Reversible Bond Logic

Hannah Amelie Earley
Department of Applied Mathematics and Theoretical Physics, University of Cambridge, UK

Abstract

The field of molecular programming allows for the programming of the structure and behavior of matter at the molecular level, even to the point of encoding arbitrary computation. However, current approaches tend to be wasteful in terms of monomers, gate complexes, and free energy. In response, we present a novel abstract model of molecular programming, Reversible Bond Logic (RBL), which exploits the concepts of reversibility and reversible computing to help address these issues. RBL systems permit very general manipulations of arbitrarily complex "molecular" structures, and possess properties such as component reuse, modularity, compositionality. We will demonstrate the implementation of a common free-energy currency that can be shared across systems, initially using it to power a biased walker. Then we will introduce some basic motifs for the manipulation of structures, which will be used to implement such computational primitives as conditional branching, looping, and subroutines. Example programs will include logical negation, and addition and squaring of arbitrarily large numbers. As a consequence of reversibility, we will also obtain the inverse programs (subtraction and square-rooting) for free. Due to modularity, multiple instances of these computations can occur in parallel without cross-talk. Future work aims to further characterize RBL, and develop variants that may be amenable to experimental implementation.

2012 ACM Subject Classification Computer systems organization → Molecular computing; Theory of computation → Models of computation

Keywords and phrases Molecular Programming, Reversible Computing, Structural Manipulation

Digital Object Identifier 10.4230/LIPIcs.DNA.29.6

Acknowledgements The author would like to thank Gos Micklem, Jim Lathrop, William Poole, and three anonymous reviewers for their valuable comments and suggestions.

1 Introduction

Molecular programming is the powerful idea that we can program the very structure and behavior of matter at the molecular level. Not only can we build nanostructures of nearly arbitrary shape and functional properties, but we can design molecules whose interactions encode computation. Perhaps the earliest known example was able to compute solutions to NP-complete problems, in particular the Hamiltonian path problem [1]. Since then, implementations of models such as the Tile Assembly Model (TAM) and finite Chemical Reaction Networks (CRNs) have arisen [32, 28], leading to the possibility of general Turing-universal computation [27].

Versatile as these schemes are, for the most part our approaches to molecular computation are wasteful. For one, they tend to be one-shot [1, 23, 35]. Computations are set up in a state far from equilibrium, with free energy dispersed throughout the various species (typically complexes of DNA). Computation then typically corresponds to the process of equilibration of this system [26]. While this certainly provides a high driving force for the experiment, it has a number of drawbacks. In the case of many DNA-strand displacement systems, which typically consist of signal strands and gate complexes, many of the components cannot be reused; rather they are transformed into a plethora of waste complexes [35]. As the variety of waste complexes is large, it is non-trivial to selectively remove them and replace them with fresh gate complexes. Consequently, the runtime of computation is generally constrained by the initial concentrations of species. Moreover, the prospect of a long-running computational

system that responds to changing inputs over time – a feature that may be desirable in a smart therapeutic, for example – becomes impractical at best. Nevertheless, some work on renewable or long-running dynamic DNA-based systems is beginning to emerge [7, 10], as well as systems that reuse components [6]. As for Tile Assembly systems, free tile monomers are typically locked into a final assembly (except for transiently at the growing edge) [11]. Not only does this eventually starve future computation, but often these consumed tiles serve no ongoing functional purpose save that of storing the history of computation. Signal-passing Tile Assembly models [22] are capable of modifying the state of tiles after incorporation, such as to remove them, but these are irreversible changes.

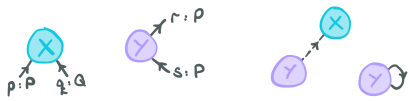
Is this waste – of free energy, monomers, and special complexes – unavoidable? Turning to life, the maestra of molecular machines, we see that this is not the case. Biochemical systems routinely recycle monomers – building up and breaking down macromolecules from their constituent components. Their molecular machines (enzymes), which we loosely identify with the gate complexes mentioned earlier, are in general fully reusable – acting catalytically. Lastly, free energy is not distributed casually across a large number of components. Rather, a select few species are designated as free energy currency. The prototypical example is that of ATP/ADP. This pair of chemical species can be converted between one another, so by maintaining a ratio of the two that is far from equilibrium, the hydrolysis of ATP into ADP can be used to store free energy. The missing ingredient is then to build molecular machines that couple a desired reaction to this hydrolysis, so the free energy can be supplied to other reactions. In this way, living systems need only inject fresh free energy into these few subsystems in order to continually sustain the operation of all other processes.

From this we draw one main conclusion for the design of more effective molecular systems. Free energy should be separated out from the molecular machines it powers. As a consequence, the molecular machines will generally be catalytic, returning to their original state after performing their function. This is best implemented by exploiting reversible dynamics, which is already a characteristic of the microscopic realm. In reversible dynamics, the previous state of a system (and indeed, its entire history) is uniquely determined by the current state. More strongly, we have time reversal symmetry: the laws of physics are the same both forwards and backwards. An interesting corollary of these reversible dynamics is that the system's evolution (at least at the local level) can be reversed by inverting the free energy supply. Additionally, the speed of evolution can be controlled by increasing or decreasing the amount of free energy stored. Of course, these are not unknown ideas to the molecular programming community. One of the earliest proposals for a molecular computer, arguably predating the field, is Bennett's enzymatic Turing machine [3]. Bennett's design consisted of a polymeric tape, bespoke enzymes performing reversible computational steps, and a pool of free energy currency and structural monomers. While not experimentally realized, other enzymatic approaches to molecular computation have been, such as the PEN toolbox [21] and PER [17]. As designing custom enzymes remains highly non-trivial, however, these approaches reuse naturally occurring enzymes such as DNA polymerase. Non-enzymatic approaches include the proposed DNA polymer stack machines of Qian et al. [23], the reversible surface CRNs of Brailovskaya et al. [4], and intricate chemomechanical systems [5, 24]. Furthermore, reversibility often features as a building block in other systems [12, 14, 16].

We present a novel abstract model of molecular programming, Reversible Bond Logic (RBL). RBL systems consist of "atoms", which can be combined by "bonds" into arbitrarily complex "molecules". By programming the energy landscape of the atom-bond configurations, reversible paths through configuration space can be carved. This will prove sufficient to implement a variety of systems, including catalytic molecular machines and a common free

$\mathcal{S} = (\{X, Y\}, \{P, Q\}, \{\text{solid}, \text{dashed}\}, \mathbb{R}_{\geq 0} \cup \{\infty\})$
$P = \{\text{solid}, \text{dashed}\} \qquad Q = \{\text{solid}\}$
$X = (\{p, q\}, (p : P, q : Q), (p : \text{in}, q : \text{in}), e_X)$
$Y = (\{r, s\}, (r : P, s : P), (r : \text{out}, s : \text{in}), e_Y)$

(a)



(b) **(c)**

Figure 1 An example RBL scheme, $\mathcal{S}$. **(a)** The formal definition of $\mathcal{S}$, which comprises two atom types, X and Y, two port types, P and Q, and two bond colors, solid and dashed. Energies may be any non-negative real or infinite. Each atom has two ports, and the energy configurations (e_X, e_Y) are left unspecified. **(b)** The RBL atoms defined in diagrammatic form. The label $p : P$ indicates that the port has label p and is of type P. **(c)** An example system configuration with one copy of X and two of Y. As we keep the same port positions as in (b), we omit port labels for brevity.

energy currency. Moreover, as RBL atoms are manipulated in a reversible fashion, they can be freely reused. Indeed, in RBL complex and diverse macromolecular structures can be built up and broken down as required. The informational content of these structures can then be exploited to build modular and compositional computational entities, allowing for rich computational primitives such as looping, recursion, and subroutines. The goal of RBL is to be a new platform for molecular programming, in particular it is hoped that it will enable the routine manipulation of complex structures and that it will make the design of energy efficient systems easier. As a corollary, RBL may also provide a new route for the implementation and study of reversible computing.

It is important to note that reversibility has profound implications on the very nature of computation, and so will affect how are our programs are written. Conventional approaches to computation make liberal use of non-invertible computational primitives, such as overwriting variables or (freely) merging branches of control flow. The most immediate consequence is that it is not generally possible to determine the previous state of a computer from the current state, as there are often many possible consistent histories. While beyond the scope of this paper, there is a deep and rich connection between irreversibility in computation and the thermodynamics of information [29, 19]. As we seek reversible dynamics, we will leverage the principles of reversible computing [3] and programming [20, 34][9, pp. 11–23]. Namely, we will avoid loss of information and take care to distinguish branches of control flow when they are merged.

The paper begins with a definition of RBL. Next, RBL is introduced more concretely with the implementation of a walker powered by an external fuel supply. Then, one possible scheme for computation is presented; a small selection of structural-manipulation primitives are introduced, and used to implement conditional branching, looping, and subroutines. Additionally, the computation is coupled to the same fuel supply introduced earlier, in order to drive the otherwise-unbiased reversible computation forward. The paper concludes with a discussion of the advantages and limitations of RBL. Elaboration of the computational primitives introduced and the design decisions behind them is presented in the technical appendix.

2 Definition

Informally, an RBL scheme consists of a set of atoms. These atoms are decorated with ports of certain types, and like ports can form bonds between one another – whether between ports on two different atoms or on the same atom (a self-loop). Ports additionally come

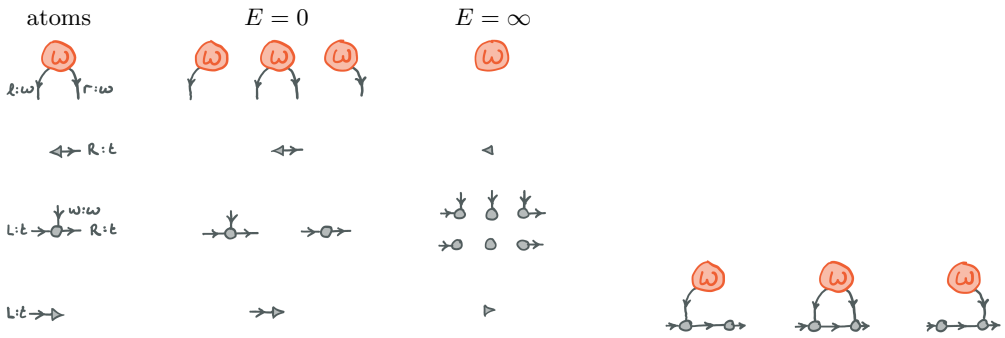
in "oriented" pairs: "in" and "out"; a bond is then formed *directionally* from an out-port to an in-port. A given port type may support bonds of multiple different "colors" (not necessarily literal colors), which can be useful for signal passing. The decoration of ports on an atom is geometry-free: there is no intrinsic ordering or positioning of ports, and they can be considered freely labile. There is some resemblance of RBL to Thermodynamic Binding Networks (TBNs) [8], which will be discussed in Section 5.

Key to RBL is the logic that dictates permissible bond formation and transitions. Each possible configuration of an atom – namely, whether each of its ports is bonded and, if so, with what color bond – is assigned an energy. As expected, higher energies are thermodynamically less likely with "$E = \infty$" representing an impossible configuration. In fact, in practice configurations are usually only assigned energies of $E = 0$ or $E = \infty$, i.e. possible or impossible, although other finite positive energies may be employed for short-lived transitional states. A system configuration – the configuration of multiple atoms – may transition to another configuration if and only if: (1) the configurations differ by a single bond (equivalently, by two port states), (2) the configuration energies are finite. That is, only one bond transition (whether formation, color change, or breakage) may occur at a given time. The kinetics may depend on the energy and associated entropy changes, e.g. due to a change in particle number, but we leave a detailed treatment of this to future work.

Designing an RBL scheme then amounts to carefully picking the energy landscapes of each atom so as to carve out a guided path(s) through configuration space. In general, we seek to restrict the system so that at any one time there are precisely two possible transitions: one to the previous state, and one to the next state. Sometimes, however, it may be desirable to provide parallel paths through configuration space when the order of operations is unimportant.

Although we will introduce RBL schemes diagramatically, we define RBL formally for completeness. An RBL scheme is specified by a tuple (A, P, B, E) where A is an indexed set of atoms, P an indexed set of ports, B a set of bond colors, and E a set of possible energies. The indices of A and P are used to uniquely label each type of atom and port. A port p consists of a subset of B, i.e. $P_p \subseteq B$, corresponding to the bond colors it can form. An atom of type α consists of a tuple $(J, \vec{p}, \vec{o}, e)$, where J is an index set enumerating the ports of the atom, $\vec{p}$ is a vector of port types indexed by J, $\vec{o}$ is a vector of port orientations indexed by J, and e is an energy function. An atom's energy function maps each possible configuration to its energy, i.e. $e : \prod_{\ell \in J}(\{\varepsilon\} \cup P_{p_\ell}) \to E$ where ε corresponds to an unbound port. An example scheme is shown in Figure 1.

A system configuration is a directed graph. Each atom of type α, with atom-tuple $(J, \vec{p}, \vec{o}, e)$, has an associated sub-graph. This sub-graph consists of an "atom" node labeled α and a set of "port" nodes: for each $j \in J$ we add an edge labeled j from the atom-node to the port-node, which is labeled (p_j, o_j). A configuration consists of a (disjoint) union of atom-graphs, where there may be any number $\in \mathbb{N}$ of copies of each type of atom-graph. Bonds correspond to edges from a node (p, out) to a node (p, in) for some port type p, and are labeled with some color in P_p. For each atom-sub-graph, we can compute its energy using the energy function e. The system configuration is valid if each of the energies of its constituent atoms is finite, and the total energy of the system is given by the sum of these. Two system configurations are *adjacent* if they are both valid and they differ by a single port-port edge; this difference can be the presence/absence of such an edge, or a change in label. A system may transition to any adjacent configuration.



(a) The RBL energy configuration of an unbiased walker.

(b) The dynamics of walking.

Figure 2 An RBL implementation of an unbiased walker. **(a)** The system consists of four types of atom. W represents the walker; it has two feet, given by the left (ℓ) and right (r) ports, which are both of type w and can form bonds of a single color, solid. The other atoms represent the track, and comprise a left-cap, track monomers, and a right-cap. The track monomers have a w port of type w, allowing the feet of the walker to bind to them. The other ports, L and R of type t (monochromatic solid), allow the track monomers to polymerize; if capped on the ends they will form a linear track, else a circular loop. The energy configuration is defined by assigning each possible state to $E = 0$ (allowed) or $E = \infty$ (impossible). The energy configuration of W allows a walker to bind to one or two track monomers, but it can never dissociate from the track as the unbound state is impossible. Meanwhile, the energy configuration of the track enforces that it must be complete and cannot fall apart. Optionally, the "transition" state where W binds to two track monomers can be assigned a higher energy, e.g. $E = 1$. **(b)** The energy configuration leads to the walking dynamics shown. The walker is free to step to the left or right, forming a transient state with two feet on the track. Then either foot can dissociate, possibly leading to net movement along the track.

3 Walkers and Fuel

A walker is a molecular machine that walks along a molecular track. In nature these are often referred to as motor proteins, and serve a vital role in cells performing such tasks as transporting cargo or contracting muscle cells. Molecular programmers have designed many instances of walkers. Some walkers have no requirement of directional movement and perform a random walk through their domain, such as the "cargo-sorting robot" of Thubagere et al. [30], or directionality is provided by cycling reagents [25]. Other walkers, such as that of Yin et al. [33], achieve uni-directional movement along a one dimensional track by using high-free energy fuel to permanently block previously-visited track footholds. This can be considered analogous to a Brownian ratchet. Such "burnt bridge" approaches to walking have the disadvantage that it is not possible to send multiple walkers down the same track, and the track needs to be rebuilt in order to walk along it again. Motor proteins do not modify their track. Instead, processive movement is conferred by supply of an external free energy currency. We will implement such a walker within RBL, starting with an unbiased walker that has no directional preference and then adding both polarity and an external fuel supply that couples directional movement to the free energy supply. While this is not the first directional walker that leaves its track intact [13], its implementation is particularly natural and yields a common free energy currency.

3.1 Unbiased Walker

An unbiased walker is relatively easy to implement, and an appropriate RBL scheme is shown in Figure 2. The walker itself is given by a single RBL atom, W, and it walks along a track using its two "feet" ports to make bonds as needed. All that is required for correct operation is for W to be able to bind to the track with one or both feet, but not none. In this way, it can make "progress" by putting down a second foot and lifting the first foot. In both the unbiased and biased walkers, detachment from the track is made impossible by assigning infinite energy to this state; in a realistic system, it may be desirable to permit controlled attachment and detachment. The techniques used in Section 4 and beyond could be used to do this. As the dynamics are reversible and history-free, there is of course no guarantee that it won't just lift the second foot and make no progress, nor is there any guarantee that it will make net progress over time. Indeed, the statistics of movement are described by an unbiased random walk with zero mean displacement after t steps, and $\sim \mathcal{O}(\sqrt{t})$ variance.

A brief commentary on the geometry of the walker atom W is warranted. If the ports of W had fixed position, then there might be a risk of the walker not moving beyond the initial pair of track monomers. Free rotation about the bonds or sufficient flexibility would fix this, however recall from the definition of RBL that the ports have no intrinsic position and are freely labile. As such, the ports are free to "move" as needed. A further consequence of being geometry-free is that a walker may "skip" an arbitrary number of positions along the track; there are a number of ways to address this, but we defer discussion of this to future work which will introduce a geometric model for RBL.

Also of note is the importance of the directionality of the bonds. Without directionality, the w ports of the track monomers would be able to bind to each other, as would the ℓ and r feet of the walker W. Therefore directionality enforces a certain complementarity relation, which is not unfamiliar in the world of DNA nanostructures.

3.2 Fuel

In order to build a powered directional walker, we will need an external fuel supply. Inspired by biochemistry, we choose to store our free energy in the population of two related species. For simplicity, we design (Figure 3) a bistable atom G with two distinct states, G_+ and G_-, that can be interconverted by an external signal at the coupling port f. Consequently, the equilibrium state corresponds to $[\mathsf{G}_+] = [\mathsf{G}_-]$ and free energy may be stored by increasing the ratio $[\mathsf{G}_+] : [\mathsf{G}_-]$ above 1. Specifically, the amount of free energy is given simply by $kT \log([\mathsf{G}_+]/[\mathsf{G}_-])$. For another system X to usefully access this free energy, it needs to couple carefully to the fuel species. The initial state of the system $X_{\text{init.}}$ must couple to the G_+ state, and the final state of the system $X_{\text{fin.}}$ must couple to the G_- state. That is, we need to implement the "reaction" $X_{\text{init.}} + \mathsf{G}_+ \rightleftharpoons X_{\text{fin.}} + \mathsf{G}_-$. This is the purpose of the colorings available on the f port, and will be shown more concretely in the next subsection.

Critical to the design is the double self-loop on G. A single self-loop would be able to autonomously transition between bond colors due to the rules of RBL, but a double self-loop allows us to impose an (infinite) energy barrier between the G_+ and G_- states. This barrier is lowered by the action of the coupling port, f, which thus acts as a catalyst for the interconversion of states.

3.3 Biased Walker

To construct a walker with a preferred walking direction, we must introduce polarity into both the track and the walker. In nature, a single monomer type (e.g. actin) is sufficient for this by taking advantage of its spatial substructure. In RBL, however, such spatial

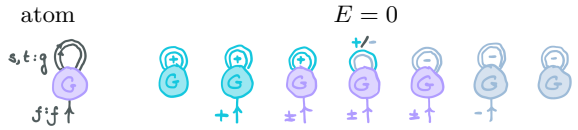


Figure 3 An RBL implementation of a bistable fuel species, G. The atom has five ports, s_1/s_2, t_1/t_2, and f. The ports s_1/s_2 form a self-loop, and so we refer to them together as s; similarly for the self-loop t. The loops are each of type g, which has two colors: $+$ (bright blue) and $-$ (gray-blue). The "coupling" port f is of type f and has three colors: $+$, $\pm$ (lilac), and $-$. The two stable states of G correspond to both loops being $+$ (G_+) or both being $-$ (G_-). These can be interconverted via the coupling port, f, passing through a transitional state $\mathsf{G}_\pm$ where s is colored $+$ and t is colored $-$. The $+$ and $-$ colorings allow another RBL atom to distinguish G_+ from G_-, whilst the $\pm$ coloring allows interconversion. If there is more G_+ than G_- then there will be a negative (favorable) free energy change ΔG associated with the reaction $\mathsf{G}_+ \to \mathsf{G}_-$. Note that we only show the possible states ($E = 0$); all other configurations may be presumed impossible ($E = \infty$).

substructure is non-existent. Instead we employ an alternating sequence of monomers. Two monomers, i.e. a sequence $\cdots\alpha\beta\alpha\beta\cdots$, would be insufficient to distinguish forward from backward movement; the minimum sequence to introduce polarity consists of three monomers, i.e. $\cdots\alpha\beta\gamma\alpha\beta\gamma\cdots$. With such a polarized track, the walker can then be modified such that, from an initial position on monomer α, the forward step takes it to β and the backward step to γ. Similar behavior applies to the other track positions. Of course, without a fuel supply forward and backward steps are identical and so movement is still non-directional. To complete the implementation, we couple these steps to consumption of fuel as follows:

$$\mathsf{W}_\alpha + \mathsf{G}_+ \rightleftharpoons \mathsf{W}_\beta + \mathsf{G}_- \qquad \mathsf{W}_\beta + \mathsf{G}_+ \rightleftharpoons \mathsf{W}_\gamma + \mathsf{G}_- \qquad \mathsf{W}_\gamma + \mathsf{G}_+ \rightleftharpoons \mathsf{W}_\alpha + \mathsf{G}_- \tag{1}$$

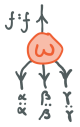
There are many possible designs, but we choose a three-footed walker with a fuel coupling port f. Each foot is specific to a particular monomer type, i.e. α, β, or γ, and the dynamics of "walking" resemble a wheel rolling along the track. This design is elaborated in Figure 4. The main challenge in designing such an RBL system is to ensure that correct system configurations cannot jump to invalid configurations. For example, if W did not maintain a foothold on both α and β during an $\alpha \to \beta$ step, then it would be possible to "jump" into either of the other step processes. To assist in this, a computational suite (to be released in the future) for designing and testing RBL schemes was developed. To finish, we prove (Theorem 1, Appendix A) that the designed scheme has the desired properties:

▶ **Theorem 1.** *The dynamics of the biased walker, in the long-run, are that of a biased random walk.*

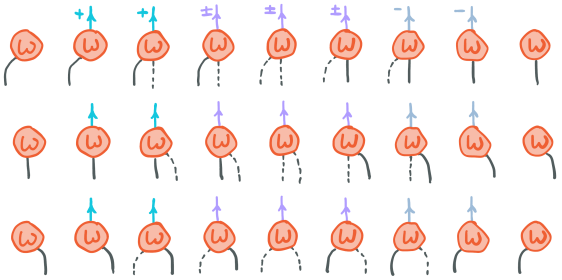
4 Data and Computation

The advantages of structured data and programming abstractions are well known to users of high-level programming languages. In this section, we will develop a set of conventions and motifs for representing and computing with structured data in RBL. We will use these to implement three example "programs": (1) logical negation of a Boolean value, with which we will introduce **conditional branching**; (2) addition of natural numbers (using a Peano representation), with which we will introduce **looping**; (3) squaring of natural numbers, using addition as a **subroutine**. A recursive implementation of addition is left as an exercise for the reader. High level schemata to motivate these implementations are presented in Figure 5.

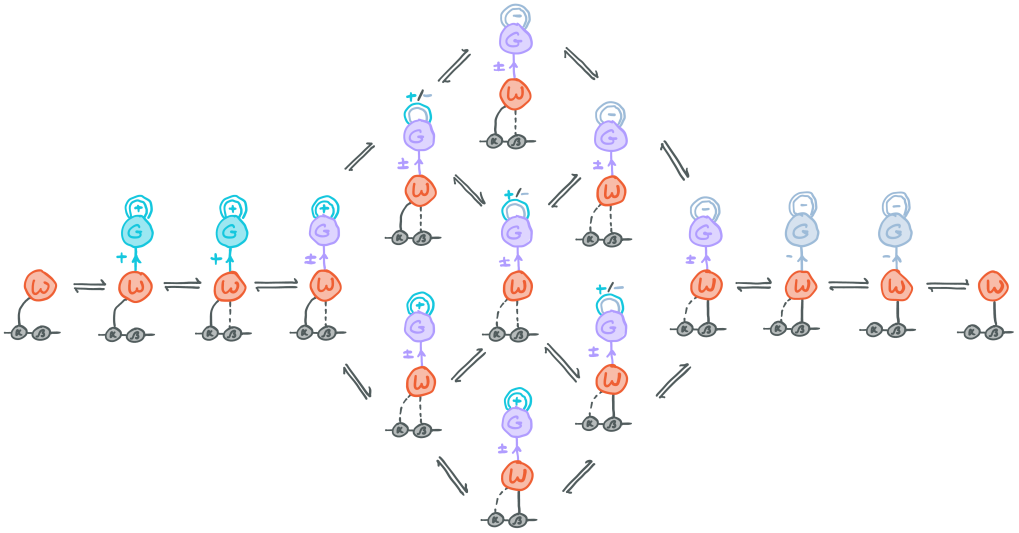
(a) The track.



(b) The walker, W.



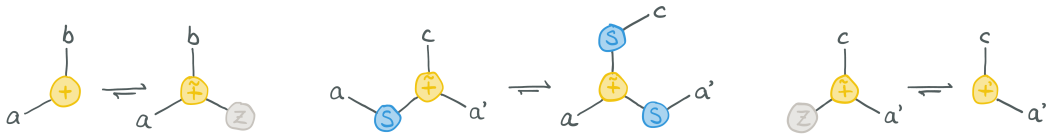
(c) The permissible energy configurations of W ($E = 0$).



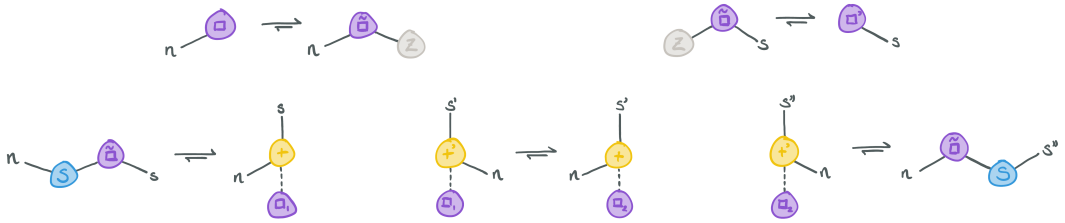
(d) The transition diagram for an $\alpha \to \beta$ step, i.e. $\mathsf{W}_\alpha + \mathsf{G}_+ \rightleftharpoons \mathsf{W}_\beta + \mathsf{G}_-$.

Figure 4 An RBL implementation of a biased walker. **(a)** The track consists of a polymer of α, β, and γ track monomers. Their implementation is nearly identical to that of the track in Figure 2a, except that: (1) the L/R ports enforce the $\cdots\alpha\beta\gamma\alpha\cdots$ sequence, by R_α/L_β, R_β/L_γ, and R_γ/L_α each having a distinct port type; and (2) each monomer's foothold port has a distinct type (α, β, or γ). Another difference is that the foothold ports have two bond colors: "solid" and "dashed", with the latter corresponding to a transitional state. The track can again be linear or cyclic. **(b)** The walker atom W has three feet which specifically bind each type of track monomer. It also has a fuel coupling port f. **(c)** The set of permissible energy configurations of W implements each of the "reactions" in Equation (1), one per row. Considering the top row, we start with W bound to an α track monomer. If it then binds to G_+, it can begin to attempt an $\alpha \to \beta$ step (looking to the end of the third row, if it instead bound G_- then it would begin to attempt an $\alpha \to \gamma$ backstep). From here, it tentatively (dashed bond color) places a foot on the β monomer. Then it changes the color of f to $\pm$ (lilac), the transitional state. While the fuel is interconverting between G_+ and G_-, it changes its α binding to dashed and its β binding to solid in two steps. Then, it attempts to change the color of f to $-$ (gray-blue), which would indicate the successful consumption of fuel ($\mathsf{G}_+ \to \mathsf{G}_-$). It can then dissociate its α foot followed by the spent fuel G_-. While we have described this step as if W has "intent", it is important to remember that this is only for narrative benefit; the system is really performing a random walk through configuration space, with bias provided by the free energy stored in the concentration of the fuel. **(d)** This $\alpha \to \beta$ step process is expounded in the transition diagram, which shows which system configurations can be reached from which other system configurations. This makes clear that the consumption of fuel and transition of foothold states occur in parallel.

(a) Logical negation. A Boolean value, T or F, is tagged with an atom (¬) which serves as a "reference" to the logical negation "function". Then, two "reactions" are implemented recognizing each of the possible inputs. The return value is tagged with ($\neg'$) to represent the result of the logical negation.

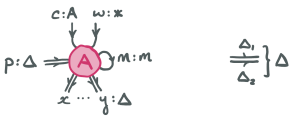


(b) Addition. As with logical negation, we use two special atoms to represent an addition to be performed, (+), and the result of an addition, ($+'$). As the computation must be reversible, we retain one of the inputs: for example, after adding $a = 3$ and $b = 4$, we would return $c = 7$ and $a' = 3$. The inputs and outputs are represented as Peano numbers, i.e. polymers of the form $\mathsf{S}-\mathsf{S}-\cdots-\mathsf{Z}$ where the number of S atoms corresponds to the number. For example, $3 \equiv \mathsf{S}-\mathsf{S}-\mathsf{S}-\mathsf{Z}$. The core of the computation is a loop (middle reaction, tagged by ($\tilde{+}$)), which iteratively decrements the bottom-left number and increments the top and bottom-right numbers. To see that this reaction functions as a loop, notice that the product can serve as a reactant to the same reaction while the loop condition holds. Note also how any possible molecule can participate in at most two reactions (corresponding to a forward and backward transition), giving a deterministic path from input to output. As the bottom-left number is originally a and the top number is originally b, these become respectively 0 and $c = a + b$. The bottom-right number starts at 0 and becomes $a' = a$. The left reaction is responsible for entering the loop, and the right reaction for exiting the loop.

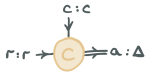


(c) Squaring. A number n can be squared by using the identity $n^2 = (2n-1) + (2n-3) + \cdots + 3 + 1$: that is, n^2 is the sum of the first n odd numbers. The implementation of squaring uses (□) and ($\square'$) as initial and final tags. It takes a Peano number n as input and returns a Peano number $s = n^2$ as output. We implement squaring using a loop (tagged by ($\tilde{\square}$)) with two variables, n and s. Initially, n is the number to be squared and $s = 0$. On each iteration, we decrement n, add this to s twice, and then increment s; the net result is $s \mapsto s + (2n-1)$ and $n \mapsto n - 1$. The addition is performed by using the program in Figure 5b as a subroutine. By starting with the largest odd number and ending at 1, we ensure that we consume the value n (by reducing it to 0) and produce only $s = n^2$ as an output, even though addition always returns one of its inputs.

Figure 5 High level schematic representations of three simple programs. These are not themselves RBL systems, but will be used as a blueprint for the construction of equivalent RBL systems. Biased arrows indicate the preferred direction for forward computation.



(a) The general form of a data-atom and data-port (Δ). Its c and w ports correspond to the specific and wildcard control ports respectively, and the m port pair corresponds to the "monomer" self-loop. The p data-port binds the data-atoms parent, and $x \cdots y$ provide bindings to some number of children.



(b) The C atom. The a data-port binds some computational data, the r port binds its root or origin, and the c port may be bound by some compuzyme.

Figure 6 The RBL atoms for data-atoms and the C atom. The energy configurations and dynamics are explored in Appendix B.

4.1 Motifs

Recall that we intend for the computational systems we design to be modular, compositional, and to support component reuse. To realize these properties, we will require a common convention for the representation of data. Moreover, as this data may be reused in different contexts (for example, Peano numbers may be added or multiplied), the data itself should generally be inert. Computation will be performed by dedicated catalytic "machines", taking inspiration from biochemical systems. We will call these "compuzymes" (computational enzymes). Consider the schematic for addition (Figure 5b). In principle, we will be able to construct RBL molecules corresponding to each of the three types of abstract molecule in the scheme, and we can also construct a compuzyme implementing each of the three reactions. To demonstrate the power of RBL, however, we will construct a single compuzyme performing the entire addition (loop included).

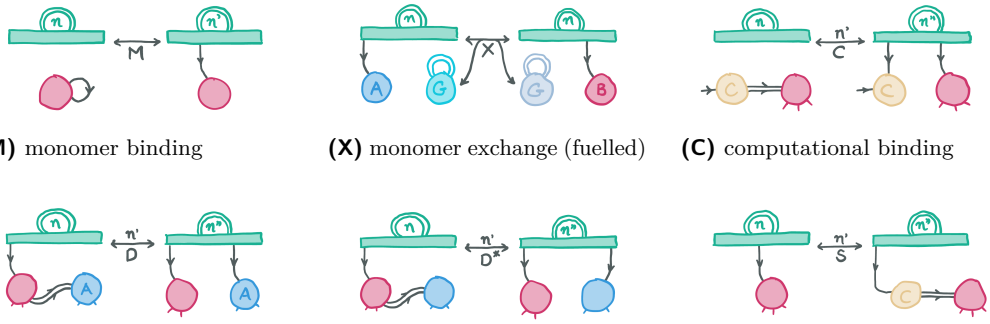
Data will be constructed from "data-atoms", RBL atoms with a certain structure. Consider Peano numbers, where a number is either zero (Z) or the successor (S) of another number. In haskell, this may be represented as `data` $\mathbb{N}$ `= Z | S` $\mathbb{N}$. We can therefore see that the S atom has one "child" while the terminal Z atom has no children. For an example of a data-atom with more than one child, consider the nodes of a binary tree[1]. These data-atoms would have three children, two for its child nodes (or leaves), and one for its associated data. As all data-atoms could be the child of another atom, each data-atom also has a parent. Therefore, an RBL representation of a data-atom must have an in-port for a parent to bind, and an out-port for each of its children; we refer to these special ports as "data-ports". These ports will in fact be pairs of ports: as with the double self-loop on the fuel atom G, a pair of bonds can be used to provide an energy barrier against unintentional bond-breakage.

In addition to parent and child data-ports, a data-atom needs a few other ports. To manipulate data, we add two control ports. These allow compuzymes to externally signal data-atoms to alter their configuration. One control port is specific to the type of data-atom; for example, the control port of an S atom is distinct from that of a Z atom. The other control port is a "wildcard" control port that is common to all data-atoms, and enables (limited) manipulation of variable data. Lastly, we have a pair of ports that can form a self-loop. This self-loop is present only on free monomers. The reason why this is required is somewhat subtle, and is explained in Appendix B.

The final key component of data is the C atom. Some data represent computations (either to be performed or that have completed), such as the molecules in Figure 5. These data are capped by a C atom, e.g. C–(¬)–T. The C atom therefore identifies computational data. It may seem extraneous, but it provides important indirection for subcomputations by allowing two compuzymes to interact with the same computational data simultaneously. As such, C has a "root" port (r) bound by the computation's origin, a "compuzyme" port (c) optionally bound by a compuzyme, and a data-port (a) binding the actual computational data.

The RBL atoms for data-atoms and the C atom are shown in Figure 6. These data structures can then be operated on by "compuzymes" to perform computation. Compuzymes are special atoms that behave as a platform for structural manipulation, and are typically represented as a rectangle. By recruiting data-atoms and interacting with them via their control ports, a sequence of structural manipulations can be effected. Compuzymes also maintain an internal state, via the color of a pair of self-loops. These state colors are typically numbered, but any label will suffice. Internal states are useful for distinguishing between

[1] Recall that the definition of a binary tree in haskell is `data Tr` a `= Lf | Nd (Tr` a`)` a `(Tr` a`)`.



(M) monomer binding **(X)** monomer exchange (fuelled) **(C)** computational binding

(D) destructuring **(D*)** wildcard destructuring **(S)** subcomputation initiation

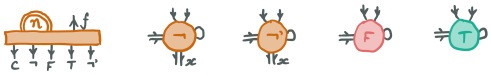
Figure 7 The five core motifs underlying our computational convention. Only the net effect of the motif is shown; the detailed RBL implementations are given in Appendix B. The green rectangle is a compuzyme, and the encircled value n corresponds to the internal compuzyme state. In all motifs except X, there are one or two compuzyme state changes (from n to n', and possibly to n''). **(M)** The binding of a free monomer (pink data-atom). **(X)** The exchange of a monomeric data-atom A for B. **(C)** The binding of some computational data (pink data-atom, possibly with children). **(D,D*)** The destructuring of a child (blue data-atom) from its parent (pink data-atom). In the first case, the identity of the child data-atom (A) is known by the compuzyme, while in the second case it is not. **(S)** The initiation of a subcomputation. The pink data-atom (and children) represents some pre-prepared computational data. It is bound to a fresh C atom, exposing it to the action of other compuzymes.

similar configurations, such as may occur during branches, loops, or long sequences. There are five core compuzyme motifs that are needed to perform arbitrary manipulations. These are shown in Figure 7, and their detailed implementations may be found in Appendix B, along with a description of the bond colorings of the data-ports and control ports. As RBL is reversible, each of these motifs may also act in reverse; for example, Motif D may also be used to bind a child data-atom to a parent. By convention, Motif X also expends one unit of fuel to drive computation forward, but in principle fuel coupling can occur at any point(s) during compuzyme operation.

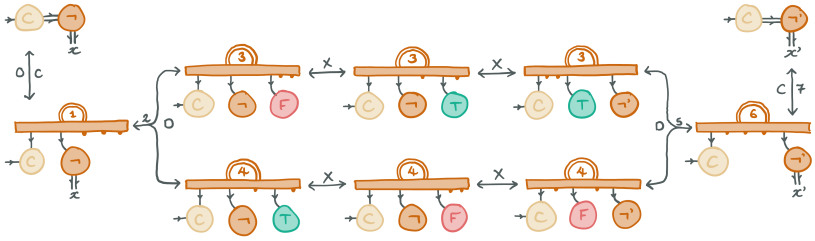
To construct a compuzyme in RBL, we should first prepare an (abbreviated) transition diagram of its operation using the above motifs. Having done this, we will be able to determine what ports the compuzyme will require. With this, a suitable RBL atom can be designed. Then, the RBL implementations of each motif (shown in Appendix B) dictate which configurations are possible. For the most part, each motif will correspond to a distinct compuzyme state, and so the sets of possible configurations will generally be disjoint. We need merely extend the configurations to include the state of the other compuzyme bonds, which will be static within the motif. In some cases, particularly with branching control flow, motifs may share configurations. Thus the complete RBL description of the compuzyme is simply the union of all the configurations of the motifs.

4.2 Logical Negation

With these motifs we are now in a position to implement our three example programs. One approach to implement logical negation is to prepare two compuzymes performing each of the two "reactions" in Figure 5a, $\mathsf{Not_T}$ and $\mathsf{Not_F}$ respectively. The compuzyme implementing $\neg\mathsf{T}$ would then use Motif C to bind a $(\neg)$ computational data-atom, and then Motif D specialized to T. It can then use Motif X twice to swap T for F and $(\neg)$ for $(\neg')$, followed



(a) The RBL atoms for logical negation. From left to right, the compuzyme Not, the computational data-atoms (¬) and (¬′), and the data-atoms for Boolean values, F and T. Not has ports for fuel (f), atom C (C), and the control ports of each of the data-atom types; port labels and types are the same.



(b) The abbreviated transition diagram for logical negation, showing the motifs employed. The compuzyme Not starts and ends in its pure unbonded atomic state, and so acts catalytically. The f port is not shown, and unbound ports are shown with a dot. To aid comprehension, port order is consistent with that shown in (a).

Figure 8 The implementation of logical negation using conditional branching.

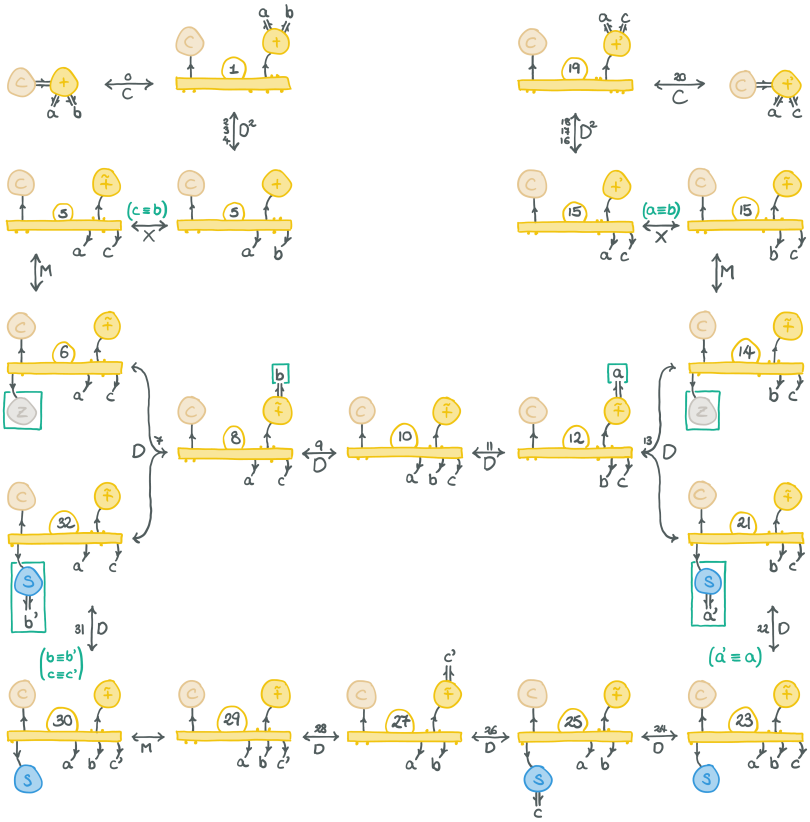
by Motif D to bind F to (¬′), and finally Motif C to eject the result of the computation. The compuzyme for ¬F would be very similar. Of course, it is possible that compuzyme Not_T might inadvertently bind C–(¬)–F, but the specificity of Motif D would mean the compuzyme stalls. The only available option would be for it to backtrack; only the second compuzyme can complete the computation in this case. Moreover, only once the correct input to the compuzyme is bound can it expend fuel.

However, RBL allows us to go beyond this and implement conditional branches in control flow without backtracking. That is, we can create a single compuzyme Not implementing both cases. We can overlap two (or more) instances of Motif D, each recognizing different data-atoms. The motifs would use the same states n and n', but differing states n'' for each case; these distinct states n'' then allow us to distinguish the different branches of control flow. This is shown in Figure 8. Indeed, the dynamics of Not are simply the conjunction of those for Not_T and Not_F. Note that, when the two branches of control flow are later merged, this is a reversible operation: the two branches have to be *clearly* distinct. Here this is the case because we are using Motif D against two cases, T and F, "combining" the data-atoms into a single variable data-atom x'.

4.3 Addition

Similarly, we could implement addition using three compuzymes as indicated in Figure 5b. However, we again leverage the power of RBL to show how looping can be implemented directly with a single compuzyme Add. This implementation is given in Figure 9. Note that in this condensed implementation, the atom ($\tilde{+}$) now just serves to hold temporary variables and so only has one child compared to three in our original schematic. Looping is not much different from the control flow for logical negation; whereas for conditional branching, we had a branch followed by a merge, here we have a merge followed by a branch. At the beginning of a loop, we merge control flow from two branches: entry and loop-continuation. This would usually be trivial, but in a reversible context we need to be careful to conditionally distinguish these two events (so that, in reverse, we know when to "un-start" the loop). At the end of a loop, we branch into exit and loop-continuation.

(a) The RBL atoms for addition. From left to right, the compuzyme Add, the computational data-atoms $(+)$, $(\tilde{+})$, and $(+')$, and the data-atoms for Peano numbers, S and Z. Add has ports for fuel (f, not shown), atom C, and control ports for each type of data-atom, but also wildcard control ports a, b, and c for holding variable values during computation; port labels and types are the same, except for the variables which are of type $*$.



(b) The abbreviated transition diagram for addition, showing the motifs employed. Sometimes, two Motif Ds are elided into a single transition (shown as D^2). Green annotations show where variables are renamed between transitions or where two alternative data-atoms are bound to a variable position. To aid comprehension, port order is consistent with that shown in (a) except that atoms above the compuzyme are flipped vertically; additionally, unbound ports are shown with a dot, and the compuzyme state is shown with a single circle. Addition is implemented as a loop where the value of a is added to both b (renamed to c, and yielding $c = a + b$) and 0 (yielding a copy of a). The top left set of transitions enter the loop. The center set of transitions implement the control flow of the loop: on the left, they merge branches corresponding to loop entry and loop continuation, using b to discriminate these branches; on the right, they branch into the exit path or the main loop body, using a to discriminate these branches; the middle transitions tidy up/set up these branch operations. The bottom set of transitions implement the loop body, decrementing a and incrementing both b (now representing the copy of a) and c.

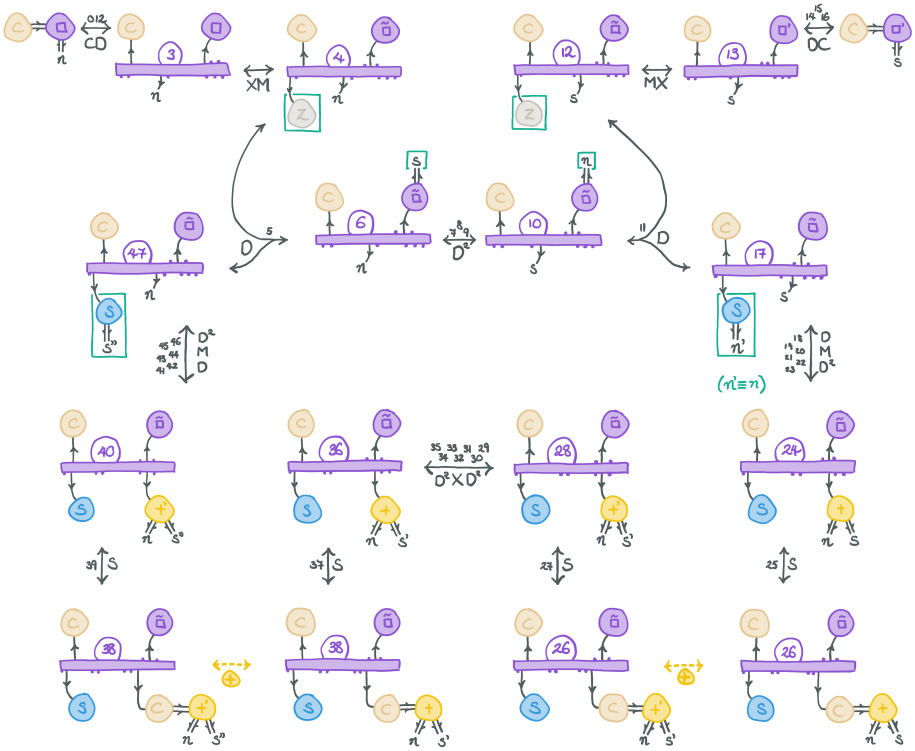
Figure 9 The implementation of addition using looping.

In our implementation of addition, we start with (a, b) and end with (a, c) where $c = a + b$. We begin by renaming b to c, to which we will add the value of a. Then we set the now free variable b to 0, to which we will also add the value of a. In our loop, the entry condition is therefore that the accumulator (b) is 0; if it is greater, then we have performed at least one iteration. The exit condition is that $a = 0$, as we decrement it on each iteration. Therefore, we merely need to use Motif D to check whether these variables are Z or S$-n$.

Due to reversibility, this program can also perform subtractions. To achieve this, one could either reverse the bias of the free energy currency, duplicate the implementation with reversed fuel coupling, or introduce a "switch" for the direction of computation.

4.4 Squaring

(a) The RBL atoms for squaring. From left to right, the compuzyme Sq and the computational data-atoms (□), (□̃), and (□′). Sq has ports for fuel (f, not shown), atom C, and control ports for the computational data-atoms shown, Peano data-atoms, two variables (n and s), as well as a set of ports for the addition subroutine: the computational data-atoms (+) and (+′) and the atom C; port labels and types are the same, except for the variables which are of type $*$.



(b) The abbreviated transition diagram for squaring, showing the motifs employed. Some motifs are elided into a single transition for brevity. Green annotations show where variables are renamed between transitions or where two alternative data-atoms are bound to a variable position. To aid comprehension, port order is consistent with that shown in (a) except that atoms above the compuzyme are flipped vertically; additionally, unbound ports are shown with a dot, and the compuzyme state is shown with a single circle. The implementation of squaring consists of a loop which on each iteration maps $s \mapsto s + (2n - 1)$ and $n \mapsto n - 1$, with s initially 0 and n ending at 0. The loop structure is essentially the same as for addition (Figure 9). To perform the map $s \mapsto s + (2n - 1)$, we first decrement n, then add the new value of n to s twice before incrementing it. The additions are performed using Add by preparing computational data representing the two additions.

Figure 10 The implementation of squaring using subroutines.

Finally, we implement squaring. Again, squaring consists of a loop; however, this loop involves calling addition as a subroutine. This is quite simply done. We use the data manipulation motifs to bind the numbers we wish to add to a (+) atom; then Motif S binds this computational data to a C atom, thus "presenting" it to Add compuzymes. The implementation is shown in Figure 10.

As with addition, the reverse of this program is also useful and performs square roots (over a suitably restricted domain – attempting to take the square root of 10 would lead to something resembling a runtime error).

4.5 Reversible Turing Machines

The implementation of reversible Turing Machines (defined by Bennett [2]) is left as an exercise for the reader. It only requires conditional branching, but the reversible implementation of a bi-infinite tape requires some care. For a hint, a reference implementation in a related language is given in the author's thesis [9] (p. 147).

5 Discussion

As required, the RBL model admits a variety of forms of component reuse: monomers used in the construction of molecules can be fully recycled, being drawn from and returned to a pool of free components; dynamic components can be designed to act catalytically, and so may be reused multiple times; and the same monomers (e.g. those representing Peano numbers) can be reused in distinct contexts without interference. These properties are strengthened by the modularity conferred by molecular structure: any number of instances of the same program or system can run in parallel using the same components and without crosstalk; and common sub-systems can be factored out and used by multiple distinct systems, such as a single common fuel species supplying all the free energy in the system. Moreover, RBL is particularly amenable to implementing hierarchies of abstraction. For instance, we saw the following hierarchy of computational abstractions: the definition of RBL "data-atoms"; a set of basic structural-manipulation motifs; control flow primitives such as branching and looping; and subroutines such as addition, which could then be used as a black-box by the squaring routine. Further abstractions which are possible but not shown include recursion and concurrency.

While RBL is a powerful model, it also has some limitations in its current formulation. Simulations of the squaring routine encounter 1700 distinct configurations when computing 3^2 (alternatively, $\sqrt{9}$). This is a reflection of the hierarchy of abstractions used; the structural manipulation motifs incur a cost of $\sim$ 10–15 transitions each, and multiple motifs must be employed to perform a given structural manipulation leading to an overhead on the order of 50–100 transitions for each useful computational step. Simpler "machines" such as the biased walker lie closer to "pure" RBL and incur 8–10 transitions per step (counting the shortest path in Figure 4d, and depending on whether we count the parallel transitions separately), though this is still perhaps larger than desired. Arguably the most significant reason for these overheads originates in the lack of spatial/geometric awareness of RBL. In biochemical systems, enzymes are able to take advantage of shape complementarity and differently-shaped conformations to perform their manipulations. This description omits the series of microstate transitions involved in an enzymatic reaction, but even so enzymes are (usually) particularly fast and efficient. Meanwhile in RBL we must very carefully coordinate structural manipulations through the use of, e.g., control ports: our lack of spatial awareness means that, without such coordination, we could not selectively displace a particular child data-atom from its parent. This suggests an obvious RBL variant to pursue in the future: one in which geometry plays a first-class role. Such a model should be able to more directly perform desired structural manipulations. Moreover, incorporating geometry would ensure that our designs are physically possible: the non-geometric RBL model can easily encode structures that are too crowded for Euclidean space, such as a complete binary tree of depth 20.

Another challenge for the experimental realization of RBL is that the number of configurations of an RBL atoms scales exponentially with the number of ports, and each of these may have a distinct energy. Restricting the set of allowed energies E to $\{0, \infty\}$ would considerably simplify this, as we may be able to get away with just programming the allowed configurations. Nevertheless, compuzymes such as Add and Sq have on the order of 200 allowed configurations. Furthermore, control ports and compuzyme states involve dozens of possible bond colors. Future work should investigate whether the number of required bond colors can be reduced, or whether bond colors are even required (i.e. whether ports can be monochromatic). Another question is whether atom configurations can be "factored" into simpler subsystems. Additionally, it would be useful to know what the minimum required complexity is to implement useful computational abstractions. For example, while Add may require ~ 200 configurations, the division into three compuzymes $\mathsf{Add}_{\text{init.}}$, $\mathsf{Add}_{\text{loop}}$, and $\mathsf{Add}_{\text{fin.}}$ may be substantially simpler. Combining with a geometric variant of RBL should also reduce the number of distinct configurations required.

Lastly, a more accurate treatment of the kinetics of RBL is warranted. Not only will configuration energy contribute to transition rates, but also any associated entropy changes. These entropy changes are particularly significant when particle number changes, such as when a compuzyme binds some computational data. There are also entropy changes associated with interactions with the monomer pools that cannot be eliminated through any "clever" means (see the author's thesis [9] (pp. 101–120)).

There is some similarity between RBL and Thermodynamic Binding Networks (TBNs) [8]. TBNs consist of monomers with a geometry-free collection of domains, similar to RBL atoms with a geometry-free collection of ports. Domains have complementary codomains, and these can form bonds. However, the TBN model explicitly does not consider the kinetics of system evolution. Instead it focuses on analyzing the possible stable configurations admitted by a given TBN to determine whether leak reactions are possible, and presumes that the desired state corresponds to thermodynamic equilibrium. In this way, the goals of TBNs diverge from the goal of RBL. In RBL, it is the programming of kinetic pathways that is important, and equilibration is to be avoided. Nevertheless, a realistic implementation of RBL would likely be subject to error conditions such as leak reactions, and so it would be interesting to use a TBN-like framework to evaluate or improve the robustness of RBL systems. Other related systems worth comparing in the future include Polymer Reaction Networks [15] and graph-theoretic molecular systems [18, 31].

In conclusion, RBL is an interesting new model with powerful properties, but it is also currently too unwieldy for experimental realization. Future work will further develop RBL, including variant models, to determine the possibility of achieving these powerful properties in a real chemical system.

References

1 Leonard M Adleman. Molecular computation of solutions to combinatorial problems. *science*, 266(5187):1021–1024, 1994.

2 Charles H Bennett. Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532, 1973.

3 Charles H Bennett. The thermodynamics of computation—a review. *International Journal of Theoretical Physics*, 21:905–940, 1982.

4 Tatiana Brailovskaya, Gokul Gowri, Sean Yu, and Erik Winfree. Reversible computation using swap reactions on a surface. In *DNA Computing and Molecular Programming: 25th International Conference, DNA 25, Seattle, WA, USA, August 5–9, 2019, Proceedings 25*, pages 174–196. Springer, 2019.

5 Rory A Brittain, Nick S Jones, and Thomas E Ouldridge. What would it take to build a thermodynamically reversible universal turing machine? computational and thermodynamic constraints in a molecular design. *arXiv preprint arXiv:2102.03388*, 2021.

6 Anne Condon, Alan J Hu, Ján Maňuch, and Chris Thachuk. Less haste, less waste: on recycling and its limits in strand displacement systems. *Interface Focus*, 2(4):512–521, 2012.

7 Erica Del Grosso, Elisa Franco, Leonard J Prins, and Francesco Ricci. Dissipative DNA nanotechnology. *Nature Chemistry*, 14(6):600–613, 2022.

8 David Doty, Trent A Rogers, David Soloveichik, Chris Thachuk, and Damien Woods. Thermodynamic binding networks. In *DNA Computing and Molecular Programming: 23rd International Conference, DNA 23, Austin, TX, USA, September 24–28, 2017, Proceedings 23*, pages 249–266. Springer, 2017.

9 Hannah A Earley. *On the performance and programming of reversible molecular computers*. PhD thesis, University of Cambridge, 2021.

10 Abeer Eshra, Shalin Shah, Tianqi Song, and John Reif. Renewable DNA hairpin-based logic circuits. *IEEE Transactions on Nanotechnology*, 18:252–259, 2019.

11 Constantine G Evans and Erik Winfree. Physical principles for DNA tile self-assembly. *Chemical Society Reviews*, 46(12):3808–3829, 2017.

12 Anthony J Genot, Jonathan Bath, and Andrew J Turberfield. Reversible logic circuits made of DNA. *Journal of the American Chemical Society*, 133(50):20080–20083, 2011.

13 SJ Green, Jonathan Bath, and AJ Turberfield. Coordinated chemomechanical cycles: a mechanism for autonomous molecular motion. *Physical review letters*, 101(23):238101, 2008.

14 Hope A Johnson and Lulu Qian. Simplifying chemical reaction network implementations with two-stranded DNA building blocks. In *26th International Conference on DNA Computing and Molecular Programming (DNA 26)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

15 Hope A Johnson and Erik Winfree. Verifying polymer reaction networks using bisimulation. *Theoretical Computer Science*, 843:84–114, 2020.

16 Hope Amber Johnson and Anne Condon. A coupled reconfiguration mechanism for single-stranded DNA strand displacement systems. In *28th International Conference on DNA Computing and Molecular Programming (DNA 28)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

17 Jocelyn Y Kishi, Thomas E Schaus, Nikhil Gopalkrishnan, Feng Xuan, and Peng Yin. Programmable autonomous synthesis of single-stranded DNA. *Nature chemistry*, 10(2):155–164, 2018.

18 Eric Klavins. Directed self-assembly using graph grammars. *Foundations of nanoscience: self assembled architectures and devices, Snowbird, UT*, 2004.

19 Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM journal of research and development*, 5(3):183–191, 1961.

20 Christopher Lutz and Howard Derby. Janus: a time-reversible language. *Letter to R. Landauer*, 2, 1986.

21 Kevin Montagne, Raphael Plasson, Yasuyuki Sakai, Teruo Fujii, and Yannick Rondelez. Programming an in vitro DNA oscillator using a molecular networking strategy. *Molecular systems biology*, 7(1):466, 2011.

22 Jennifer E Padilla, Matthew J Patitz, Robert T Schweller, Nadrian C Seeman, Scott M Summers, and Xingsi Zhong. Asynchronous signal passing for tile self-assembly: Fuel efficient computation and efficient assembly of shapes. *International Journal of Foundations of Computer Science*, 25(04):459–488, 2014.

23 Lulu Qian, David Soloveichik, and Erik Winfree. Efficient turing-universal computation with DNA polymers. In *DNA Computing and Molecular Programming: 16th International Conference, DNA 16, Hong Kong, China, June 14-17, 2010, Revised Selected Papers 16*, pages 123–140. Springer, 2011.

24 Ian Seet, Thomas E Ouldridge, and Jonathan PK Doye. Simulation of reversible molecular mechanical logic gates and circuits. *Physical Review E*, 107(2):024134, 2023.

25 Jong-Shik Shin and Niles A Pierce. A synthetic DNA walker for molecular transport. *Journal of the American Chemical Society*, 126(35):10834–10835, 2004.

26 Friedrich C Simmel, Bernard Yurke, and Hari R Singh. Principles and applications of nucleic acid strand displacement reactions. *Chemical reviews*, 119(10):6326–6369, 2019.

27 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7:615–633, 2008.

28 David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

29 Leo Szilard. Über die Entropieverminderung in einem thermodynamischen System bei Eingriffen intelligenter Wesen. *Zeitschrift für Physik*, 53(11-12):840–856, 1929.

30 Anupama J Thubagere, Wei Li, Hope A Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjan Srinivas, Damien Woods, et al. A cargo-sorting DNA robot. *Science*, 357(6356):eaan6558, 2017.

31 Kohji Tomita, Haruhisa Kurokawa, and Satoshi Murata. Graph-rewriting automata as a natural extension of cellular automata. *Adaptive Networks: Theory, Models and Applications*, pages 291–309, 2009.

32 Erik Winfree. *Algorithmic self-assembly of DNA*. California Institute of Technology, 1998.

33 Peng Yin, Harry MT Choi, Colby R Calvert, and Niles A Pierce. Programming biomolecular self-assembly pathways. *Nature*, 451(7176):318–322, 2008.

34 Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In *Proceedings of the 5th Conference on Computing Frontiers*, pages 43–54, 2008.

35 David Yu Zhang and Georg Seelig. Dynamic DNA nanotechnology using strand-displacement reactions. *Nature chemistry*, 3(2):103–113, 2011.

A Proofs

▶ **Theorem 1.** *The dynamics of the biased walker, in the long-run, are that of a biased random walk.*

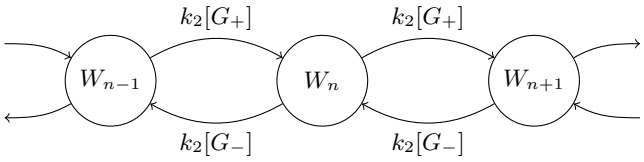
Proof. Recall the definition of the biased walker (Section 3.3) and its transition diagram (Figure 4d). We will assume typical CRN dynamics, namely that the evolution of the system may be described by a Continuous-Time Markov Chain (CTMC) with transition rates given by the law of mass action. For example, the reaction $W_\alpha + G_+ \rightleftharpoons W_\alpha{:}G_+$ has forward transition rate $k_2[W_\alpha][G_+]$ and reverse transition rate $k_1[W_\alpha{:}G_+]$ where $[X]$ is the concentration of species X, k_1 is the rate constant for unimolecular reactions, and k_2 that for bimolecular reactions (all the species have equal energy, and we assume no activation energy). Note the periodic symmetry of the walker scheme; without loss of generality, we can relabel the transition diagram to go from W_n to W_{n+1} where n corresponds to the n^{th} position along the track. We assume the track extends infinitely in both directions, so that $n \in \mathbb{Z}$. We first prove that the net reaction $W_n \rightleftharpoons W_{n+1}$ has rate constants $\propto [G_+]$ and $\propto [G_-]$ for the forward and reverse reactions.

The dynamics of the walker are somewhat complicated by the number of intermediate states – 13 between each net step (there are 15 distinct states in Figure 4d: 13 intermediate states, and W_α and W_β). However, after some convergence time, any initial distribution will converge to a steady state distribution (up to periodicity of the Markov Chain (MC)); this is analogous to the steady state approximation in chemistry. To see this, note that the behavior within each step-window (between W_n and W_{n+1}) is identical. Therefore, we can overlap each of these "sub-chains", reducing the infinite MC to the finite MC from W_n to W_{n+1} by symmetry. As this MC is aperiodic, irreducible, and reversible, it admits a steady

state given by detailed balance and this steady state is reached exponentially fast. After this "burn-in" time, the initial distribution is effectively forgotten.

For any pair of adjacent intermediate states i and j (states of the form $W{:}G$), the transition rate for $i \to j$ is $k_1[i]$. Consequently, by detailed balance their steady state concentrations are all equal (within a given step-window), and we can write this concentration as $[W{:}G]$. The remaining reactions are $W_n + G_+ \rightleftharpoons W_n{:}G_+$ and $W_{n+1}{:}G_- \rightleftharpoons W_{n+1} + G_-$. Using the transition rates given earlier, $k_2[W_n][G_+] = k_1[W{:}G] = k_2[W_{n+1}][G_-]$, and hence $[W_{n+1}]/[W_n] = [G_+]/[G_-]$. Furthermore, the net forward reaction rate constant for $W_n \rightleftharpoons W_{n+1}$ is $k_2[G_+]$, and the reverse reaction rate constant is $k_2[G_-]$.

Having proved this, we can reduce the original CTMC to the effective CTMC



which is prototypical of a random walk with bias $b = ([G_+] - [G_-])/([G_+] + [G_-])$. The expected value of n at time t, given $n(0) = 0$, will be $k_2 bt$. The variance is non-trivial for a continuous-time process, but will be approximately $\sqrt{k_2 t[G_+][G_-]}/([G_+] + [G_-])$. ◀

B Data & Compuzyme Motifs

Missing from the description of data-atoms and data-ports in Section 4.1 are the bond colors and atom configurations.

Data-ports consist of two ports. The first port (counting clockwise) has two bond colors, solid and dashed/‡. The second port is monochromatic. This design allows for the controlled and coordinated breaking of bonds.

Control ports have more colors to allow for the intricate signaling required by the motifs. These colors are: m, for "monomer"; solid/neutral, a bond that has no signaling intent; dashed/‡, for transitional states; C•, C‡, and C○, for displacing a data-atom from its C parent; and x•, x‡, and x○ for each child data-port x, for displacing child data-atoms. •, ‡, and ○ indicate that the relevant entity is bound, is transitioning between bound and unbound, or is unbound, respectively.

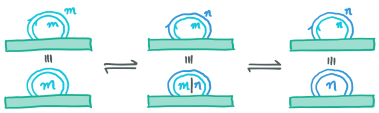
Wildcard control ports are much simpler, having just two bond colors: solid and dashed/‡.

With the bond colors enumerated, we now define the allowed RBL configuration for data-atoms and compuzymes. These are illustrated piecemeal in the remainder of this Appendix.

State changes

Not a motif in itself, compuzyme state changes are common to almost all our motifs. Compuzymes may perform a long series of manipulations, including branched control flow. As a result, indistinguishable configurations may be encountered, unexpectedly linking distinct parts of the computation. To prevent this, we imbue compuzymes with an internal "state" to track progress through configuration-space and correctly handle control flow. The

implementation is below:

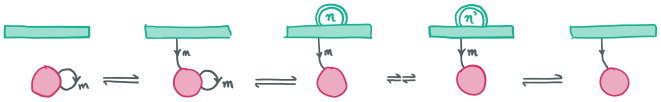


The top row is an RBL schematic, whereas the bottom row presents an abbreviated notation. The state is represented by a pair of self-loops. Each of these have the same set of bond colors, with each color representing a distinct state (e.g. m, n). Note that m and n are variables rather than explicit colors, i.e. m is not the same as the monomer color. Typically these states will be numbered, but any set of useful labels may be employed. There is also a special "$\varnothing$" state in which the bonds are broken; compuzymes typically start and end in this state. Changing state from m to n is simply a matter of changing the bond colors of the loops in turn.

The requirement of a pair of loops may seem superfluous; however, consider the case of two otherwise identical configurations that are meant to be distinguished by this state. If a single loop was used, then these configurations would be adjacent and so the state would not serve as a barrier. The double loop prevents this scenario, as the intermediate $m|n$ state will be constructed so as to be inaccessible.

M Monomer binding

Data molecules are formed from data-atoms, and so the manipulation of individual data-atom "monomers" is foundationally important. When building up data, fresh monomers will need to be drawn; conversely, when breaking down data the extraneous monomers will need to be discarded. We suppose that the environment provides an unlimited pool of fresh monomers for these purposes. Both drawing and discarding monomers can be implemented by a single motif, as they are inverses of each other and our system is reversible:



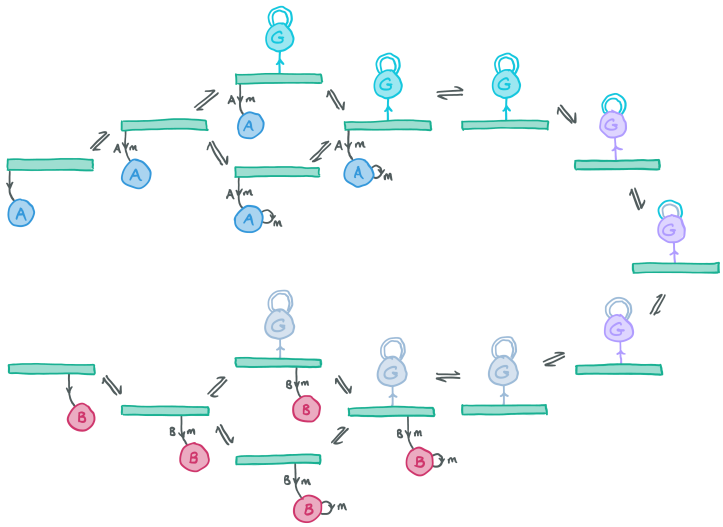
Here the monomer is the pink data-atom, and the compuzyme drawing (resp. discarding) the monomer is the green rectangle. In general we consider compuzymes as a platform for manipulations, hence the rectangular form. Both the free compuzyme and free monomer have an adjacent configuration in which the control ports form an "m-colored" bond. Consequently, the compuzyme is able to spontaneously form a bond with a free monomer of the correct type, drawing it from solution. Going forward, we will not explicitly mention adjacent configurations when they can be inferred from the diagrams. Recall that control ports are unique to a given data-atom type. As such, a compuzyme can be sure it is drawing the desired monomer from solution. We conclude the motif by switching the control port to the "neutral" bond color, ready for subsequent transitions.

Notice that the monomer in solution has a self-loop, also m-colored. Suppose that it didn't and instead had 0 bonds. In this case, the neutral bond in the final configuration could readily break. To address this, we assign an infinite energy to data-atoms with 0 bonds, and use the self-loop to mark monomers. Observe further that the compuzyme has no direct way to break the self-loop on the monomer. This is the reason for the existence of control ports: through different bond colors, we can signal different intents. Upon receiving such a signal, the normally-inert data-atom is able to transition to other configurations. Specifically,

there are usually $E = \infty$ energy barriers preventing a data-atom from transitioning to other configurations; but for specific control port colors, these energy barriers are lowered and a small region of configuration space is made available to explore. Note also the state change, the purpose of which we will elaborate further in Motif C.

X Monomer exchange

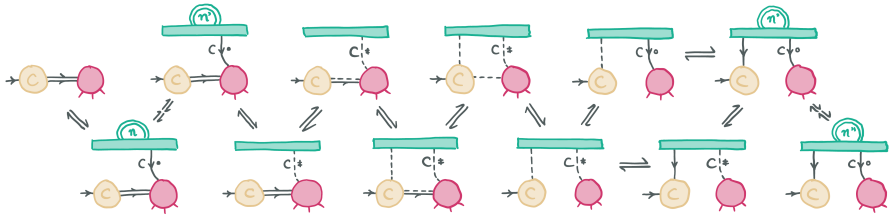
Related to the monomer binding motif is the monomer exchange motif. Arguably this is not a single motif in its own right, as it can be realized by combining two monomer-binding motifs back-to-back. However, monomer exchange is sufficiently common and useful that we promote it to its own motif. The implementation follows:



Notice that its implementation differs from two occurrences of Motif M: the state changes have been replaced by coupling to fuel. Though in principle any motif or state change can be coupled to the burning of fuel, we choose this motif as the canonical such point for convenience.

C Computational data binding

To bind computational data in order to manipulate it we need to (1) recognize it as such via its control port, and (2) displace it from its C parent:

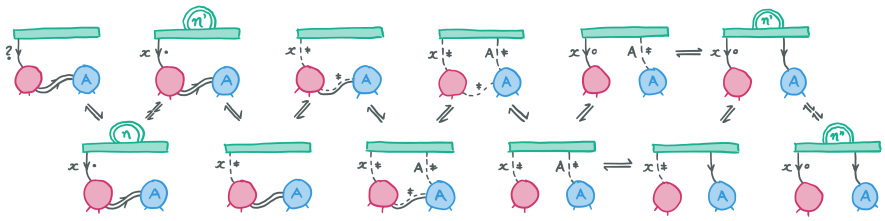


The C atom is in beige, and the computational atom in pink. Recall that the control port has three colors indicating progress of binding and displacement: C• corresponds to computational data with a C parent while C○ corresponds to computational data displaced from its parent. C‡ is a "transitional" state between these. We bind with C•, and then immediately change state from n to n'. These state changes are frequent in motifs and act as

bidirectional assertions. Without this, we could – for example – break the control port-bond during a transitional state. After this assertion, we switch to the transitional bond color C‡. In this configuration, the bond between computational atom and C parent can be weakened in a number of steps until it's broken completely. At the same time, we must form a bond with the soon-to-be-displaced C atom so as not to lose the association between them. This association is important, because this computation may be a subcomputation, with the particular C atom bound by another compuzyme. If we were to lose the association, then we could not return the result of this computation to the caller. Indeed, the series of adjacent configurations is programmed carefully so that the bond between C and the computational atom cannot be completely broken until the compuzyme has bound both. Finally we have another bidirectional assertion from state n' to n'', after which further transitions and motifs may occur. Notice that two of the last bond changes can happen in either order, hence the parallelism in the transition diagram.

D Destructuring

Data molecules are tree structures formed from data-atoms. To effectively manipulate these, we need to be able to break down (and build up) these structures into their constituent parts. We achieve this with the destructuring motif, which can be used to displace (resp. replace) a child data-atom from its parent. This motif is nearly identical to that of binding computational data, primarily differing in the control port bond colors used:



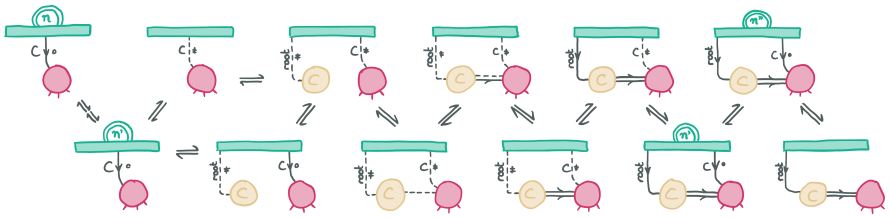
The ? bond "color" indicates that this motif doesn't care what the initial state of the control port is. Recall that a data-atom may have multiple children, each bound by a differently-labelled data-port. Here we are displacing a child data-atom (blue) from the x data-port of its parent (pink). Specifically, the child data-atom is of type A, and indeed this particular instance of Motif D can *only* displace an atom of type A. Note that the child data-atom may have its own children.

If we wish to introduce a branch (resp. merge) in control flow, then multiple of these motifs can be overlaid – one for each type of data-atom to be recognized. The first 5 configurations shown are common, as they are independent of the child data-atom type. The remaining configurations diverge. The nature of configuration adjacency ensures that this works as expected. We may also want to displace a data-atom regardless of its type. For this, we substitute the interactions with the control port for the wildcard control port.

S Subcomputation initiation

Because of the indirection provided by the C atom, compuzymes act independently of the context of a computation. This means that any routines we implement can also be used as subroutines. To call a subroutine, we prepare a "subcomputation": computational data that is bound to a host compuzyme and presented on a C atom. Other relevant compuzymes can then act on this subcomputation. Upon completion, the host compuzyme can accept the result and use it as required.

Prior to using this motif, the appropriate data representing the subcomputation should be prepared by means of the other motifs. Then, the subcomputation initiation motif will bind it to a fresh copy of the C atom:



This motif is specialized to the type of the data-atom. To complete the subroutine, we use another instance of this motif in reverse and specialized to the result data-atom. For example, if the initial data is tagged with $(+)$, then the final data would be tagged with $(+')$.

Now the purpose of the C atom becomes clear. We need to release the computational atom's control port so other compuzymes can interact with it, but then we would lose track of it. Therefore, we need to maintain a separate bond to this molecule. However, we cannot simply bind the data-atom on another port: it is important that this data-atom can be swapped out, for example $(+)$ for $(+')$, so as to indicate the completion of computation. As such, we use the intermediate atom C to achieve the necessary indirection to satisfy all of these conditions. In particular, we bind the C atom via its "root" port; its c port can then be bound by other compuzymes as needed.