\Gamma_{f_8} = \Gamma_{f_7} \cup \{(f_8, \tau_{13})\}$.

14. $\Gamma_{f_8} : \mathit{removeBetweenCirc}(\mathit{circularize}(f_8, C, E), C, A) \vdash \tau_R^{\otimes}$. Strand t_4 in type τ_{13} is mandatory and a relation-schema-type. Hence, by Lemma 8.3, we know that the output type of $\mathit{circularize}(f_8, C, E)$ consists of a circular version of strand t_4. Call this strand c_4. Strand c_4 is mandatory and circular. There are no nodes between the attribute-value blocks of attributes C and A. Hence, by Lemma 8.5, we know that the output type consists of a linear strand, starting with attribute A and ending with attribute C that is a relation-schema type. Hence, the output type is $\tau_R^{\otimes}$.

□

Projection

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

Three or More Attributes.

Lemma 8.11. *Let $e = \widehat{\pi}_C(e_1)$, with $\mathcal{D} : e_1 \vdash R$, $C \in R$ and $\mathcal{D} : e \vdash S$, where $S = R \backslash \{C\}$. Let τ_R be a relation-schema-type of relation schema R with $k \geq 3$ attributes, and let $S = \{A_1, \ldots A_{i-1}, A_{i+1}, \ldots, A_k\}$ be a relation-schema-type with $k-1$ attributes. Denote attribute A_i with C, let $A = A_1$, and let $B = A_k$. Consider R to be circular, then let X be the attribute just in front of C and Y be the attribute directly following C. Let A' be the first attribute of S and*

let B' be the last attribute of S. If expression e^{DNA} is defined as

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

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

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

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

2. $\Gamma' : \mathit{circularize}(x, A, B) \vdash \tau_1$. Strand t_R is linear and without blockings, matchings and immobilizations. Furthermore t_R is mandatory in type τ'_R. The first attribute of t_R is A and the last attribute is B. By Lemma 8.3, we know that type τ_1 consists of a circular version of t_R. Denote this circular strand c_R. Strand c_R is mandatory. The $\mathfrak{h}$-bit of type τ_1 is set to *true*.

3. $\Gamma' : \mathit{blockfromto}(\mathit{circularize}(x, A, B), X, Y) \vdash \tau_2$. Strand c_R of type τ_1 is circular, and attribute X is just in front of attribute C and attribute Y is just behind attribute C. The blockfromto abbreviation creates a new circular strand, call it c^2_R, in which all nodes, except those in the substrand between attribute X and Y, are blocked. Consequently, only two nodes labeled $\#_2$ are free. The first node of the attribute-value block of C and the first node of the attribute-value block of attribute Y, i.e., the first node after the attribute-value block of C. Strand C^2_r is mandatory in type τ_2 and the $\mathfrak{h}$-bit of type τ_2 is set to *true*.

4. $\Gamma' : \texttt{split}(\mathit{blockfromto}(\mathit{circularize}(x, A, B), X, Y), \#_2) \vdash \tau_3$. The split operation introduces two new strands. The first one, called t'_S, contains the attribute-value blocks of relation schema S. The first attribute of t'_S is Y, the last attribute is X. The nodes from attribute Y up to and including attribute X are blocked. The second strand, called t_C, contains the attribute-value block of attribute C. No nodes are blocked in this strand. Both components are mandatory. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.

From this point on, we use the type assignment $\Gamma_1 = \Gamma' \cup \{(y, \tau_3)\}$.

5. $\Gamma_1 : \texttt{cleanup}(y) \vdash \tau_4$. Type τ_3 contains two strands, namely, t'_S and t_C. The length of these strands is $(k-1)(4+\ell)$ respectively $4+\ell$. As $k > 1$, strand t'_S qualifies. Because t'_S is mandatory in τ_3, it qualifies for mandatory. Strand t'_S without blockings is called t''_S. Hence, type τ_4 consists of t''_S as a mandatory strand. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.

6. $$\Gamma_1 : \mathit{removeBetweenCirc}(\mathit{circularize}(\texttt{cleanup}(y), Y, X), B', A') \vdash \tau_5$$

   Strand t''_S is mandatory, linear and without blockings and matchings. Its first attribute is X and its last attribute is Y. By Lemma 8.3, we know that c_S is the circular version of t''_S. Strand c_S is mandatory in the output type of the circularize abbreviation. Next, the *removeBetweenCirc* abbreviation cuts open the circle between attributes B' and A'. Type τ_5 thus consists of a linear version of c_S with first attribute A' and last attribute B'. Call this strand t_S. Strand t_S is mandatory in type τ_5. The $\mathfrak{h}$-bit of type τ_5 is set to *true*.

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

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

□

Two attributes.

Lemma 8.12. *Let $e = \widehat{\pi}_C(e_1)$, with $\mathcal{D} : e_1 \vdash R$, $R = \{A, C\}$, and $\mathcal{D} : e \vdash S$, where $S = \{A\}$. Let τ_R be a relation-schema-type of relation schema R, and let τ_S be a relation-schema-type of relation schema S. If expression e^{DNA} is defined as*

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

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

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

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

2. $\Gamma' : \texttt{split}(x, \#_4) \vdash \tau_1$. Strand t_R in type τ'_R, has two nodes labeled $\#_4$, namely, the last nodes of the attribute-value blocks of A and C. Consequently, the split operation splits t_R into two linear strands t_A (the attribute-value block of attribute A) and t_C (the attribute-value block of attribute C). Both strands are mandatory in type τ_1. The $\mathfrak{h}$-bit of type τ_1 is set to *true*.

3. $\Gamma' : \texttt{split}(x, \#_4) \cup \texttt{immob}(\overline{A}) \vdash \tau_2$. Type τ_2 consists of strands t_A, t_C and a probe n labeled $\overline{A}$. Because probe n can match with a node of strand t_A, the $\mathfrak{h}$-bit of type τ_2 is set to *false*. All three components are mandatory.

4. $\Gamma' : \texttt{hybridize}(\texttt{split}(x, \#_4) \cup \texttt{immob}(\overline{A})) \vdash \tau_3$. All components in type τ_2 are mandatory and no node is labeled with ?. Hence, all three components are necessary. The hybridization produces two mandatory components: the first is formed by the attribute-value block of attribute A and the probe, call it C_1, the second component is isomorphic to t_C. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.

5. $\Gamma' : \texttt{cleanup}(\texttt{flush}(\texttt{hybridize}(\texttt{split}(x, \#_4) \cup \texttt{immob}(\overline{A})))) \vdash \tau_4$. The flush operation retains component C_1 because it is immobilized. The cleanup operation removes the probe from component C_1, whence extracting strand t_A from C_1. Strand t_A is mandatory in type τ_4. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.

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

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

□

Renaming

Lemma 8.13. *Let $\mathcal{D} : e_1 \vdash R$ with R a relation schema containing at least two attributes and $C \in R$ and $F \notin R$. Let $e = \rho_{C/F}(e_1)$, with $\mathcal{D} : e \vdash T$, where $T = (R \setminus \{C\}) \cup \{F\}$ is a relation schema. Let A be the first attribute and let E be the last attribute of R. Consider R to be circular, then let B be the attribute just in front of C and let D be the attribute just after C. If expression e^{DNA} is defined as*

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

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

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

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

2. $\Gamma' : \mathit{circularize}(x, A, E) \vdash \tau_1$. Strand t_R is linear, without blockings and matching and mandatory in type τ'_R. By Lemma 8.3, we know that type τ_1 consists of a circular version of strand t_R. Call this circular strand c_R. Strand c_R is mandatory in type τ_1. The $\mathfrak{h}$-bit of type τ_1 is set to *true*.

3. $\Gamma' : \mathit{blockfromto}(\mathit{circularize}(x, A, E), C, D) \vdash \tau_2$. The blockfromto abbreviation constructs a new strand, call it c^2_R, that is isomorphic with c_R, except that starting from the node labeled D up to and including the node labeled C, all nodes are blocked. The only nodes that are not blocked are the ones labeled $\#_3, *, \#_4$, and $\#_2$, following the node labeled C. Strand c^2_R is mandatory in type τ_2. The $\mathfrak{h}$-bit of type τ_2 is set to *true*.

4. $\Gamma' : \texttt{cleanup}(\texttt{split}(\mathit{blockfromto}(\mathit{circularize}(x, A, E), C, D), \#_3)) \vdash \tau_3$. The only free node labeled $\#_3$ in strand c^2_R of type τ_2, is the node

directly following the node labeled C. The split operation makes the circular strand linear again, however, this time the first node is labeled $\#_3$ and the last node is labeled C. Call this strand t_1. There is only one strand in the output of the split operation, whence the cleanup operation removes all blockings from t_1. We call the resulting strand t_2. The $\mathfrak{h}$-bit of type τ_3 is set to *true*.

From this point on, we use the type assignment $\Gamma_{f_1} = \Gamma' \cup \{(f_1, \tau_3)\}$.

5. $\Gamma_{f_1} : \mathit{blockfromto}(f_1, C, D) \vdash \tau_4$. The blockfromto abbreviation constructs a new strand, call it t_3, from strand t_2 in type τ_3. Strands t_2 and t_3 are isomorphic, except that all nodes in t_3 are blocked, except for the first four nodes, labeled $\#_3 * \#_4\#_2$. Strand t_3 is mandatory in type τ_4. The $\mathfrak{h}$-bit of type τ_4 is set to *true*.

6. $\Gamma_{f_1} : \mathit{blockfromto}(f_1, C, D) \cup \#_2 F \cup \overline{F\#_3}) \vdash \tau_5$. Type τ_5 consists of strand t_3, strand t_4 labeled $\#_2 F$ and sticker s_1 labeled $\overline{F\#_3}$. All three components are mandatory in type τ_5. The $\mathfrak{h}$-bit of type τ_5 is set to *false*, because matchings are possibly between t_3 and s_1 and between t_4 and s_1.

7. $\Gamma_{f_1} : \mathit{connect}(\mathit{blockfromto}(f_1, C, D) \cup \#_2 F \cup \overline{F\#_3}) \vdash \tau_6$. The $\mathfrak{h}$-bit of type τ_5 is *false*, whence hybridization takes place. As all components in type τ_5 are mandatory and no node is labeled ?, there is only one set X to hybridize. The only free node labeled F is the second node of t_4. The only free node labeled $\#_3$ is the first node of t_3. Hence, the result of the hybridization is a new component, consisting of one isomorphic copy of strand t_3, one isomorphic copy of strand t_4, and one isomorphic copy of sticker s_1. The sticker isomorphic to s_1 binds the strand isomorphic to t_4 to the front of the strand isomorphic to strand t_3. Consequently, there is a gap between the positive strands.

   The ligate operation fills the gap between the positive strands, uniting them in a single strand. The cleanup operator constructs a new strand, call it t_6, which is the concatenation of strands t_4 and t_3. Strand t_6 is mandatory in type τ_6. The $\mathfrak{h}$-bit of type τ_6 is set tot *true*.

   From this point on, we use the type assignment $\Gamma_{f_2} = \Gamma_{f_1} \cup \{(f_2, \tau_6)\}$.

8. $\Gamma_{f_2} : \texttt{blockfrom}(f_2, B) \vdash \tau_7$. The blockfrom operation constructs a new strand, call it t_7, isomorphic to t_6, except that all nodes starting from the first node of the strand up to the node labeled B are blocked. Consequently, the only free node labeled $\#_2$ is the second to last node, just in front of the node labeled C. Strand t_7 is mandatory in type τ_7 and the $\mathfrak{h}$-bit of type τ_7 is set to *true*.

9. $\Gamma_{f_2} : \texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(f_2, B), \#_2)) \vdash \tau_8$. The split operation constructs two new strands, call them t_8 and t_9. Strand t_8 is isomorphic to strand t_7, except that the last two nodes of t_7 are removed. Strand t_9 consists of two nodes labeled $\#_2 C$. Both strands are mandatory. The length of strand t_8 is $|R|(4+\ell)$. The length of strand t_9 is 2. Hence, type τ_8 consists of strand t_8, which is mandatory in τ_8. The $\mathfrak{h}$-bit of type τ_8 is set to *true*.

   From this point on, we use the type assignment $\Gamma_{f_3} = \Gamma_{f_2} \cup \{(f_3, \tau_8)\}$.

10. $\Gamma_{f_3} : \mathit{circularize}(f_3, F, B) \vdash \tau_9$. Strand t_8 is mandatory in type τ_8. It starts with attribute F and ends with attribute B. Hence, by Lemma 8.3, we know that type τ_9 consists of the circular version of t_8. Call this circular strand c'_T. The $\mathfrak{h}$-bit of type τ_9 is set to *true*.

11. $\Gamma_{f_3} : \mathit{blockfromto}(\mathit{circularize}(f_3, F, B), E, A) \vdash \tau_{10}$. Attributes A and E are adjacent on the circle. The blockfromto abbreviation constructs a new strand, call it c''_T, which is isomorphic to c'_T, except that all nodes from E up to and including A are blocked. There is a single free node labeled $\#_2$ in c''_T, namely, the node between the attribute-value blocks of attributes A and E. Strand c''_T is mandatory in type τ_{10}. The $\mathfrak{h}$-bit of type τ_{10} is set to *true*.

12. $$\Gamma_{f_3} : \texttt{cleanup}(\texttt{split}(\mathit{blockfromto}(\mathit{circularize}(f_3, F, B), E, A), \#_4)) \vdash \tau_{11}$$

    The split operation splits strand c''_T between attributes E and A. Call the resulting strand t'_T. Attribute A is the first on strand t'_T, attribute E is the last on strand t'_T. There is only one strand, namely, t'_T. Hence, the cleanup operation constructs a new strands t_T isomorphic to strand t'_T except that it has no blockings.

13. $$\begin{array}{ll}\Gamma' : & \texttt{if empty}(x) \texttt{ then empty else} \\ & \texttt{let } f_1 := \texttt{cleanup}(\texttt{split}(\mathit{blockfromto}(\\ & \quad \mathit{circularize}(x, A, E), C, D), \#_3)) \texttt{ in} \\ & \texttt{let } f_2 := \mathit{connect}(\mathit{blockfromto}(f_1, C, D) \cup \#_2 F \cup \overline{F \#_3}) \texttt{ in} \\ & \texttt{let } f_3 := \texttt{cleanup}(\texttt{split}(\texttt{blockfrom}(f_2, B), \#_2)) \texttt{ in} \\ & \texttt{ cleanup}(\texttt{split}(\mathit{blockfromto}(\mathit{circularize}(f_3, F, B), E, A), \#_4)) \\ & \vdash \tau_T\end{array}$$

    The then-part of the if-statement has the empty type. The else-part of the if-statement has type τ_{11}. Combining both types, results in type τ_T.

□

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

Selection

Lemma 8.14. *Let $e = \sigma_{B=D}(e_1)$, with $\mathcal{D} : e_1 \vdash R$ and $\mathcal{D} : e \vdash R$. Let A be the first attribute of R and let F be the last attribute of R. Let C be the attribute directly following attribute B. Let E be the attribute directly following attribute D. If expression e^{DNA} is defined as*

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

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

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

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

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

2. $\Gamma' : \mathit{circularize}(x_s, A, F) \vdash \tau_c$. Strand t_R in τ'_R is mandatory, has first attribute A and last attribute F. Let c_R be the circularized version of

the t_R. Type τ_c consists of strand c_R. Strand c_R is mandatory in type τ_c. The $\mathfrak{h}$-bit of type τ_c is set to *true*.

From this point on, we use the type assignment $\Gamma_c = \Gamma' \cup \{(x_c, \tau_c)\}$.

3. $\Gamma_c : \mathit{blockselect}(x_c, B, C, a) \vdash \tau_y$. Type τ_c consists of a mandatory circular strand, strand c_R. On strand c_R attribute B is followed by attribute C. By Lemma 8.6, we know that type τ_y thus consists of strand c_R. Strand c_R is non-mandatory in type τ_y.

   From this point on, we use the type assignment $\Gamma_y = \Gamma_c \cup \{(y, \tau_y)\}$.

4. Γ_y : `if empty`(y) `then empty else` $\mathit{blockselect}(y, D, E, a) \vdash \tau_y$. Strand c_R in type τ_y is non-mandatory and circular. To type check the else-part of the if-statement, we may assign complex variable y type τ_c. Hence, by Lemma 8.6 we know that the output type of $\mathit{blockselect}(y, D, E, a)$ equals τ_y. Combining the empty type of the then-part with τ_y of the else-part, results again in τ_y.

5. $\Gamma_c : \bigcup_{a \in \Lambda}$ `let` $y :=$ $\mathit{blockselect}(x_c, B, C, a)$ `in`
   `if empty`(y) `then empty else` $\mathit{blockselect}(y, D, E, a) \vdash \tau_y$. All parts of the union over all atomic value symbols have the same type, namely, τ_y. Hence, the union has type τ_y.

   From this point on, we use the type assignment $\Gamma_f = \Gamma' \cup \{(f, \tau_y)\}$.

6. Γ_f : `cleanup(split(`$\mathit{blockfromto}(f, F, A), \#_4$`))` $\vdash \tau_R$. The *blockfromto* abbreviation constructs a new circular strand, call it c'_R, isomorphic to c_R except all nodes the nodes from A to F are blocked. Consequently, The only free node labeled $\#_4$ is the last node of the attribute-value block of attribute F. Hence, the split operation constructs a strand t'_R which is isomorphic to t_R except that all nodes from A to F are blocked. Because there is only one strand, the cleanup operation results in a strand isomorphic to t_R. Strand t_R is not mandatory in type τ_R.

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

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

8.

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

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

□

8.3 Maximality and Tightness for Non-Atomic Expressions

We introduced the notions of maximality and tightness on arbitrary DNAQL expressions. However, Theorems 7.4 and 7.5 apply to atomic expressions only. In this section, we show that a maximal typing relation on DNAQL is undecidable, and that the typing relation is not tight for arbitrary expressions due to the interplay between union and the $\mathfrak{h}$-bit. An interesting future direction of research is to come up with a tight type relation or proving that a tight type relation is undecidable.

Let us first examine the maximality of a DNAQL typing relation. It is undecidable whether a relational algebra expression always outputs the empty relation [1]. Let e be a relational algebra expression. Expression e can be translated to an equivalent DNAQL expression e^{DNA} (as proven in this chapter). 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 is maximal on arbitrary expressions, expression e' would type check whenever expression e always outputs the empty relation. This is a contradiction as the emptiness problem is undecidable.

Secondly, we show by counterexample that the DNAQL typing relation is not tight on expressions. Consider the types shown in Figure 8.3. 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.

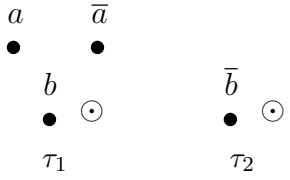


Figure 8.3: Two types τ_1 and τ_2. The types consist of one-node components. Both types have their $\mathfrak{h}$-bit, $\mathfrak{h}_1$ resp. $\mathfrak{h}_2$, set to *true*.

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 is the hybridization of two nodes, one labeled a, the other labeled $\overline{a}$. 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. On the other hand, any complex having type τ_2 cannot have a node labeled a or a node labeled $\overline{a}$.

8.4 4-Bounded

In Chapter 4 we defined c-bounded hybridization which is guaranteed to produce only a polynomial-sized output. Here we show that the relational algebra simulation is 4-bounded, by a careful analysis of the simulation programs.

8.4.1 Abbreviations

Circularize

The abbreviation $\mathit{circularize}(x, A, B)$ is defined as:

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

We assume that the complex variable x has a pseudo-relation-schema-type, with first attribute A and last attribute B. The first hybridization of has only two nodes that can interact because all $\#_3$ in the strands of x are blocked except for one. Hence, there are no choice nodes. In the second hybridization, part of the connect abbreviation, there are again no choice nodes because there is only one free $\#_2$ and only one free $\#_4$. Hence, the abbreviation is 4-bounded.

Inserting into a Circle

The abbreviation $\mathit{insertcirc}(x, A, B, s)$ is defined as:

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

Because of the blocking in the first line, there is only one free $\#_3$, meaning there are no choice nodes in the complex. In the second hybridization, there are also no choice nodes because of blockings. Finally, the hybridization in the *connect* abbreviation has no choice nodes either, because there is only one free node labeled $\#_4$ and only one free node labeled σ_2.

Removing from a Circle

There are no hybridizations in this abbreviation.

Block Selecting

The abbreviation $\mathit{blockselect}(x, A, B, a, i)$ is defined as:

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

All ℓ-vectors except one is blocked. That ℓ-vector is block-excepted, i.e., only one atomic value symbol is free when hybridization is applied, hence there is no choice node.

8.4.2 Relational Algebra Expressions

Union

There is no hybridization.

Difference

There is no hybridization.

Cartesian Product

The first part of the simulation of the cartesian product consists of concatenating the strands representing the tuples in e_1^{DNA} and e_2^{DNA}. Concatenating is defined as follows:

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

Constructing variables r and l is 0-bounded, because there is only one possible interaction. Attaching r to the strands in x encounters one choice node, viz., $\overline{\#_4}$ in the component of r. Hence, the hybridization is 1-bounded. A similar reasoning shows that combining y and l is also a 1-bounded hybridization. Next, strands are concatenated in e_2. There are two choice nodes, viz., the two nodes of the sticker $\overline{\#_5\#_1}$. The are both reachable from the sticker by an alternating path of length 1. Hence, this hybridization is 2-bounded. We have already shown that circularize is 4-bounded.

The second part of the cartesian product simulation consists of shuffling

attributes. Recall the definition of shuffling:

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

We only need to analyze the hybridizations in f_4 and f_5. In f_4 all $\#_4$ nodes are blocked, hence there is no choice node. In variable f_5 each node of the stickers $\overline{\#_6\#_8}$ and $\overline{\#_1\#_7}$ is a choice node. As each tuple is immobilized, hybridization is terminating. Nonetheless, 4 choice nodes are accessible from each component of the complex. Hence, hybridization is 4-bounded.

Projection

Three or More Attributes. This program only contains the circularize and removebetweencirc abbreviations. Hence, this program is 2-bounded.

Two Attributes. If there are just two attributes, then we need to analyze one hybridization.

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

Complex variable X has type $\#_2A\#_3 * \#_4\#_2C\#_3 * \#_4$. The hybridization has exactly no choice nodes, because there is only one node labeled A and one node labeled $\overline{A}$.

Renaming

This program only contains the circularize abbreviation, which is 2-bounded.

Selection

The heavy lifting is done by the blockselect abbreviation, which is 2-bounded. All other operations and abbreviations perform no hybridization.

Conclusion

The double bridging scheme is the "heaviest" hybridization utilized in the simulation of the relational algebra. This hybridization is 4-bounded. Hence, we may concluded that the simulation of the relation algebra is 4-bounded.

9

Discussion

We have developed an abstract model of the DNA molecule, called the sticker complex model, aimed at data storage and data manipulation on the application level and aimed at practical viability on the implementation level. On the sticker complex model an applicative programming language is defined with an emptiness-test, a for-loop and a let statement, called DNAQL. The sticker complex model and its programming language DNAQL is a descendant of Adleman-style DNA computing, with a set of operators implemented by enzymes alongside the omnipresent hybridization. Nonetheless, its focus on data manipulation makes it substantially different from previous computing models. Turing completeness is sacrificed for efficiency and optimizability, as is conventional in database research. Indeed, databases need to handle vast amounts of data and quickly answer a continuous stream of queries. Hence, queries need to be efficiently answerable. One issue in the sticker complex mode, is that the DNAQL operator hybridization can result in infinite complexes. A thorough analysis of this issue resulted in a characterization of terminating hybridization. This laid the foundation of a type system that identifies well behaving programs. The usefulness of the model and the programming language with regards to data storage and manipulation are demonstrated by a simulation of the relational algebra, a corner stone of current database systems. Moreover, the simulation proves to be well typed establishing the power of the type system.

That being said, there are still many open research questions. First of all, the simulation of the relational algebra defines a lower bound on the expressive power of the sticker complex model with DNAQL. But, exactly which types

of programs can be programmed within this model? Is it possible to simulate sticker complexes and DNAQL with relational databases and the relational algebra? A positive answer implies a strict bound on the expressiveness of sticker complexes and DNAQL. The hybridization operation seems to be extremely powerful and beyond the scope of first order logic. Be that as it may, the emphasis lays on the data carried by a complex, and it might just be that the information processing capabilities of sticker complexes and DNAQL are on par with relational databases, the relational algebra and the sticker complex type system. In other words, if we would concentrate on the data, an algorithm may be devised to translate arbitrary DNAQL expressions to relational algebra expressions if typing information is present. Secondly, we have shown that the DNAQL type relation is sound for programs and it is maximal and tight for operators. A type relation that is maximal on programs is impossible because it is undecidable whether a query will evaluate to an empty relation on any well-typed database. Nonetheless, more powerful types may allow for a type system that is tight on programs as well. The main hurdle is the subtle nuances introduced by the hyb-bit of a sticker complex type. A possible track is maintaining "union types", in which case a set of types is considered and a complex has a certain union type if it has one of its types in the sense described in this dissertation. On the other hand, such an approach might make types prohibitively large.

Chemical reactions are quantitative and stochastic by nature. In contrast, the sticker complex model and DNAQL are qualitative and deterministic. In the sticker complex model, we have assumed that replicating data and error-correction techniques can counter undesirable effects of chemical reactions on DNA molecules. Although we believe that the sticker complex model is a valuable tool for designing DNA computers and researching their properties, a model incorporating the quantitative and stochastic nature of DNA computing is called for. The situation is analogue to the Abstract Tile Assembly Model (aTAM) and Kinetic Tile Assembly Model (kTAM). A *quantitative and stochastic (QS) model* of sticker complexes should not be confused with the theoretical models in vogue in chemistry. A QS model compares to chemistry models as Newtonian mechanics compares to quantum mechanics. Although Newtonian mechanics is not correct on the atomic scale, it is accurate enough for any object observable with the naked eye. Furthermore, the mathematics involved in Newtonian mechanics are easier to compute than the mathematics underlying quantum mechanics. Hence for many applications Newtonian mechanics is preferred over quantum mechanics because its error is negligible. Likewise, a QS model operates on DNA molecules of 100 or more bases, this is several orders of magnitude larger than the atoms on which the chemistry models operate. Consequently, we can trade some accuracy for simplicity.

The merit of a QS model is judged on four criteria: (1) the *expressiveness*

of the representation, (2) the *efficiency* of the programming language, (3) the *expressive power* of computations in the model, and (4) the *predictive accuracy* with regards to the actual sticker DNA computer. The expressiveness of a representation relates to its ability to distinguish between different states of a sticker DNA computer. Equally important, the transformations of the representation, mimicking the state transitions of the sticker DNA computer, should be efficiently computable, preferably in polynomial time. Otherwise, the model becomes useless for simulations. Thirdly, the model should support the expression of powerful computations. In particular, the relational algebra should be expressible, under certain conditions, e.g., return the correct answer with high probability. Finally, if a model predicts that a sticker DNA computer produces a certain output, concrete runs of the sticker DNA computer should fall within the prediction.

In moving from a qualitative model to a quantitative and stochastic (QS) model more is involved than assigning quantities to the components of a sticker complex, where a component is a maximal set of strands and stickers connected, possibly indirectly, by hybridization bonds. At the start of a computation in a sticker DNA computer, there is almost absolute certainty about the state of the DNA computer. Tight intervals on the quantities of the DNA strands initiating the computation can be measured. Each step in the computation introduces uncertainty into the state of the DNA computer, due to the stochastic nature of chemical processes and the possibility of errors. Indeed, although for each concrete run of the DNA computer the exact quantities of all involved components can be measured at any given point in the computation, the state kept by a QS model must account for each possible run. Moreover, at a given point in the execution of a sticker DNA computer, some assignments of quantities to components are more likely than other assignments. For example, if strand X and strand Y can bond, then after a hybridization it is very likely that X and Y occur less than their product XY.

The problem of defining a suitable representation is not unique to DNA computing, it is also encountered in the context of *probabilistic databases* [14]. A probabilistic database manages uncertain data, i.e., the database may contain conflicting information and each piece of information is, with some probability, correct or wrong. An example of uncertain data is a sensor network in which some sensors may occasionally produce wrong measurements. Notwithstanding the link with probabilistic databases, the representation problem for QS models is worthwhile to pursue because several crucial differences exist between probabilistic databases and QS models for sticker DNA computers that affect the design of a suitable representation. Firstly, probabilistic databases and QS models are geared towards on different types of computations: probabilistic databases are focused on querying data, whereas a QS model is focused on transformations. Computing a query may be intractable in a representa-

tion, yet computing a transformation of the representation could be tractable. Secondly, a probabilistic database manages relational data, whereas a QS model manages quantities of a set of objects, hence a QS model deals with inherently simpler and unstructured data. Thirdly, probabilistic databases are qualitative, whereas a QS model is quantitative. Fourthly, a QS model can take advantage of knowledge about the behavior of DNA molecules. A QS model representation can assume a certain type of probability distribution, whereas the more general probabilistic database must be prepared for any type of probability distribution.

A joint probability distribution on the quantities of all the possible components maximizes the representational expressiveness: every possible situation is represented and is assigned a probability of occurring. However, due to the stochastic nature of DNA computing and the presence of errors, the number of possible combinations rapidly explodes, making the storage of and computations on such a distribution potentially hard. One way of alleviating this drawback is switching to a probabilistic graphical model, e.g., a Bayesian network. Probabilistic graphical models allow, under certain conditions, a more compact representation of an otherwise huge joint probability distribution, by decomposing the joint probability distribution into smaller conditional distributions over subsets of correlated components. Nonetheless, in the worst case all components are correlated, then a Bayesian network looses its edge. To avoid such a situation, the number of components allowed in any conditional distribution of a Bayesian network can be restricted. This results in a hierarchy of representations. On one side of the hierarchy is the Bayesian network, with no restrictions on the size of its distributions. On the other side of the hierarchy, only one component per distribution is allowed, thus the representation defines a separate probability distribution on each quantity. Maintaining separate probability distributions prohibits modeling correlations between the quantities of components. Let X and Y be strands that can bond, the quantities of X, Y and their product XY will be highly correlated. The hypothesis is that allowing more components results in greater expressiveness and a greater computational cost. Finding the tipping point, at which the computational cost becomes excessive, is an interesting research question.

Up till now, the merit of QS models is judged on purely theoretical standards. Desirably, a concrete instantiation of a QS model is an accurate predictor of real life DNA computations. Instantiating a QS model of sticker DNA computers demands estimating numerous parameters involving the stochastic processes underlying the operations and errors. For example, recall the strand ab and the sticker AB, to accurately model the behavior of the hybridization operation, it is essential to know the probability distribution over all the possible conformations that can be formed by combining one or more copies of the strand and sticker. Concretely, if there are x copies of strand ab and y

copies of sticker AB, how many copies will there be in which one copy of the strand is bent into a circle by one copy of the sticker?

To estimate the parameters, data on the behavior of the DNA molecule must be available. Most DNA computing research groups are equipped with a staffed wet-lab. However, smaller or theoretical groups do not have the funds or knowledge to operate a wet-lab, hence such groups need to fall back on simulations. Nonetheless, theoretical results validated by simulations are still rare. I intend to use simulation software to collect the necessary data to instantiate the proposed models. Concretely, the oxDNA simulation software appears to be the best option, because it is an open source implementation of a course-grained model of the DNA molecule that has proven to be both fast and accurate on larger DNA molecules, i.e., more than one hundred bases [31, 49]. Because simulation backed publications are rare, the first step in collecting the necessary data with simulation software is setting up a protocol to work in a uniform and systematic manner. Questions that need to be addressed are for example: how many repetitions are needed, what is the correct size of the bounding box constraining the movement of DNA molecules, how to store results, and what about post-processing of results? Once the simulation protocol is established, concrete simulations can be run. For example, for estimating the probability distribution on the hybridization of strand ab and sticker AB, many simulations can be run with different initial quantities of the strand and the sticker. The output of these simulations is post- processed to count, for each distinct conformation, the number of occurrences. These numbers form the basis for estimating the probability distribution.

Analyzing the data collected from simulations will draw on the extensive research on inferring probabilistic graphical models [18]. This is a natural fit to the discussed hierarchy of quantitative and stochastic models.

Bibliography

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

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

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

[4] M. Arita, M. Hagiya, and A. Suyama. Joining and Rotating Data with Molecules. In *International Conference on Evolutionary Computation*, pages 243–248, 1997.

[5] R. Braich, N. Chelyapov, C. Johnson, P. Rothemund, and L. Adleman. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296:499–502, 2002.

[6] R. Brijder, J. Gillis, and J. Van den Bussche. A comparison of graph-theoretic DNA hybridization models. *Theoretical Computer Science*, 429:46–53, 2011.

[7] L. Cardelli. Strand Algebras for DNA Computing. In R. Deaton and A. Suyama, editors, *15th International Meeting on DNA Computing and Molecular Programming*, volume 5877 of *Lecture Notes in Computer Science*, pages 12–24, Fayetteville, 2009. Springer.

[8] L. Cardelli. Two-Domain DNA Strand Displacement. In S. Cooper, P. Kashefi, and P. Panangaden, editors, *Developments in Computational Models*, pages 47–61, 2010.

[9] L. Cardelli. Two-Domain DNA Strand Displacement. *Mathematical Structures in Computer Science*, In print, 2012.

[10] H. Chen and A. Goel. Error Free Self-assembly Using Error Prone Tiles DNA Computing. In C. Ferretti, G. Mauri, and C. Zandron, editors, *10th*

International Conference on DNA Computing, volume 3384 of *Lecture Notes in Computer Science*, pages 702–707. Springer Berlin / Heidelberg, 2005.

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

[12] G. M. Church, Y. Gao, and S. Kosuri. Next-Generation Digital Information Storage in DNA. *Science*, 337(6102):1628, 2012.

[13] A. Condon, A. J. Hu, J. Manuch, and C. Thachuk. Less haste, less waste: on recycling and its limits in strand displacement systems. *Interface Focus*, 2(4):512–21, 2012.

[14] N. Dalvi, C. R, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

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

[16] L. Diatchenko, Y. Lau, A. Campbell, A. Chenchik, F. Moqadam, B. Huang, S. Lukyanov, K. Lukyanov, N. Gurskaya, E. Sverdlov, and P. 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.

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

[18] N. Friedman. Inferring cellular networks using probabilistic graphical models. *Science*, 303(5659):799–805, 2004.

[19] H. Garcia-Molina, J. 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] N. Goldman, P. Bertone, S. Chen, C. Dessimoz, E. LeProust, B. Sipos, and E. Birney. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature*, 2013.

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

[23] J. Hartmanis. On the Weight of Computations. *Bulletin of the EATCS*, 55, 1995.

[24] N. Jonoska and G. L. McColm. Complexity classes for self-assembling flexible tiles. *Theoretical Computer Science*, 410(45):332–346, 2009.

[25] N. Jonoska, G. L. McColm, and A. Staniska. On stiochiometry for the assembly of flexible tile DNA complexes. *Natural Computing*, 10(3):1121–1141, 2011.

[26] T. Lempiinen, E. Czeizler, and P. Orponen. Synthesizing Small and Reliable Tile Sets for Patterned DNA Self-Assembly. In L. Cardelli and W. Shih, editors, *17th International Conference on DNA Computing and Molecular Programming*, Lecture Notes in Computer Science, Pasadena, CA, USA, 2011. Springer.

[27] R. Lipton. DNA solution of hard computational problems. *Science*, 268(5120):542–545, 1995.

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

[29] U. Majumder and J. Reif. A Framework for Designing Novel Magnetic Tiles Capable of Complex Self-assemblies. In C. Calude, J. Costa, R. Freund, M. Oswald, and G. Rozenberg, editors, *Unconventional Computing*, volume 5204 of *Lecture Notes in Computer Science*, pages 129–145. Springer Berlin / Heidelberg, 2008.

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

[31] T. Ouldridge. *Coarse-grained modelling of DNA and DNA self-assembly.* PhD thesis, University of Oxford, 2011.

[32] 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.

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

[34] A. Phillips and L. Cardelli. A programming language for composable DNA circuits. *Journal of the Royal Society Interface*, 6:419–436, 2009.

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

[36] L. Qian and E. Winfree. A simple DNA gate motif for synthesizing large-scale circuits. *Journal of the Royal Society Interface*, 2011.

[37] L. Qian and E. Winfree. Scaling Up Digital Circuit Computation with DNA Strand Displacement Cascades. *Science*, 332:1196–1201, 2011.

[38] L. Qian, E. Winfree, and J. Bruck. Neural Network computation with DNA strand displacement cascades. *Nature*, 475:368–372, 2011.

[39] J. Reif. Local Parallel Biomolecular Computation. In *DNA Based Computers III*, volume 48, pages 217–254. DIMACS, 1997.

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

[41] J. Reif, T. LaBean, M. Pirrung, V. Rana, B. Guo, C. Kingsford, and G. 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*, Lecture Notes in Computer Science, pages 231–247, London, UK, 2002. Springer-Verlag.

[42] P. Rothemund and E. Winfree. The program-size complexity of self-assembled squares. In *Thirty-Second Annual ACM Symposium on Theory of Computing*, pages 459–468. ACM Press, 1999.

[43] S. Roweis, E. Winfree, R. Burgoyne, N. Chelyapov, M. Goodman, P. Rothemund, and L. Adleman. A Sticker-Based Model for DNA Computation. *Journal of Computational Biology*, 5(4):615–629, 1998.

[44] 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.

[45] G. Seelig, D. Soloveichik, D. Zhang, and E. Winfree. Enzyme-Free Nucleic Acid Logic Circuits. *Science*, 314(1585–1588), 2006.

[46] M. Shortreed, S. Chang, D. Hong, M. Phillips, B. Campion, D. Tulpan, M. Andronescu, A. Condon, H. Hoos, and L. Smith. A thermodynamic approach to designing structure-free combinatorial dna word sets. *Nucleic Acids Research*, 33(15):4965 – 4977, 2005.

[47] D. Soloveichik, M. Cook, and E. Winfree. Combining self-healing and proofreading in self-assembly. *Natural Computing*, 7(2):203–218, 2008.

[48] D. Soloveichik, G. Seelig, and E. Winfree. DNA as a Universal Substrate for Chemical Kinetics. *LNCS*, 5347:57 – 69, 2009.

[49] P. Sulc, F. Romano, T. Ouldridge, L. Rovigatti, J. Doye, and A. Louis. Sequence-dependent thermodynamics of a coarse-grained DNA model. *Journal of Chemical Physics*, 137, 2012.

[50] 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.

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

[52] H. Wang. Proving theorems by pattern recognition. *Bell System Technical Journal*, 40:1–42, 1961.

[53] E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, 1998.

[54] E. Winfree and R. Bekbolatov. Proofreading Tile Sets: Error Correction for Algorithmic Self-Assembly. In *10th International Workshop on DNA Computing*, volume 2943 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2003.

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

[56] P. Yin, B. Guo, C. Belmore, W. Palmeri, E. Winfree, T. LaBean, and J. Reif. TileSoft: Sequence Optimization Software for Designing DNA Secondary Structure. In *10th International Workshop on DNA Computing*, volume 3384 of *Lecture Notes in Computer Science*, 2004.

[57] D. Zhang, A. Turberfield, B. Yurke, and E. Winfree. Engineering Entropy-Driven Reactions and Networks Catalyzed by DNA. *Science*, 318:1121–1125, 2007.