Compiling DNA strand displacement reactions using a functional programming language

Matthew R. Lakin[1,2] and Andrew Phillips[1]

[1] Biological Computation Group, Microsoft Research, Cambridge, CB1 2FB, UK
[2] Department of Computer Science, University of New Mexico, NM 87131, USA

mlakin@cs.unm.edu, andrew.phillips@microsoft.com

Abstract. DNA nanotechnology is a rapidly-growing field, with many potential applications in nanoscale manufacturing and autonomous *in vivo* diagnostic and therapeutic devices. As experimental techniques improve it will become increasingly important to develop software tools and programming abstractions, to enable rapid and correct design of increasingly sophisticated computational circuits. This is analogous to the need for hardware description languages for VLSI. In this paper we discuss our experience implementing a domain-specific language for DNA nanotechnology using a functional programming language. The ability to use abstract data types to describe molecular structures and to recurse over these types to derive the various interactions between structures was a major reason for the use of a functional language in this project.

Keywords. DNA strand displacement, process calculus, biological modelling.

1 Introduction

DNA is an attractive engineering material for controlling matter at the nanoscale, as it is robust and undergoes predictable, sequence-specific, programmable interactions. Previous work has shown that synthetic DNA circuits can be used to implement computational systems including digital logic circuits [1], neural networks [2] and game-playing automata [3]. In this setting, DNA is used both as an information carrier *and* as an engineering material, simultaneously. Furthermore, DNA is inherently biocompatible, meaning that DNA-based computing devices could feasibly operate in living cells, autonomously monitoring the cell state and administering appropriate treatment for diseases at the cellular level [4].

As the scale and complexity of DNA-based computing devices continues to grow, tool support will become ever more important. A key goal is to formalize the structures and interactions of DNA molecules, so that their behaviour may be analyzed [5]. To this end we developed a domain-specific language known as DSD [6], which is a process calculus for describing a particular class of DNA circuits that interact via *strand displacement* reactions [7]. Prior to the development of the DSD language, strand displacement circuits were largely designed by hand, which was time-consuming and not scalable. The key aspects of the DSD language design are its syntax for representing a particular class of DNA structures, and the operational semantics which models the real-world interactions between those structures. We implemented a compiler, stochastic and deterministic simulators and state space analysis tools for the DSD language in

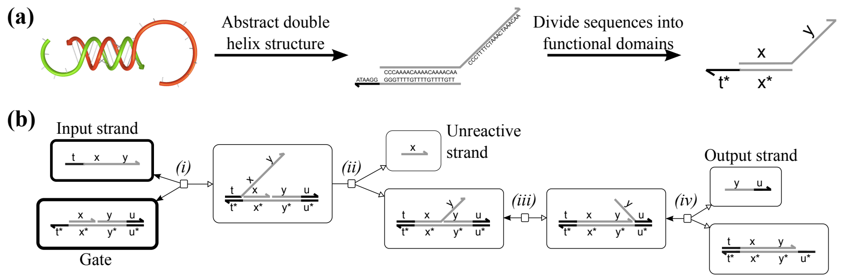


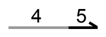
Fig. 1. (a) DNA secondary structure abstraction. (b) Basic strand displacement reactions.

F# [8], and in this paper we describe the experience of modelling and compiling DNA reactions in a functional language.

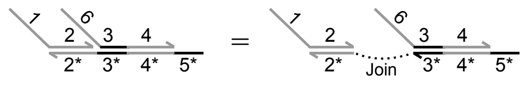
2 DNA strand displacement

DNA strand displacement [7] is a robust mechanism for engineering sequence-specific interactions between DNA molecules. As shown in Figure 1a, we use the *secondary structure* abstraction of DNA structure, which ignores the double helical structure and absolute positions of the molecules and represents the relative positions of the strands by parallel lines, with arrowheads denoting strand orientations. Instead of dealing with nucleotide sequences (G, A, T, C) we define *domains* as shorthands for particular finite sequences. We write x^* for the *complement* of the domain x, which is the domain that binds to x. This is defined by the standard Watson-Crick base-pairing rules for DNA ($G \leftrightarrow C$, $A \leftrightarrow T$). We assume that domains have been chosen to be non-interfering, so that each domain only binds to its complement. Domains are divided into *toeholds* (drawn in black in figures and denoted by a caret in the text), which are sufficiently short that they bind reversibly to their complements, and *long domains* (drawn in grey in figures), which are sufficiently long that they do not spontaneously unbind from their complements. Toeholds are identified by a caret, for example $t\hat{}$ and $t\hat{}^*$.

Figure 1b illustrates the fundamental reactions involved in DNA strand displacement, in which a single strand of DNA interacts with a multi-strand complex, which we call a *gate*. In reaction *(i)*, the input strand binds reversibly to the gate via the toehold $t\hat{}$. The next long domain on the input strand matches the neighbouring domain on the gate structure, which allows the remainder of the input strand to continue binding to the gate across the x reaction, as in the *strand displacement* reaction *(ii)*. In this reaction, the input strand completely displaces another strand from the gate. (We refer to this displaced strand as *unreactive* because it contains no toeholds, and we require that the only exposed complementary domains are toeholds.) Since the remaining domains also match, the input strand can displace the y domain from the input gate, in a reversible *branch migration* reaction *(iii)*. Finally, when the output strand is only bound to the gate by the toehold $u\hat{}$, the output strand may unbind, as in reaction *(iv)*, which is reversible because the output strand may rebind to the gate via the exposed toehold $u\hat{}$.

(a) 

```
Upper([Long("4",false), Toe("5",false)]) : strand
```

(b) 

```
LowerJoin(Seg([Long("1",false)],[],[Long("2",false)],[],[]),
          Seg([Long("6",false)],[],[Toe("3",false),Long("4",false)],
              [],[Toe("5",true)])) : gate
```

Fig. 2. DNA strand and gate structures, and their translation to abstract data types. Examples are from [9]. (a) An upper strand. (b) A gate made by joining two segments along their lower strand.

Despite their apparent simplicity, strand displacement reactions like those outlined in Figure 1b are capable of rich behavior. Since the output strand produced by one strand displacement reaction may serve as the input to another reaction, strand displacement systems may be scaled up to produce more complex circuits. Our goal is to formalize these structures and their reactions, at various levels of detail.

3 Modelling DNA structures

We consider the class of DNA structures introduced in [6]: either single *strands* or multi-strand *gates*. For user convenience, we distinguish between *upper* and *lower* strands. A gate is made up of one or more *segments*, which consist of a double-stranded section of one or more complementary domains and possible single-stranded overhanging regions. A segment is connected to its neighbour by joining *either* the upper or the lower strand. These structures are translated to the following ML data types:

```
type domain = Toe of (string * bool) | Long of (string * bool)
type strand = Upper of domain list | Lower of domain list
type segment = Seg of domain list * domain list * domain list
                * domain list * domain list
type gate = Single of segment
          | LowerJoin of segment * gate
          | UpperJoin of segment * gate
```

In the case of domains, the `string` represents the name of the domain and the `bool` represents whether that domain is complemented. The translation of DNA structures into these data types is illustrated in Figure 2. Note that if two segments are joined across an overhang, then there are multiple ways to express the resulting structure: hence the representation is not unique. Therefore, we normalize structures to a common representation by gathering overhangs on joining strands, using the following functions

```
let normLower Seg(L1,L1',S1,R1,R1') Seg(L2,L2',S1,R2,R2')
          = (Seg(L1,L1',S1,R1,(R1'@L2')),Seg(L2,[],S1,R2,R2'))
```

of type `segment -> segment -> (segment * segment)`, and a corresponding function `normUpper` for upper strand joins. By applying the appropriate normalization function to each segment join in the gate structure, we obtain a canonical gate representation.

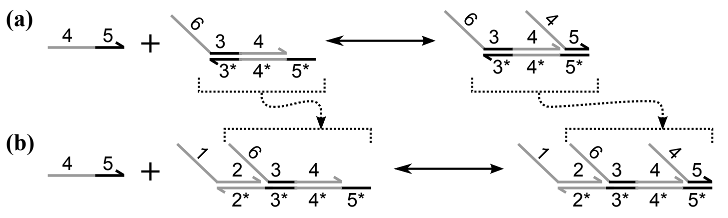


Fig. 3. Enumerating DNA reactions, using the example species from Figure 2. (a) An upper strand binding to a single segment. (b) An upper strand binding to a multi-segment gate, which is inferred from the segment-level interaction.

4 Compiling DNA reactions

To analyze the dynamic behavior of DNA strand displacement systems it is necessary to enumerate the interactions between the species. We achieve this by defining a compiler that takes a set of initial DNA species as input and produces the set of all possible generated species and all possible reactions that could occur. This enables the dynamic behaviour of a strand displacement system to be simulated before one attempts a laboratory implementation. We begin by defining a type for reactions:

```
type species = Strand of strand | Gate of gate
type reaction = Reac of species list * float * species list
```

The structures of the abstract syntax trees for the DNA strands and gates guide the definition of the functions that enumerate reactions—consider the problem of enumerating all possible reactions between a strand and a gate. A free upper strand will only bind to the exposed lower single-stranded parts of a gate, and only then if a complementary pair of toeholds are present. As in the definition of the structure normalization function described above, the basic approach here is to write a "segment-level" compilation function that enumerates all possible ways that a strand can interact with a particular segment:

```
strandBindsToSegment : strand -> segment -> segment list
```

This function returns a list of all segments that could result from the binding of the strand in question to the segment, as shown in Figure 3a. Using a custom `map`-like functional that collects all the segments that result from the binding of a particular strand at any point along the gate struture, we can define a compilation function that produces all possible reactions between the strand and the gate, as shown in Figure 3b:

```
strandBindsToGate : strand -> gate -> reaction list
```

Note that these reactions will produce a new gate in which the incoming strand is just bound by the toehold, since this function only considers the binding reaction.

The unimolecular reactions are strand unbinding, branch migration and strand displacement. These can be enumerated similarly, using segment-level compilation functions that are then mapped across the gate structure.

```
strandUnbindings : segment -> (strand * segment) list
branchMignUpper : segment -> segment -> (segment * segment) option
strandDispUpper : segment -> segment -> (segment * segment) option
```

Strand unbinding reactions can be identified at the single segment level, but branch migration and strand displacement reactions occur across the boundary between two neighbouring segments, hence the `branchMignUpper` and `strandDispUpper` functions require two segments as arguments. Note that we have only presented signatures for functions to calculate branch migration and strand displacement reactions on the upper strand of a gate. This is because the set of possible DNA reactions is closed under a "mirror" operation which swaps the top and bottom strands, so we only need to define enumeration functions for the top strand and use mirroring to check for possible reactions on the bottom strand.

To enable modelling at various levels of detail it is desirable to compile reactions at different levels of abstraction, defined in [6]:

```
type absLevel = Detailed | Finite | Default | Infinite
```

We achieve this by categorizing the classes of reactions as *fast* or *slow*, depending on the desired level of abstraction. Strand binding is always a slow reaction, and any fast reactions that may occur after a slow reaction are simply merged with the slow reaction to produce a single reaction. In all but the most detailed levels of abstraction, *branch migration* reactions (reaction *(iii)* from Figure 1b) are assumed to happen so quickly that we use a structural congruence that identifies gates up to branch migration.

In addition, we have implemented reaction rules to enable two gates to interact end to end, forming linear heteropolymers that may be used to design DNA strand displacement stack machines [10, 11], and to allow modelling of "crosstalk" reactions to study failure modes of DNA circuits [6].

5 Discussion

While the use of domain-specific languages for formal modelling of biological processes is a well-established technique [12], the design of engineered biochemical systems can also benefit from domain-specific languages for specification and simulation. In addition to DSD, other such languages include GEC [13] and `gro` [14]. We believe that this is a fruitful new direction for research in programming languages.

Our experience developing the DSD compiler in F# convinced us that functional languages are an ideal implementation vehicle for this kind of domain-specific language, since in DNA nanotechnology, structure and function are closely linked. Base-level representations of DNA secondary structure based on strings [15] or numeric encodings [16] are often too detailed, meaning that some abstraction of structures into high-level features is typically required. Abstract data types provide a convenient means of representing DNA secondary structures at the level of domains, since each DNA structure is reflected in the structure of the abstract syntax tree of the corresponding value. Furthermore, the ability to pattern-match on these values and recurse over them allows concise definitions of the structure-function relationship.

The DSD language has been used to develop a number of DNA strand displacement systems [1, 2, 17]. In particular, the strand displacement digital logic circuit that was used to compute the square roots of four-bit binary numbers [1] consists of 74 initial species (a total of 130 DNA strands). This illustrates the scale of systems that can be modelled using the DSD language and constructed in the laboratory. Our implementation can be used online via the Visual DSD web server [18], which is accessible at `http://research.microsoft.com/dna/` with accompanying documentation.

References

1. L. Qian and E. Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332:1196–1201, 2011.
2. L. Qian, E. Winfree, and J. Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475:368–372, 2011.
3. M. N. Stojanovic and D. Stefanovic. A deoxyribozyme-based molecular automaton. *Nat. Biotechnol.*, 21(9):1069–1074, 2003.
4. Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, and E. Shapiro. An autonomous molecular computer for logical control of gene expression. *Nature*, 429:423–429, 2004.
5. M. R. Lakin, D. Parker, L. Cardelli, M. Kwiatkowska, and A. Phillips. Design and analysis of DNA strand displacement devices using probabilistic model checking. *JRS Interface*, 9(72):1470–1485, 2012.
6. M. R. Lakin, S. Youssef, L. Cardelli, and A. Phillips. Abstractions for DNA circuit design. *JRS Interface*, 9(68):470–486, 2012.
7. D. Y. Zhang and G. Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nat. Chem.*, 3(2):103–113, Feb 2011.
8. D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Springer, 2008.
9. D. Y. Zhang, A. J. Turberfield, B. Yurke, and E. Winfree. Engineering entropy-driven reactions and networks catalyzed by DNA. *Science*, 318:1121–1125, 2007.
10. L. Qian, D. Soloveichik, and E. Winfree. Efficient Turing-universal computation with DNA polymers. In Y. Sakakibara and Y. Mi, editors, *DNA16 Proceedings*, volume 6518 of *LNCS*, pages 123–140. Springer, 2011.
11. M. R. Lakin and A. Phillips. Modelling, simulating and verifying Turing-powerful strand displacement systems. In L. Cardelli and W. Shih, editors, *DNA17 Proceedings*, volume 6937 of *LNCS*, pages 130–144. Springer, 2011.
12. C. Priami, A. Regev, E. Shapiro, and W. Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, 2001.
13. M. Pedersen and A. Phillips. Towards programming languages for genetic engineering of living cells. *JRS Interface*, 6(Suppl. 4):S437–S450, 2009.
14. S. S. Jang, K. T. Oishi, R. G. Egbert, and E. Klavins. Specification and simulation of synthetic multicelled behaviors. *ACS Synthetic Biology*, 1:365–374, 2012.
15. P. Hogeweg and B. Hesper. Energy directed folding of RNA sequences. *Nucleic Acids Res.*, 12(1):67–74, 1984.
16. M. L. Fanning, J. Macdonald, and D. Stefanovic. ISO: numeric representation of nucleic acid form. In *Proceedings of ACM-BCB 2011*. ACM, 2011.
17. Y.-J. Chen, N. Dalchau, N. Srinivas, A. Phillips, L. Cardelli, D. Soloveichik, and G. Seelig. Programmable chemical controllers made from DNA. *Nat. Nanotechnol.*, 41(1):e33, 2013.
18. M. R. Lakin, S. Youssef, F. Polo, S. Emmott, and A. Phillips. Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics*, 27(22):3211–3213, 2011.