Error-free stable computation with polymer-supplemented chemical reaction networks

by

Allison Yeu Yang Tai

B.Sc., University of British Columbia, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Bioinformatics)

The University of British Columbia
(Vancouver)

November 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Error-free stable computation with polymer-supplemented chemical reaction networks

submitted by **Allison Yeu Yang Tai** in partial fulfillment of the requirements for the degree of **Master of Science** in **Bioinformatics**.

Examining Committee:

Anne Condon, Computer Science
Supervisor

David Kirkpatrick, Computer Science
Supervisory Committee Member

Raymond Ng, Computer Science
Supervisory Committee Member

Steven Jones, Medical Genetics and Molecular Biology Biochemistry
Additional Examiner

Abstract

When disallowing error, traditional chemical reaction networks are very limited in computational power: Angluin et al. and Chen et al. showed that only semilinear predicates and functions are stably computable by CRNs. Qian et al. and others have shown that polymer-supplemented chemical reaction networks are capable of Turing-universal computation. However, their model requires that inputs are loaded onto the polymers at protocol initialization, in contrast with the traditional convention that inputs are represented by counts of molecules in solution. Here, we show that polymer-supplemented chemical reaction networks can stably simulate Turing-universal computations even with solution-based inputs. However, such simulations use a unique ”leader” polymer per input type and thus involve many slow bottleneck reactions. We further refine the polymer-supplemented chemical reaction network model to allow for clone polymers, that is, multiple functionally-identical copies of a polymer, and provide an illustrative example of how bottleneck reactions can be reduced in this new model.

Lay summary

In this paper, we design algorithms for the chemical reaction networks that leverages the additional power of polymers to compute functions and simulate powerful computational models that cannot be done traditionally. In particular, we demonstrate that we can simulate these functions and models in an error-free manner, meaning that our algorithm arrives at the correct answer every single time. This is not possible with the traditional chemical reaction network model, which is capable of computing only a very limited class of functions and predicates. Chemical reaction networks use molecules and their reactions to perform computation, by using a finite set of molecular species and a specified input to generate an output that corresponds to the answer. We also suggest methods to speed up some of the algorithms we design, by allowing multiple copies of each polymer species to exist simultaneously.

Preface

This thesis is the original, independent work of the author, A. Tai, with input on ideas and writing from A. Condon. A version of the document body was published as conference proceedings for DNA 25 [Tai A, Condon A. Error-free stable computation withpolymer-supplemented chemical reaction networks. DNA Computing and Molecular Programming, 197-218, 2019].

Table of contents

List of tables

List of figures

Acknowledgments

Thank you to my supervisor, Anne Condon, for her support with ideas and feedback throughout my degree, and for helping me with the technical aspects of the writing. I also thank my committee for their valuable feedback and insight while preparing this thesis.

Chapter 1

Introduction

The logical, cause-and-effect nature of chemical reactions has long been recognized for its potential to carry information and make decisions. Indeed, biological systems exploit interacting digital molecules to perform many important processes. Examples include inheritance with DNA replication, passing information from the nucleus to the cytoplasm using messenger RNA, or activating different cellular states through signal transduction cascades. Chemical reaction networks (CRNs) exploit these capabilities to perform molecular computations, using a finite set of molecular species (including designated input and output species) and reactions. Reactions among molecules in a well-mixed solution correspond to computation steps. CRN models may use mass-action kinetics, where the dynamics of the reactions are governed by ordinary differential equations, or stochastic kinetics, where the choice of reaction and length of time between reactions depends on counts of molecular species. We focus on stochastic CRNs in this work.

Stochastic CRNs using unbounded molecular counts are Turing-universal [19], but have a non-zero chance of failure. One challenge is that CRNs are unable to detect the absence of a molecule, and therefore when all molecules of a particular species have been processed. For example, when trying to simulate a counter machine, if their count of a species corresponds to a counter value, then a test-for-zero instruction needs to detect when all molecules have been depleted, i.e., their count is zero. Indeed, while the error of a CRN simulation of Turing-universal computation can be made arbitrarily small, it can never reach zero [9].

Error-free CRNs include those that exhibit *stable computation*: the output can change as long as it eventually converges to the correct answer; and *committing computation*: the presence of a designated "commit" species indicates that the output is correct and does not change subsequently. The class of predicates stably computable by CRNs is limited to semilinear predicates [3], and functions computable by committing CRNs are just the constant functions [9].

Cummings et al. [11] introduced the notion of *limit-stable computation*, which relaxes the stability requirement. In a computation of a limit-stable CRN, the output may change repeatedly, but the probability of changing the output from its correct value goes to zero in the limit. Cummings et al. show that any halting counter machine can be simulated by a limit-stable CRN. Their construction involves repeated simulations of a counter machine, resetting the counter values each time, along with slowing down any error-prone reactions each time they occur. They show that the computational power then becomes equivalent to that of a Turing machine with the ability to change its output a finite number of times, capable of deciding predicates in the class Δ_2^0 of limit-computable functions.

From these insights, we can see that CRN computations that produce the correct answer with probability 1 are still severely limited. We ask, "Is there any way to extend CRNs to work around the lack of ability to detect absence?" Qian et al. [17] gave a promising answer to this question by introducing a CRN model that is supplemented by polymers that behave as stacks, onto which monomers can be pushed and popped. Most importantly, this extended model allows for the stack base unit $\perp$ to be used as a reactant in implementing a "stack empty" operation. Indeed, Qian et al. use this operation in an error-free simulation of a stack machine. The resulting protocol, however, requires that the entire input is pre-loaded onto one of the stacks, a large change from traditional CRNs which assumes the inputs are well-mixed in a solution.

Motivated by the work of Qian et al., we wish to go one step further: Is Turing-universal stable computation by polymer-supplemented CRNs (psCRNs) possible when the input is represented by counts of monomers in a well-mixed solution? Intuitively, if input monomers can be loaded on to a polymer, absence of that species from the system could be detected by emptiness of the polymer, and thus circumvent a significant barrier to error-free, Turing-universal computation. The obvious obstacle is that it seems impossible to guarantee that all inputs are loaded on the polymer before computation can begin, if we can't reliably check for absence of inputs in the environment. At first glance, the logic appears circular, but we show that indeed stable Turing-universal computation is possible, and also present ideas for speeding up such computations.

In the rest of this chapter we describe our four main contributions and review related work. Chapter 2 introduces our polymer CRN models, Chapters 3, 4, and 5 describe our results, and Chapter 6 concludes with a summary and directions for future work.

1 Contributions and highlights

Stable counter machine simulation using CRNs with leader polymers. We design a polymer-supplemented CRN (psCRN) that unlike Qian et al., performs error-free Turing universal computation without requiring all inputs to be pre-loaded on polymers. To this end, instead of a binary stack machine, we simulate a schema that depends on unary counters. We do this by designing a protocol simulating a counter machine that performs computations assuming perfect loading of inputs, then add a mechanism that forces the simulation to continuously restart as long as inputs still exist in solution, only stopping once all inputs have been loaded onto their respective polymers. Therefore the protocol then completes successfully if and only if the last simulation began with all inputs properly loaded. Our scheme is similar to the error correction scheme of Cummings et al. in how we restart the simulation multiple times, but leverages polymers to ensure stable computation with 0 probability of error. Our polymer simulation of counter machines, and thus Turing-universal computation, uses a unique polymer per input species, as well as a "program counter" to ensure that execution of reactions follows the proper order. These molecules each have a count of one and so in the parlance of traditional CRNs, serve as "leaders". As a consequence, the simulation has a high number of so-called bottleneck reactions, which involve two leader reactants. Bottleneck reactions are undesirable because they are slow.

Clone polymers can help reduce bottleneck reactions. To reduce the number bottleneck reactions, we propose a CRN polymer model with no limit on the number of polymers of a given species, other than the limit posed by the volume of the system. In addition to type-specific increment, decrement and jump-if-empty operations (which are applied to one polymer of a given species), polymer stubs can be created or destroyed. We call such polymers "clones." We illustrate the potential of psCRNs with clone polymers to reduce bottleneck reactions, by describing psCRN to compute $f(n) = n2^{\lfloor \lg n \rfloor}$ (which is the same as n^2 when n is a power of 2).

Abstractions for expressing CRN multi-threading and synchronization. Our CRN for $f(n) = n2^{\lfloor \lg n \rfloor}$ uses threading to ensure that polymer reactions can happen asynchronously, and uses a "leader" polymer for periodic synchronization. To describe our psCRN, we develop threading abstractions for psCRNs with clone polymers.

Time complexity and a simulator for CRNs with clone polymers. To test the correctness of our psCRNs and evaluate their running times, we developed a custom CRN simulator designed to support clone polymers and their associated reactions. Underlying our simulator is a stochastic model of psCRN kinetics that is a natural extension of traditional stochastic CRNs and population protocols. We also use this model to analyze the expected time complexities of our psCRNs examples in this paper, showing how speedups are possible with clone polymers.

2 Related work

Soloveichik et al. [18] demonstrated how Turing-universal computation is possible with traditional stochastic CRNs, achieving arbitrarily small (but non-zero) error probability. For the CRN model without polymers Cummings et al. [11] showed how to reset computations so as to correct error and achieve limit-stable computation (which is weaker than stable computation).

In order to understand the inherent energetic cost of computation, Bennett [4, 5] envisioned a polymer-based chemical computer, capable of simulating Turing machines in a logically reversible manner. Qian et al. [17] introduced a stack-supplemented CRN model in which inputs are pre-loaded on stacks, and showed how the model can stably simulate stack machines. Johnson et al.[15] introduce a quite general linear polymer reaction network (PRN) model, allowing for arbitrary growth from both ends of a polymer as well as for polymers to have monomers of multiple types. With this model, they can simulate not only the original DNA stack machine model from Qian et al., which we also focus on, but copy-tolerant Turing Machines and microtubule dynamics. However, as the focus of their research is on simulation and verification, and does not address the issues raised in our work, namely achieving stable computation without pre-loaded inputs and reducing bottleneck reactions. Cardelli et al. [6] also demonstrated Turing-universal computation using polymers, using process algebra systems, but these systems are not stochastic. Jiang et al. [14] also worked on simulating computations with mass-action chemical reactions, using a chemical clock to synchronize reactions and minimize errors.

Lakin et al. [16] described polymerizing DNA strand displacement (DSD) systems, and showed how to model and verify stack machines at the DSD level. They also simulated their stochastic systems using a "just-in-time" extension of Gillespie's algorithm. Their model has a single complex to represent a stack. Recognizing limitations of this, they noted that "it would be desirable to invent an alternative stack machine design in which there are many copies of each stack complex...", which is what we do in this paper. They propose that updates to stacks could perhaps be synchronized using a clock signal such as that proposed by Jiang et al. [14]. In contrast, our synchronization mechanism is based on detection of empty polymers.

The population protocol (PP) model introduced by Angluin et al. [1], which is closely related to the CRN model, focuses on pairwise-interacting agents that can change state. In Angluin et al.'s model, agents in a PP are finite-state. An input to a computation is encoded in the agents' initial states; the number of agents equals the input size. Any traditional CRN can be transformed into a PP and vice versa. Chatzigiannakis et al. [7] expand the n agents to be Turing machines, then examine what set of predicates such protocols can stably compute using $O(\log n)$ memory. Although the memory capacity of our polymers can surpass $O(\log n)$, polymer storage access is constrained to be that of a counter or stack, unlike the model of Chatzigiannakis et al.

Chapter 2

Polymer-supplemented chemical reaction networks

A polymer-supplemented stochastic chemical reaction network (psCRN) models the evolution of interacting molecules in a well-mixed volume, when monomers can form polymers. We aim for simplicity in our definitions here, providing just enough capability to communicate our key ideas. Many aspects of our definitions can be generalized, for example by allowing multiple monomer species in a polymer, or double-end polymer extensibility, as is done in the work of Johnson et al. [15], Lakin et al. [16], Qian et al. [17], and others.

Reactions. A traditional CRN describes reactions involving molecules whose species are given by a finite set Σ. A *reaction*

$$r + r' \longrightarrow p + p' \tag{2.1}$$

describes what happens when at most one each of species $r, r' \in \Sigma$ are picked to be reactant molecules: they produce molecules of species $p \in \Sigma$ and $p' \in \Sigma$. In our model, exactly one of r, r' and one of p, p' can be null, so each reaction has at least one, but at most two, reactants and products. When both r, r' are not null, a single r molecule and single r' are assumed to have *interacted* or *collided*, and the reaction is *bimolecular*. Otherwise we say the non-null molecule *decomposes* into

p, p', one of which may be null, and the reaction is *unimolecular*. We assume that for all valid reactions, the multi-sets $\{r, r'\}$ and $\{p, p'\}$ are not equal, that for any r and r', there is at most one reaction with reactants of species r and r'. For now we do not ascribe a rate constant to a reaction; we will do that in Chapter 5.

Polymer-supplemented chemical reaction networks (psCRNs) also have reactions pertaining to polymers. A designated subset $\Sigma^{(m)}$ of Σ is a set of *monomers*. A *polymer* of type $\sigma \in \Sigma^{(m)}$, which we also call a σ-polymer, is a string $\perp_\sigma \sigma^i$, $i \geq 0$; its *length* is i and we say that the polymer is *empty* if its length is 0. We call $\perp_\sigma$ a *stub* and let $\perp = \{\perp_\sigma \mid \sigma \in \Sigma^{(m)}\} \subset \Sigma$. [16, 17]. Polymer reactions also involve molecules in a set $\mathscr{A} = \{A_{\sigma_{push}} \mid \sigma \in \Sigma^{(m)}\} \cup \{A_{\sigma_{pop}} \mid \sigma \in \Sigma^{(m)}\}$ of *guard molecules*, where $\mathscr{A} \subseteq \Sigma - \Sigma^{(m)}$. Generally, for each $\sigma \in \Sigma^{(m)}$ there are two polymer reactions, corresponding to σ-push and σ-pop, respectively:

$$\begin{aligned} [\perp_\sigma \ldots] + \sigma &\longrightarrow [\perp_\sigma \ldots \sigma] + A_{\sigma_{push}} \\ [\perp_\sigma \ldots \sigma] + A_{\sigma_{pop}} &\longrightarrow [\perp_\sigma \ldots] + \sigma \end{aligned}$$

A σ-push consumes a single monomer σ and a σ-polymer of length i to generate a guard $A_{\sigma_{push}}$ and a σ-polymer of length $i+1$. A σ-pop instead does almost the reverse, consuming a single guard $A_{\sigma_{pop}}$ and a σ-polymer of length i to generate a monomer σ along with a σ-polymer of length $i-1$. We cover the details of these reactions in the section discussing configurations.

Configurations. A *configuration* specifies how many molecules of each species are in the system at a specified time, keeping track also of the lengths of all σ-polymers. Formally, a configuration is a mapping $\mathbf{c} : \Sigma \cup \{\perp_\sigma \sigma^i \mid i \geq 1\} \rightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of nonnegative integers. We let $\mathbf{c}([\perp_\sigma \ldots])$ denote total number of σ-polymers in the system at a given time (including stubs) and let $\mathbf{c}([\perp_\sigma \ldots \sigma])$ denote the total number of σ-polymers in the system that have length at least 1. With respect to configuration $\mathbf{c}$, we say that a molecule of species $\sigma \in \Sigma$ is a *leader* if $\mathbf{c}(\sigma) = 1$.

A reaction of type (2.1) is *applicable* to configuration $\mathbf{c}$ if, when $r \neq r'$, $\mathbf{c}(r) \geq 1$ and $\mathbf{c}(r') \geq 1$, and when $r = r'$, $\mathbf{c}(r) \geq 2$. If the reaction is applied to $\mathbf{c}$, a new configuration $\mathbf{c}'$ is reached, in which the counts of r and r' decrease by 1 (when

$r = r'$ the count of r decreases by 2), the counts of p and p' increase by 1 (when $p = p'$ the count of p increases by 2), and all other counts remain unchanged. We also for now add a special restriction to all σ-polymer molecules: the only reactions they can be involved in are σ-pushes, σ-pops, and bimolecular reactions where they are not consumed or generated, such as $L_i + \perp_\sigma \longrightarrow L_k + \perp_\sigma$. At the same time, we enforce that $\mathbf{c}_0([\perp_\sigma \ldots]) = 1$ for all $[\perp_\sigma \ldots]$, restricting the total number of any σ-polymer to be 1, meaning every polymer in the system is a *leader* polymer. We defer the case of *clone* polymers, where $\exists[\perp_\sigma \ldots]$, $\mathbf{c}([\perp_\sigma \ldots]) > 1$, to Chapter 4.

Under the above, a σ-push is applicable if and only if $\mathbf{c}(\sigma) > 0$. The *result* of applying a σ-push reaction is that $\mathbf{c}(\sigma)$ decreases by 1, $\mathbf{c}(A_{\sigma_{push}})$ increases by 1 and also for some $i \geq 0$ such that $\mathbf{c}(\perp_\sigma \sigma^i) = 1$, $\mathbf{c}(\perp_\sigma \sigma^i)$ is set to 0 and $\mathbf{c}(\perp_\sigma \sigma^{i+1})$ is set to 1.

On the other hand, a σ-pop is applicable if and only if $\mathbf{c}([\perp_\sigma]) = 0$ and $\mathbf{c}(A_{\sigma_{pop}}) > 0$. This differs significantly from the traditional pops seen when computing with stacks, as unlike there, where only the stack itself is required for the pop, the presence of at least one $A_{\sigma_{pop}}$ is absolutely necessary. Similarly, the *result* of applying a σ-pop reaction is that $\mathbf{c}(\sigma)$ increases by 1, $\mathbf{c}(A_{\sigma_{pop}})$ decreases by 1, and for some $i \geq 1$ such that $\mathbf{c}(\perp_\sigma \sigma^i) = 1$, $\mathbf{c}(\perp_\sigma \sigma^i)$ is set to 0 while $\mathbf{c}(\perp_\sigma \sigma^{i-1})$ is set to 1.

Intuitively, after these reactions, the length of the σ-polymer in the system either grows or shrinks by 1, and correspondingly, the count of $A_{\sigma_{push}}$ either increases by 1, or $A_{\sigma_{pop}}$ decreases by 1.

If $\mathbf{c}'$ results from the application of some reaction to $\mathbf{c}$, we write $\mathbf{c} \to \mathbf{c}'$ and say that $\mathbf{c}'$ is *directly reachable* from $\mathbf{c}$. We say that $\mathbf{c}'$ is reachable from $\mathbf{c}$ if for some $k \geq 0$ and configurations $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k$,

$$\mathbf{c} \to \mathbf{c}_1 \to \mathbf{c}_2 \ldots \to \mathbf{c}_k \to \mathbf{c}'.$$

Computations and stable computations. We're interested in CRNs that compute, starting from some initial configuration $\mathbf{c}_0$ that contains an input. For simplicity, we focus on CRNs that compute functions $f : \mathbb{N}^k \to \mathbb{N}$. For example, the function may be Square, namely $f(n) = n^2$.

In a function-computing psCRN, the input $\mathbf{n} = (n_1, \ldots, n_k) \in \mathbb{N}^k$ is represented by counts of species in a designated set $\mathscr{I} = \{X_1, X_2, \ldots, X_k\} \subseteq \Sigma^{(m)}$ and the output is represented by the count of a different designated species $Y \in \Sigma^{(m)}$. In the initial configuration $\mathbf{c}_0 = \mathbf{c}_0(\mathbf{n})$, the initial counts of the input species X_i is n_i, $1 \leq i \leq k$, and the counts of all species other than the input species, including polymers and guard molecules, is 0, with the exception that there may be some leader molecules or polymers present. A leader molecule can function as a program counter, for example.

A *computation* of a psCRN is a sequence of configurations starting with an initial configuration $\mathbf{c}_0$, such that each configuration (other than the first) is directly reachable from its predecessor.

Let $\mathscr{C}$ be a psCRN, and let $\mathbf{c}$ be a configuration of $\mathscr{C}$. We say that $\mathbf{c}$ is *stable* if for all configurations $\mathbf{c}'$ reachable from $\mathbf{c}$, $\mathbf{c}(Y) = \mathbf{c}'(Y)$, where Y is the output species. The psCRN *stably computes* a given function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ if on any input $\mathbf{n} \in \mathbb{N}^k$, for any configuration $\mathbf{c}$ reachable from $\mathbf{c}_0(\mathbf{n})$, a stable configuration $\mathbf{c}'$ is reachable from $\mathbf{c}$ and moreover, $\mathbf{c}'(Y) = f(n)$. Finally if psCRN $\mathscr{C}$ stably computes a given predicate, we say that $\mathscr{C}$ is *committing* if $\mathscr{C}$ has a special "commit" species L_H such that for all $\mathbf{n} \in \mathbb{N}^k$, for any configuration $\mathbf{c}$ reachable from $\mathbf{c}_0(\mathbf{n})$ that contains species L_H, if $\mathbf{c}'$ is reachable from $\mathbf{c}$ then $\mathbf{c}'$ also contains L_H and $\mathbf{c}'(Y) = f(n)$.

To summarize, a *polymer-supplemented chemical reaction network (psCRN) function computer* is a 9-tuple $\mathscr{C} = (\Sigma, \Sigma^{(m)}, \mathscr{I}, \perp, \mathscr{A}, \mathscr{R}, \mathscr{L})$, where
– $\Sigma^{(m)} \subseteq \Sigma$ is the set of monomer types,
– $\mathscr{I} \subseteq \Sigma$ is the set of input types,
– $\perp = \{\perp_\sigma \mid \sigma \in \Sigma^{(m)}\}$ is the set of stub types,
– $\mathscr{A} = \{A_{\sigma_{push}} \mid \sigma \in \Sigma^{(m)} \cup \{A_{\sigma_{pop}} \mid \sigma \in \Sigma^{(m)}\}$ is the set of guard types,
– the sets $\Sigma^{(m)}$, $\perp$, and $\mathscr{A}$ are disjoint,
– $\mathscr{L} \subseteq \Sigma$ is the set of initial leader species,
– Σ contains the sets $\Sigma^{(m)}, \mathscr{I}, \perp, \mathscr{A}$ and $\mathscr{L}$,
– Σ also contains Y, and L_H, the output, and commit species types, and
– $\mathscr{R}$ is a set of reactions, including σ-push and σ-pop for all $\sigma \in \Sigma^{(m)}$.

Bottleneck reactions. In our CRN algorithms of Chapter 3, many reactions involve a sole leader ”state” molecule, or program counter, that reacts with a sole leader polymer. Such reactions, in which the count of both reactants are 1, are often described as bottleneck reactions [8]. As explained in Chapter 5, in a stochastic CRN that executes in a well-mixed system with volume V, the expected time for such a reaction is $\Theta(V)$ [18]. Our motivation for the clone polymer model in Chapter 4 is to explore how to compute with polymers in a way that significantly reduces bottleneck reactions.

Chapter 3

Stable, Turing-universal computation by sequential psCRNs with leader polymers

Here we describe how psCRNs with leader polymers can stably simulate counter machines, thereby achieving Turing-universal computation. Before doing so, we first introduce psCRN "pseudocode" which is convenient for describing psCRN algorithms. Then, as an illustration, we describe a psCRN to compute the Square function $f(n) = n^2$, under a special condition: we "fast-forward" from initial configuration $\mathbf{c_0}$ to some configuration $\mathbf{c_s}$ where all input-loading onto polymers has finished, meaning $\forall\sigma \in \Sigma^{(m)}$, $\mathbf{c}(\sigma) = 0$, and only run the protocol starting from that configuration.

We then describe afterwards how to guarantee that we can reach this special configuration $\mathbf{c_s}$, by coupling it with a simple counter machine simulation that builds strongly on Cummings et al. [11]'s register machine illustration. Note that although our method, like theirs, relies on re-simulating the protocol multiple times, we only re-simulate one final time after we reach $\mathbf{c_s}$, upon which the probability of error drops to 0, thus achieving stable computation, while Cummings et al. must keep repsimulating indefinitely, with the error of the simulations falling over time, but never reaching 0.

Sequential psCRN pseudocode. Following earlier work [2, 11, 17, 18], we describe a psCRN program as a set of instructions. Because one instruction must finish before moving on to the next, we call these *sequential* psCRNs. Corresponding to each instruction number i is a molecular "program counter" species $L_i \in \Sigma$. One copy of L_1 is initially present, and no other $L_{i'}$ for $i' \neq i$ is initially present.

The instructions `inc`(σ) and `dec`(σ) of Table 3.1 increase and decrease the length of a σ-polymer by 1, respectively, making it possible to use the polymers as counters. The `inc` instruction produces a σ which is then pushed on to the σ-polymer. In order to ensure that the push and pop reactions happen only within the `inc`() and `dec`() reactions, the `inc`(σ) operation generates the guard $A_{\sigma_{push}}$, which we convert into a separate molecule $I_\sigma \in \Sigma - \Sigma^{(m)}$ that cannot be used to pop polymers before the instruction execution completes, while the `dec`(σ) instruction changes I_σ into $A_{\sigma_{pop}}$ in order to reduce the length of a σ-polymer by 1. When a psCRN executes instructions of Table 3.1, starting from an initial configuration in which there is no polymer of length greater than 0, then we have the following invariant:

> **Invariant:** Upon completion of any instruction, the sum of the count of I_σ, $A_{\sigma_{push}}$, and $A_{\sigma_{pop}}$ equals the length of the σ-polymer.

Even more specifically, within this chapter, by using only `dec`(σ) and `inc`(σ) to pop and push σ onto polymers, upon the completion of any instruction, $\mathbf{c}(A_{\sigma_{push}})$ and $\mathbf{c}(A_{\sigma_{pop}})$ are always 0, and therefore the sum of the count of I_σ equals the length of the σ-polymer. Note that a σ-polymer is empty (has a length of 0) if and only if a stub $\perp_\sigma$ is present in the system. Therefore assuming that our invariant holds, the leader σ-polymer is not empty if and only if at least one I_σ molecule is in the system. Therefore the `jump-if-empty` instruction is very useful in ensuring that the program counter advances properly: a danger is that when the σ-polymer is empty, any `dec`(σ) instructions cannot proceed and may cause an algorithm to stall. The `jump-if-empty`(σ,k) instruction provides a way to first check whether the σ-polymer is empty, and if not, `dec`(σ) can safely be used. The `create` and `destroy` instructions provide a way to create and destroy copies of a species. While often more than one reaction is needed to implement one instruction, all will have completed when the instruction has completed and the program counter is set to the number of the next instruction to be executed in the pseudocode.

i: `inc(`σ`)`	$L_i \longrightarrow L_i^* + \sigma$ $\sigma + [\perp_\sigma \ldots] \longrightarrow A_{\sigma_{push}} + [\perp_\sigma \ldots \sigma]$ $L_i^* + A_{\sigma_{push}} \longrightarrow L_{i+1} + I_\sigma$
i: `dec(`σ`)`	$L_i + I_\sigma \longrightarrow L_i^* + A_{\sigma_{pop}}$ $A_{\sigma_{pop}} + [\perp_\sigma \ldots \sigma] \longrightarrow \sigma + [\perp_\sigma \ldots]$ $L_i^* + \sigma \longrightarrow L_{i+1}$
i: `jump-if` `-empty(`σ, k`)`	$L_i + \perp_\sigma \longrightarrow L_k + \perp_\sigma$ $L_i + I_\sigma \longrightarrow L_{i+1} + I_\sigma$
i: `goto(`k`)`	$L_i \longrightarrow L_k$
i: `create(`σ`)`	$L_i \longrightarrow L_{i+1} + \sigma$
i: `destroy(`σ`)`	$L_i + \sigma \longrightarrow L_{i+1}$
i: `halt`	$L_i \longrightarrow L_H$

Table 3.1: Instruction abstractions of psCRN reactions. The decrement dec(σ) instruction can complete only if the σ-polymer has length is at least 1.

Pseudocode instructions may also be function calls, where a function is itself a sequence of instructions expressed as pseudocode. Suppose again that there is a leader σ-polymer and also a leader σ'-polymer in the system. Then the `add-to`(σ, σ') function (using a temporary τ-polymer) extends the length of the σ'-polymer by the length of the σ-polymer. Another useful function is `flush`(σ) which decrements the (leader) σ-polymer until its length is 0. A third function, `release-output`(σ), is useful to "release" molecules on a (leader) σ-polymer as Y molecules into the solution. This function uses an additional special leader Y'-polymer which is empty in the initial configuration, and whose length at the end of the function equals the number of released Y molecules. The Y' molecule will be useful later, when we address how a psCRN can be restarted (and should not be used elsewhere in the code).

i: `add-to`(σ,σ')	i : `goto`$(i.1)$
	$i.1$: `jump-if-empty`$(\sigma, i.6)$
	$i.2$: `dec`(σ)
	$i.3$: `inc`(σ')
	$i.4$: `inc`(τ)
	$i.5$: `goto`$(i.1)$
	$i.6$: `jump-if-empty`$(\tau, i.10)$
	$i.7$: `dec`(τ)
	$i.8$: `inc`(σ)
	$i.9$: `goto`$(i.6)$
	$i.10$: `goto`$(i+1)$

i: `flush`(σ)	i: `goto`$(i.1)$
	$i.1$: `jump-if-empty`$(\sigma, i+1)$
	$i.2$: `dec`(σ)
	$i.3$: `goto`$(i.1)$

i: `release-output`(σ)	i: `goto`$(i.1)$
	$i.1$ `jump-if-empty`$(\sigma, i+1)$
	$i.2$ `dec`(σ)
	$i.3$ `inc`(Y')
	$i.4$ `create`(Y)
	$i.5$ `goto`$(i.1)$

Numbering of function instructions. For clarity, we use $i.1, i.2$, and so on to label the lines of a function called from line i of the main program. Upon such a function call, the CRN's program counter first changes from L_i to $L_{i.1}$. The program counter is restored to L_{i+1} upon completion of the function's instructions, e.g., via a `goto`$(i+1)$ instruction or a `jump-if-empty`$(\sigma,\ i+1)$ instruction. If one function f_B is called from line $a.b$ of another function f_A, the program counter labels would be $a.b.1$, $a.b.2$ and so on, and so the label "i" in the function description should be interpreted as "$a.b$". In this case, when the function f_B completes, control is passed back to line $a.(b+1)$ of function f_A; that is, the "`goto`$(i+1)$"

statement should be interpreted as "$\texttt{goto}(a.(b+1))$". Also for clarity, we use special labeling of instructions in a few special places, such as the restart function below, in which instructions are labeled $\texttt{S1}, \texttt{S2}$ and so on.

psCRNs with fast-forwarding. As noted in the introduction, a challenge in achieving stable computation with psCRNs is detecting the absence of inputs. To build up to our methods for addressing this challenge, we first ignore this issue, fast-forwarding to a new configuration $\mathbf{c_s}$ directly reachable from $\mathbf{c_0}$. By this we mean that if the input contains n_i molecules of a given species X_i, then in configuration $\mathbf{c_s}$ there is a unique X_i-polymer of length n_i. Furthermore, there are n_i copies of the molecule I_{X_i} in the system. Intuitively, the configuration $\mathbf{c_s}$ is one that would be reached if n_i $\texttt{inc}(X_i)$ operations were performed from an initial configuration with no inputs and an empty X_i-polymer, for every input species X_i.

A committing, sequential psCRN with fast-forwarding for Square. Our psCRN for the Square function $f(n) = n^2$ has one input species X and one output species Y, and $\Sigma^{(m)} = \{X, X', X'', Y_{int}, Y', \tau\}$. In the fast-forwarded configuration, the input is represented as the length n of a leader X-polymer, and the count of I_X is n. The only other molecules in the initial configuration are the leader program counter L_1, and stubs $\perp_{X'}$, $\perp_{X''}$, $\perp_{Y_{int}}$, and $\perp_{Y'}$, and $\perp_\tau$. Note that instead of using the X-polymer directly, we first copy it to the X'-polymer in a line not numbered as part of the main program, line $\texttt{P}$. The distinction will become useful when we remove the assumption of fast-forwarding later. The psCRN has a loop (implemented using $\texttt{jump-if-empty}$ and $\texttt{goto}$) that executes n times, adding n to an intermediate Y_{int}-polymer each time. When the loop completes, the output is released from the Y_{int}-polymer in the form of Y, so that the number of Y's in solution is n^2, and the psCRN halts. The halting state is in effect a committing state, since no transition is possible from L_H.

Committing Turing-universal computation by psCRNs with fast-forwarding. Turing-universal computation is possible with a two counters machine (2CM). To simulate a halting 2CM that computes function $f : \mathbb{N}^k \to \mathbb{N}$ using unary counters, we create a

Algorithm 1 Sequential-n^2-psCRN, with input n on X-polymer.

```
P: add-to($X$,$X'$)
1: add-to($X'$,$X''$)
2: jump-if-empty($X'$, 6)
3:     dec($X'$)
4:     add-to($X''$,$Y_{int}$)
5:     goto(2)
6: release-output($Y_{int}$)
7: halt
```

psCRN has 2 unary counters R'_1, R'_2, where R'_1 contains the input n in unary, while R'_2 is initially 0. Throughout the simulation of the counter machine, the psCRN has exactly one R'_1-polymer and one R'_2 polymer, to represent the aforementioned unary counters. In addition, there is one additional polymer, a Y'-polymer, which is initially empty and is used by the `release-output` function. A two counters machine program is a sequence of instructions, where instructions can increment a counter; decrement a non-empty counter; test if a counter is empty (0) and jump to a new instruction if so, or halt. Table 3.1 already shows how all four of these instructions can be implemented using a psCRN. We assume in what follows that these are the only instructions used by the psCRN simulator. At the end, the output is stored on counter R'_1, then released into solution, using `release-output`(R'_1), once the machine being simulated reaches its halt state.

We assume that `release-output`(R'_1) is the only function call of the 2CM simulator.

Stable, Turing-universal computation by psCRNs. We now handle the case where we start from initial configuration $\mathbf{c_0}$, where the input is represented as counts of molecules instead. That is, in the initial configuration of the psCRN all polymers are empty, including the R'_1 and R'_2-polymers; instead, we have n copies of molecule R_1 in solution, and a new R_1-polymer that is not directly part of the 2CM simulation for them to be loaded on. Our scheme uses the R_1-push reaction to load inputs. Note that here, R_1 counter, although not used by the 2CM simulator for computation, serves a very important role as a back-up counter: it is used to restore the value of R'_1 in case the simulation began too early, before input loading finishes.

We will demonstrate how, by adding CRNs to detect input-loading and to restart the simulator in this case. Once all inputs are loaded, and the protocol starts one final time, the system is never subsequently restarted. Overall our simulation has four components:

- **Input loading:** This is done as R_1-push, which can happen at any time until all inputs are loaded. Recall that the R_1-push reaction is

$$[\perp_{\mathrm{R}_1}\ldots]+\mathrm{R}_1 \quad\longrightarrow [\perp_{\mathrm{R}_1}\ldots\mathrm{R}_1]+A_{R_{1push}}$$

  Each such reaction generates a guard molecule A_{R_1} which, as explained below, triggers input detection.

- **Two counters machine (2CM) simulation:** Algorithm 2 shows the simulator. This psCRN program has a "prelude" line P, that copies the inputs on the R_1-polymer to the R'_1-polymer. Then starting from line numbered 1, the simulation does the computation as described before using the R'_1 and R'_2-polymers. Upon completion of the computation, the output is released from counter R'_2, and the simulator halts (produces theL_H species).

Algorithm 2 Sequential-2CM-psCRN.

```
P: add-to($R_1$,$R'_1$)
1: // Rest of psCRN simulation pseudocode here, using
2: // $R'_1$ and $R'_2$
 :       ...
 : // ending with release-output($R'_2$) function and halt instruction.
```

- **Input detection:** This is triggered by the presence of an active guard molecule $A_{R'_{1push}}$. For each value i of the main program counter, after the prelude add-to, as well as for L_H, we have the following reactions, where S1 is the first number of the `restart` pseudocode (see below). The reactions convert the guard $A_{R_{1push}}$ molecule into a molecule, I_{R_1}, since the input molecule is now loaded, and also changes the program counter to $L_{\mathrm{S}1}$, which triggers restart.

$$L_i + A_{R_{1push}} \longrightarrow L_{\mathrm{S}1} + I_{R_1},$$
$$L_H + A_{R_{1push}} \longrightarrow L_{\mathrm{S}1} + I_{R_1}$$

L_H is no longer a committing species, since it may change to $L_{\text{S}1}$.

Line P is a function call to `add-to`(R_1,R_1'), which executes intructions numbered P.1 through P.11 of `add-to`. Input detection is only done at line P.2, the first `jump-if-empty` instruction:

$$L_{\text{P}.2} + A_{R_{1push}} \longrightarrow L_{\text{P}.2} + I_{R_1}$$

It does not trigger a restart, but simply converts the guard molecule $A_{R_{1push}}$ into I_{R_1}.

- **Restart:** Restart happens a number of times that is exactly the total input length $n_1 + n_2 + \ldots n_k$, since each input molecule is detected by the system exactly once, generating one guard molecule. The counters R_1' and R_2' are flushed, and any outputs that have been released in solution are destroyed, assuming that the number of outputs released into the solution was tracked by some Y' counter, as before. Then the program counter is set to line P of the simulator (leader molecule L_{P}). Algorithm 3 shows the restart pseudocode.

Algorithm 3 Restart

```
S1: flush($R_1'$)
S2: flush($R_2'$)
S3: destroy-output()
S4: goto(P)
```

i: `destroy-output()`	i: `goto`(i.1)
	i.1: `jump-if-empty`(Y',i.5)
	i.2: `dec`(Y')
	i.3: `destroy`(Y)
	i.4: `goto`(i.1)
	i.5: `goto`($i+1$)

Correctness of Algorithm 2: Sequential-RM-psCRN. We claim that our complete 2CM simulator stably computes the same function as the counter machine. (We note that no "fairness" assumption regarding the order in which reactions happen is necessary to show stability, since stability is a "reachability" requirement.)

The delicate part of the simulation lies in the instruction on line P, which copies input counter R_1 to R'_1.

Unlike the previous chapter, the using the R_1-push reaction for input loading means that the count of I_{R_1} is no longer guaranteed to equal the length of the (leader) R_1-polymer. Instead, we have that I_{R_1} is less than or equal to the length of the R_1-polymer. This is because when input monomer R_1 is pushed onto its polymer, the $A_{R_{1push}}$ guard molecule is produced, not the I_{R_1} molecule, and the A_{R_1} molecule is not converted to I_{R_1} until input detection happens.

This delay in converting $A_{R_{1push}}$ and generating I_{R_1} can cause the `jump-if-empty` instruction numbered P4.2 in the `add-to` function to stall, when there is no I_{R_1} and also no $\perp_{R_1}$ (since the R_1-polymer is not empty due to input loading). In this case, the input detection reaction (introduced above)

$$L_{\mathrm{P}.2} + A_{R_{1push}} \longrightarrow L_{\mathrm{P}.2} + I_{R_1}$$

will convert $A_{R_{1push}}$ to I_{R_1}. This averts stalling, since `jump-if-empty` can proceed using I_{R_1}. The subsequent lines of the `add-to`(R_1,R'_1) code can then proceed.

Once line P has completed, the correctness of the psCRN simulation of the 2CM, using the copies R'_1 and R'_2, is not affected by input loading. Input loading and input detection can also proceed. These are the only viable reactions from the "halting" state L_H, and so eventually (since the RM machine being simulated is a halting machine), on any sufficiently long computation path, all inputs must be loaded and detected. Input detection after the prelude phase produces a "missing" I_σ molecule and triggers a restart. The restart flushes all counters used by the simulator, and also, using the Y-polymer, destroys any outputs in solution. (Since restart is triggered only when the program counter is at a line of the main program, restart does not interrupt execution of the `release-output` function.) A new simulation is then started at line P, ensuring that any inputs that have been loaded since the last detect are copied to the simulator's input counters R'_1.

Once all inputs have been detected, the invariant is restored and the simulator proceeds correctly, producing the correct output. This correct output is never subsequently changed, and so the computation is stable.

Bottleneck reactions. In our sequential psCRNs, both `inc`(σ) and `dec`(σ) contain bottleneck reactions, and so `add-to`(σ, σ') has $\Theta(|\sigma|)$ bottleneck reactions. Thus the psCRN for Square has $\Theta(n^2)$ bottleneck reactions. In the next chapter we show how to compute a close variant of the Square function with fewer bottleneck reactions, using clone polymers rather than leader polymers.

Chapter 4

Faster computation of Square by threaded psCRNs with clone polymers

Configurations when considering clone polymers. Previously we enforced that $\forall[\perp_\sigma...]$, $\mathbf{c}([\perp_\sigma...]) = 1$ by enforcing that the only reactions involving polymers were σ-pushes and σ-pops. Now, we will remove this requirement, and let $\mathbf{c}([\perp_\sigma...])$ be any non-negative integer. We do this by introducing two new instructions: `create-polymer`(σ) and `destroy-polymer`(σ).

i: `create-polymer`(σ)	$L_i \longrightarrow L_{i+1} + \perp_\sigma$
i: `destroy-polymer`(σ)	$L_i + \perp_\sigma \longrightarrow L_{i+1}$

`create-polymer`(σ) generates an empty polymer while `destroy-polymer`(σ) consumes a empty polymer. For simplicity, we also enforce that the initial configuration $\mathbf{c}_0$ of any psCRN contain no polymers, that is, $\forall[\perp_\sigma...]$, $\mathbf{c}_0([\perp_\sigma...]) = 0$, and so any polymers that are needed must be explicitly created during computation: for the polymer-dependent functions we introduced in Chapter 3, we add `create-polymer` and `destroy-polymer` as needed. As an example, `add-to(`σ, σ'`)` now requires
`create-polymer(`τ`)` before line i.1 and `create-polymer(`τ`)` before line

i.10.

With this, the mechanics of the σ-pushes and σ-pops also change. Now, the *result* of a σ-push is the same except that for exactly one $i \geq 1$ such that $\mathbf{c}[\perp_\sigma \sigma^i] > 0$, $\mathbf{c}[\perp_\sigma \sigma^{i+1}]$ increases by 1, while $\mathbf{c}[\perp_\sigma \sigma^i]$ decreases by 1. In the same way, the *result* of a σ-pop now has that for exactly one $i \geq 1$ such that $\mathbf{c}[\perp_\sigma \sigma^i] > 0$, $\mathbf{c}[\perp_\sigma \sigma^{i-1}]$ increases by 1, while $\mathbf{c}[\perp_\sigma \sigma^i]$ decreases by 1. Intuitively, the length of some arbitrary σ-polymer in the system either grows or shrinks by one, and many σ-polymers may be present in the system simultaneously.

As we will see in Chapter 5, the time complexity of σ-pops and σ-pushes are now different as they are no longer always simple bottleneck reactions; this is due to the fact that all clone polymers are indistinguishable to the system: in both cases, the affected polymer is chosen nondeterministically from among the clones. Exactly how the polymer is chosen is not important in the context of stable computation, as long as every σ-polymer with positive count in $\mathbf{c}$ has positive probability of being chosen. For example, the polymer could be chosen uniformly at random, consistent with the model of Lakin and Phillips [16], and in this work, this is what we assume.

These modifications mean that `inc(`σ`)` and `dec(`σ`)` instructions can now operate on any of the many functionally-identical clone σ-polymers that may be in this system, rather than just a single leader polymer, and thus reduce the number of mandatory bottleneck reactions. Now we demonstrate the increased power arising from these changes, using the function $f(n) = n2^{\lfloor \lg n \rfloor}$ as an example, and focus only on a system that has been fast-forwarded. Input detection and loading can be layered on, in a manner similar to Chapter 3.

Under these conditions, we start with a single Y-polymer of length n, and we wish to create a total of $2^{\lfloor \lg n \rfloor}$ Y-polymers, whose lengths sum to $n2^{\lfloor \lg n \rfloor}$. Algorithm 4 proceeds in $\lfloor \lg n \rfloor$ rounds (lines 6-9), doubling the number of Y's on each round. To keep track of the Y molecules, we introduce a distributed σ-counter data structure, and use it with $\sigma = Y$. The data structure consists of σ-polymers that we call *σ-thread-polymers*, plus a thread-polymer counter T_σ, which is a leader polymer whose length, $|T_\sigma|$, is the number of σ-thread-polymers. The *value* of this distributed counter is the total length of all σ-thread-polymers. We explain below how operations on this distributed counter work.

Algorithm 4 Threaded-$n2^{\lfloor \log n \rfloor}$-psCRN.

```
P1: create-polymer($H$)
P2: add-to($X$,$H$)
P3: create-distributed-counter($Y$)
P4: add-thread-polymer($Y$,1)
P5: add-to($X$,$Y$)
 1: halve($H$)
 2: jump-if-empty($H$,5)
 3:     double($Y$)
 4:     goto(1)
 5: halt
```

Algorithm 4 counts the number of rounds using a leader H-polymer, whose length is halved on each round. The `halve` function is fairly straightforward to implement, using instructions and functions already introduced in Chapter 3.

i: `halve`(H)	i: `goto`(i.1)
	i.1: `create-polymer`(H')
	i.2: `add-to`(H,H')
	i.3: `jump-if-empty`(H', i.9)
	i.4: `dec`(H')
	i.5: `dec`(H)
	i.6: `jump-if-empty`(H', i.9)
	i.7: `dec`(H')
	i.8: `goto`(i.3)
	i.9: `destroy-polymer`(H')
	i.10: `goto`($i+1$)

The `double`(σ) function of Algorithm 4 is where we leverage our distributed Y-counter (with $\sigma = Y$). Recall that a distributed σ-counter data structure consists of a set of clone σ-polymers, which we call σ-thread-polymers, plus a thread-polymer counter T_σ, which is a leader polymer whose length is the number of σ-thread-polymers. The `double` function first creates two other distributed counters τ and τ' (lines i.1 and i.2), and gives each the same number of thread-polymers as σ, namely $|T_\sigma|$ thread-polymers (lines i.3 and i.4), all of which are empty. The heart of `double` (line $i.5$) transfers the contents of the distributed σ-counter to τ and τ', emptying and destroying all σ-thread-polymers in the process. It then creates double the original number of (empty) σ-thread-polymers (lines $i.6$ and $i.7$; note that the number of threads of τ is the original value of $|T_\sigma|$). It finally transfers the τ and τ' polymers back to σ (lines $i.8$ and $i.9$), thereby doubling σ.

i: double(σ)	$i.1$ `create-distributed-counter`(τ)
	$i.2$ `create-distributed-counter`(τ')
	$i.3$ `add-thread-polymers`(τ, T_σ)
	$i.4$ `add-thread-polymers`(τ', T_σ)
	$i.5$ `transfer`(σ, τ, τ')
	$i.6$ `add-thread-polymers`(σ, T_τ)
	$i.7$ `add-thread-polymers`(σ, $T_{\tau'}$)
	$i.8$ `transfer`(τ, σ)
	$i.9$ `transfer`(τ', σ)
	$i.10$ `destroy-distributed-counter`(τ)
	$i.11$ `destroy-distributed-counter`(τ')
	$i.12$ `goto`($i+1$)

Next are details of instructions used to create an empty distributed counter, and to add empty threads to the counter. Again, these are all straightforward sequential implementations (no threads), using leader polymers to keep track of counts.

i: `create-distributed-counter`(σ)
// Creates an empty counter
// with zero polymers
	i: `goto`(i.1)
	i.1 `create-polymer`(T_σ)
	i.2 `goto`($i+1$)

i `add-thread-polymers`(σ, T)
// Adds $\|T\|$ empty
// thread-polymers to the
// distributed σ-counter,
// where T is a counter
	i `goto`(i.1)
	i.1: `create-polymer`(Temp)
	i.2: `add-to`(T,Temp)
	i.3: `jump-if-empty`(Temp, i.8)
	i.4: `dec`(Temp)
	i.5: `create-polymer`(σ)
	i.6: `inc`(T_σ)
	i.7: `goto`(i.3)
	i.8: `destroy-polymer`(Temp)
	i.9: `goto`($i+1$)

i `add-thread-polymer`($\sigma, 1$)
// Adds one empty
// thread-polymer to the
// distributed σ-counter
	i `goto`(i.1)
	i.1: `create-polymer`(σ)
	i.2: `inc`(T_σ)
	i.3: `goto`($i+1$)

The `transfer` function transfers the value of a distributed σ-counter to two other distributed counters called τ and τ'. In line i.2 of `transfer`, function `create-threads` creates T_σ identical "thread" program counters, L_t. Once again this is straightforward, using a leader polymer to keep track of counts. All of the thread program counters execute the `thread-transfer` function in line i.4 of `transfer`, thereby reducing bottleneck reactions (details below). The "main" program counter, now at line i.4 of the `transfer` function, can detect when all threads have completed, because each decrements Thread-Count exactly once, and so Thread-Count has length zero exactly when all threads have completed. At that point, the main program counter progresses to line i.5, destroying the thread program counters using the `destroy-threads` function (not shown, but uses the `destroy` function to destroy each single thread).

The function `transfer`(σ,τ), not shown but used in `double`, is the same as `transfer`(σ,τ,τ'), except the call to `thread-transfer` does not include τ' and the "`inc`(τ')" line is removed in the implementation of `thread-transfer`.

i: `transfer`(σ,τ,τ')
// transfer σ to
// both τ and τ'
	i: `goto`(i.1)
	$i.1$: `create-polymer`(Thread-Count)
	$i.2$: `create-threads`(T_σ, L_t)
	$i.3$: `add-to`$(T_\sigma$, Thread-Count)
	$i.4$: `loop-until-empty`(Thread-Count, $i.5$)
	    `thread-transfer`$(\sigma,\tau,\tau'$,Thread-Count)
	$i.5$: `destroy-threads`(L_t)
	$i.6$: `destroy-polymer`(Thread-Count)
	$i.7$: `goto`$(i+1)$

i: `create-threads`(T_σ,L_t)
// create T_σ thread
// program counters, L_t
	i: `goto`(i.1)
	$i.1$: `create-polymer`(Temp)
	$i.2$: `add-to`$(T_\sigma$, Temp)
	$i.3$: `jump-if-empty`(Temp, $i.7$)
	$i.4$ `dec`(Temp)
	$i.5$ `create`(L_t)
	$i.6$ `goto`$(i.3)$
	$i.7$: `destroy-polymer`(Temp)
	$i.8$: `goto`$(i+1)$

i : `loop-until-empty`(σ,k)	$L_i + \bot_\sigma \longrightarrow L_k + \bot_\sigma$

Finally, we describe how threads work in `thread-transfer`. The `threadon`() instruction executes $|T_\sigma|$ times, one per copy of L_t, thereby creating $|T_\sigma|$ L_{t_1} program counters that execute computation "threads". Using the instruction `dec-until-destroy-polymer`, each thread repeatedly (zero or more times) decrements one of the σ-thread-polymers and then increments both τ and τ'. This continues until the thread finds an empty σ-thread-polymer, i.e., the stub $\bot_\sigma$, in which case it destroys the stub and moves to line $t.5$. The `dec`(σ) and `inc`(σ) functions of Chapter 3 work exactly as specified, even when applied to distributed counters. A key point is that the threads work "clonely" with the thread-polymers; it is not the case that each thread "owns" a single thread-polymer. Accordingly, one thread may do more work than another, but in the end all thread-polymers are empty.

A thread exits the `dec-until-destroy-polymer` loop by destroying exactly one σ-polymer. Since at the start of `thread-transfer` the number of σ-thread-polymers equals the number of thread program counters, all thread program counters eventually reach line $t.5$, and there are no σ-thread-polymers once all threads have reached line $t.5$ of the code. At line $t.5$, each thread decrements Thread-Count, and then stalls at line $t.6$. Moreover, once all threads have reached line $t.6$, polymer ThreadCount is empty. At this point, the program counter for `transfer` changes from line $i.4$ to line $i.5$, and all thread program counters are destroyed.

i: `thread-transfer` (σ,τ,τ',Thread-Count)	i: `threadon()` $t.1$: `dec-until-destroy-polymer(`σ, $t.5$`)` $t.2$: `inc(`τ`)` $t.3$: `inc(`τ'`)` $t.4$: `goto(`$t.1$`)` $t.5$: `dec`(Thread-Count) $t.6$:

i: `threadon()`:	$L_i + L_t \longrightarrow L_i + L_{t.1}$
i: `dec-until-destroy -polymer(`σ,k`)`	$L_i + I_\sigma \longrightarrow L_i^* + A_\sigma$ $A_\sigma + [\perp_\sigma \ldots \sigma] \rightleftharpoons \sigma + [\perp_\sigma \ldots]$ $L_i^* + \sigma \longrightarrow L_{i+1}$ $L_i + \perp_\sigma \longrightarrow L_i^{**}$ $L_i^{**} + I_{T_\sigma} \longrightarrow L_i^{***} + A_{T_\sigma}$ $A_{T_\sigma} + [\perp_{T_\sigma} \ldots T_\sigma] \rightleftharpoons T_\sigma + [\perp_{T_\sigma} \ldots]$ $L_i^{***} + T_\sigma \longrightarrow L_k$

Correctness. We claim that on any input $n \geq 0$, as long as it's been loaded beforehand on a leader X-polymer using `inc(`X`)`, Algorithm 4: Threaded-$n2^{\lfloor \lg n \rfloor}$-

psCRN eventually halts with the value of the distributed-Y-counter being $f(n) = n2^{\lfloor \log n \rfloor}$.

The algorithm creates and initializes H to be a polymer of length n (lines 1-2), and the Y-distributed-counter to have a single polymer-thread of length n (lines 3-5). When $n = 0$, H is empty, so from line 6 the algorithm jumps to line 10 and halts, with the value of Y being $f(0) = 0$ as claimed.

Suppose that $n > 0$. Reasoning about the `halve` function is straightforward, since it is fully sequential. We claim that in each round of the algorithm (lines 6-9), lines 7 and 8 complete successfully, with $|H|$ halving (that is, $|H| \rightarrow \lfloor |H|/2 \rfloor$) in line 7, and with both the value of Y and $|T_Y|$, the number of Y-thread-polymers, doubling in line 8. As a result, $|H| = 0$ after $\lfloor \lg n \rfloor$ rounds and the algorithm halts with value(Y) $= f(n)$.

Correctness of the `double` function is also straightforward to show, if we show that the `transfer`(σ, τ, τ') (and the `transfer`(σ, τ) variant) works correctly.

Line i.4 is the core of `transfer`. We show that line i.4 does complete, that is, Thread-Count does become empty, that execution of line i.4 increases the values of distributed counters τ and τ' by the value of σ (while leaving the number of τ- and τ'-thread-polymers unchanged), and also changes value(σ) and the number of σ-thread-polymers to 0.

The `loop-if-empty` instruction ensures that the main program counter must stay at line i.4 of function `transfer` until Thread-Count is empty. Meanwhile, this main program counter can also activate threads using the `threadon`() function, that is, change the thread program counters from L_t to $L_{t.1}$. From line i.2 of `transfer`, the number of such thread program counters is $|T_\sigma|$.

Each of these program counters independently executes `thread-transfer`. At line t.1, either (i) a `dec`(σ) is performed (first three reactions of
`dec-until-destroy-polymer`), or (ii) a σ-polymer-thread is destroyed and the polymer-thread-count T_σ is decremented (last four reactions). In case (i), both τ and τ' are incremented (lines t.2 and t.3), and the thread goes back to the `dec-until-destroy-polymer` instruction. In case (ii), the thread moves to line t.4, decrements Thread-Count exactly once, and moves to line t.5.

Because the number of threads equals the value of Thread-Count at the start of the loop-until-empty (line $i.4$), and because the main program counter can't proceed beyind line $i.4$ of the `transfer` function until Thread-Count is zero, all threads must eventually be turned on each of these threads must reach line $t.4$ and must decrement Thread-Count. Only then can the main program counter proceed to line $i.5$ of `transfer`. This in turn means that each thread must destroy a σ-polymer-thread. Since the number of σ-polymer-threads, $|T_\sigma|$, equals Thread-Count, all threads are destroyed (and the T_σ-polymer is empty) upon completion of `thread-transfer`.

Bottleneck reactions. In each round, the `halve(`σ`)` function decreases the length of the H-polymer by a factor of 2, starting from n initially. Each decrement or increment of the H-polymer includes a bottleneck reaction, so there are $\Theta(n)$ bottleneck reactions in total, over all rounds. The `double` function creates 2^l thread-polymers in round l, for a total of $\Theta(n)$ thread-polymers over all rounds. The `transfer` function creates 2^l threads in round l and similarly destroys 2^l threads, and copies a polymer of length 2^l, so again has $\Theta(n)$ bottleneck reactions over all rounds. The reactions in `thread-transfer` are not bottleneck reactions (except in round 1); we analyze these in the next chapter.

Chapter 5

psCRN time complexity analysis and simulation

We follow the stochastic model of Soloveichik et al [18] for well-mixed, closed systems with fixed volume V. To achieve a fixed volume, we make two new assumptions. First, that all our reactions are bimolecular, of the form $r + r' \rightarrow p + p'$, by adding a *blank* molecule B as a second reactant r' to each reaction with only a single reactant r, and as a second product p' to each reaction with only a single product p. Second, that every psCRN initializes with sufficient numbers of B such that the psCRN can reach its final committing state. In addition, we assume that all reactions have rate constant 1.

Then when in configuration $\mathbf{c}$, the *propensity* of reaction $R : r + r' \rightarrow p + p'$ is $\mathbf{c}(r)\mathbf{c}(r')/V$ if $r \neq r'$, and is $\binom{\mathbf{c}(r)}{2}/V$ if $r = r'$. Let $\Delta(\mathbf{c})$ be the sum of all reaction propensities, when in configuration $\mathbf{c}$. When a reaction occurs in configuration $\mathbf{c}$, the probability that it is reaction R is the propensity of R divided by $\Delta(\mathbf{c})$, and the expected time for a reaction is $1/\Delta(\mathbf{c})$. When the only applicable reaction is a bottleneck reaction, and the volume V is $\Theta(n^2)$, the expected time for this bottleneck reaction is $\Theta(n^2)$. Soloveichik et al [18] consider CRNs without polymers, but the same stochastic model is used by Lakin et al. [16] and Qian et al. [17], where the reactants r or r' (as well as the products) may be polymers.

Expected time complexity of Algorithm 1: Sequential-n^2-psCRN. This psCRN has n rounds, with $\Theta(n)$ instructions per round; for example, the copy of length n in each round has n `inc` instructions. So the total number of instructions executed, over all rounds is $\Theta(n^2)$; moreover, there are $\Theta(n^2)$ `inc`(σ) instruction overall. The program's instructions execute sequentially, that is, the ith instruction completes before the $(i+1)$st instruction starts, so the total expected time is the sum of the expected times of the individual instructions. Each instruction involves a constant number of reactions. Some instructions involve bottleneck reactions; for example, the push reaction of the `inc` instruction is a bottleneck reaction. So an execution of the program involves $\Theta(n^2)$ bottleneck reactions. Each of these takes $\Theta(n^2)$ time, so the overall expected time is $\Theta(n^4)$.

Expected time complexity of Algorithm 4: Threaded-$n2^{\lfloor \log n \rfloor}$-psCRN. We noted earlier that Algorithm 4 has $\Theta(n)$ non-threaded instructions, and in fact $\Theta(n)$ bottleneck instructions. These take expected time $\Theta(n^3)$ overall, since the time for each is $\Theta(V) = \Theta(n^2)$.

Now, consider the threaded function, `thread-transfer`. In round $l, 1 \leq l \leq \lfloor \lg n \rfloor$, `thread-transfer` has 2^l threads, and pushes $n2^l$ Y monomers on to 2^l clone Y-polymers. Since each Y-push reaction is independent and is equally likely to increment each of the 2^l Y-polymers, the expected number of molecules per polymer is n. Using Chernoff tail bounds, we can show that all polymers have length in the range $[n/2, 2n]$ with all but negligibly small probability for large n.

Chernoff Tail Bounds [10] *Consider N independent trials, each with the same success probability, such that the total number of successes is X and the expected number of successes is μ. Then for $0 < \delta \leq 1$,*
(a) $\mathbb{P}[X \leq (1-\delta)\mu] \leq \exp(-\frac{\delta^2\mu}{2})$, *and*
(b) $\mathbb{P}[X \geq (1+\delta)\mu] \leq \exp(-\frac{\delta^2\mu}{3})$.

For a given polymer, let X be the number of increments to that polymer. For the upper tail bound, and setting $\delta = 1, \mu = n$ we have:

$$\mathbb{P}(X \geq (1+\delta)\mu) \leq e^{-\frac{\delta^2}{2+\delta}\mu} = \mathbb{P}(X \geq 2n) \leq e^{-\frac{n}{3}}. \tag{5.1}$$

Since this bound holds for any given polymer, and since there are at most n polymers, the probability that some polymer has length greater than $2n$ is at most $ne^{-\frac{n}{3}}$, which is exponentially small in n.

For the lower tail bound, setting $\delta = 1/2, \mu = n$ we have:

$$\mathbb{P}(X \leq (1-\delta)\mu) \leq e^{\mu\delta^2/2} = \mathbb{P}(X \leq n/2) \leq e^{-\frac{n}{8}}, \tag{5.2}$$

and so the probability that all polymers have length at least $n/2$ is at least $1 - ne^{-\frac{n}{8}}$.

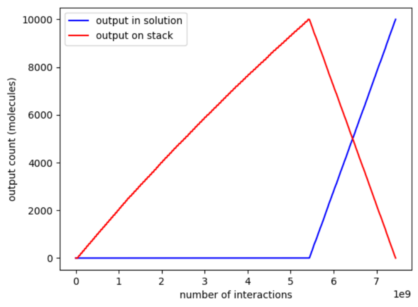
Assume that all polymers have length in the range $n/2$ and $2n$. During the first $\geq n/2$ of the `thread-transfer` decrements in round l, the count of each of the reactants is 2^l: one program counter per thread and 2^l polymers in total. So the expected time for each of these decrements is $\Theta(V/2^{2l})$. Pessimistically, all decrements would happen to the same polymer, whose length could be as little as $n/2$ by our assumption above, and so there are $2^l - 1$ polymers and threads available for the next decrements, $2^l - 2$ polymers and threads available for the next $\Theta(n)$ decrements after that once a second polymer is depleted, and so on. On top of that, the polymers would be decremented in increasing order of length, as the later in a round a polymer is decremented, the longer each decrement is expected to take. Therefore the expected time for a polymer to be decremented when there are j polymers left is $\Theta(Vn/j^2)$. Then the total expected time for round l is $O(Vn\sum_{j=1}^{2^l}(1/j^2)) = \Theta(nV)$. Multiplying by $\lfloor \lg n \rfloor$, the number of rounds, and noting that $V = \Theta(n^2)$, we have that the total expected time for the `thread-transfer` over all rounds is $\Theta(n^3 \log n)$, given that all polymers have length in the range $n/2$ and $2n$.

When our assumption does not hold, in the worst case, all Y-increments and Y-decrements happen to a single Y-polymer. In this case each Y-decrement becomes a bottleneck reaction, taking time $\Theta(n^2)$. As the number of Y-decrements during round l is $\Theta(n2^l)$, the total number of Y-pops across all rounds must be $\Theta(n\sum_{i=0}^{\lg n - 1} 2^i) = \Theta((2^{\lg n} - 1)n) = \Theta(n^2)$. This means the worst-case time for `thread-transfer` over all rounds becomes $\Theta(n^4)$, the same as Algorithm 1.

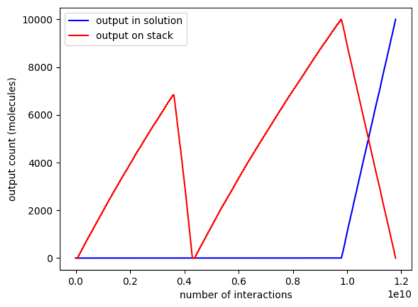
Finally, since the probability that our assumption does not hold is exponentially small in n, we have that the total expected time for the `thread-transfer` over all rounds is $\Theta(n^3 \log n)$.

Simulator. To test the correctness of our protocols, we developed a custom CRN simulator, implemented in Python, designed to support clone polymers. The simulator uses a slightly modified version of Gibson and Bruck's next reaction method [12], which itself is an extension of Gillespie's algorithm [13]. We redefine what a single "species" is from the algorithm's point of view, classifying all σ-polymers as one species, and track polymer lengths separately.

Interestingly, simulation of our stable, sequential psCRN with input detection for Square appears to take only constant factor extra time compared to the committing, fast-forwarded sequential psCRN (see both Figures 5.1 and 5.2). This is because each of the n error detection steps, and subsequent restart, is expected to happen in $O(n^2)$ time, which is negligible compared to the $\Theta(n^4)$ expected running time of the psCRN when fast-forwarding is assumed. On the other hand, comparing the runtimes of our sequential versus threaded algorithms shows that we gain significant speed from the parallelization (Figure 5.3). For all simulations, curve fitting was performed using the scipy package.

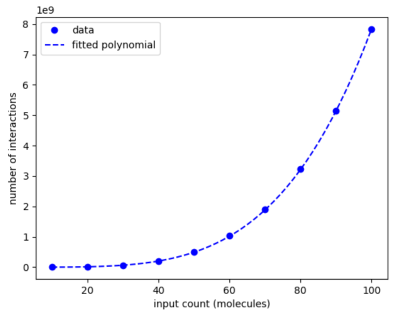


(a)

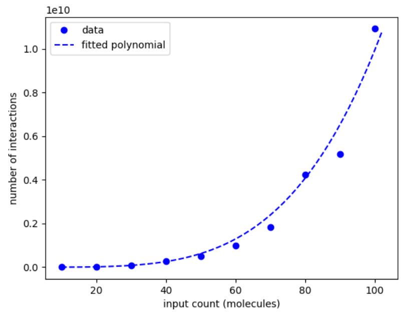


(b)

Figure 5.1: (a) Simulation of Algorithm 1: Sequential-n^2-psCRN, with fast-forwarding and input n on X-polymer, $n = 100$. (b) Simulation of Algorithm 1: Sequential-n^2-psCRN, with input detection and loading, $n = 100$. Each line in the plots shows the count of the outputs as a function of the number of interactions, with the blue line being the count of output species Y finally released into solution, while the red line shows the size of the Y_{int} polymer. By interaction, we mean a collision of two molecular species in the system, which may or may not result in a reaction.

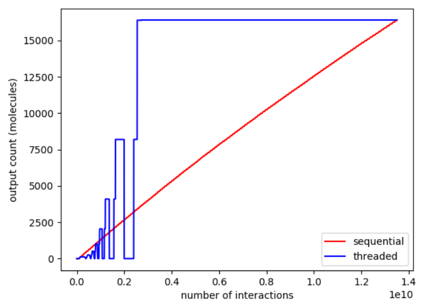


(a)

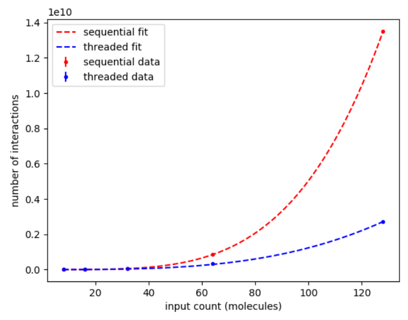


(b)

Figure 5.2: Plotting simulation runtimes as a function of initial species count n for (a) Algorithm 1: Sequential-n^2-psCRN starting with pre-loaded input (b) Algorithm 1: Sequential-n^2-psCRN with input detection and loading. Each blue marker plots a single simulation, showing the number of interactions required for protocol completion as a function of the initial count of input species X. In both cases, data was fitted with an equation of form cx^4, where (a) $c = 78.4$, and (b) $c = 99.2$.



(a)



(b)

Figure 5.3: Comparing runtimes of Algorithm 1: Sequential-n^2-psCRN (red) to Algorithm 4: Threaded-$n2^{\lfloor \log n \rfloor}$-psCRN (blue) with fast-forwarding, where (a) $n = 128$, and (b) $n = 2^i$, $i \in [3,7]$. We plot the average number of interactions over five different simulations for each input count n. Sequential data was fitted with an equation of form cx^4, $c = 50.3$. Threaded data was fitted with an equation of form $cx^3 \log x$, $c = 267$.

Chapter 6

Conclusions and future work

In this work, we've expanded the computing model of stochastic chemical reaction networks with polymers, by considering inputs that are represented as monomers in solution, as well as clone polymers that facilitate distributed data structures and threaded computation. We've shown that stable, error-free Turing-universal computation is possible in the monomer input model, by introducing an error-correction scheme that takes advantage of the ability to check for empty polymers. We've illustrated how programming with clone polymers can provide speed-ups, compared with using leader polymers only, and how leader polymers can be used for synchronization purposes by CRNs with clone polymers.

There are many interesting directions for future work. First, we have shown how to use clone polymers to get a speed-up for the Square problem, but we have not shown that such a speed-up is not possible without the use of clone polymers. Is it possible to show lower bounds on the time complexity of problems when only leader polymers are available? Or, could bottleneck reactions be reduced by a psCRN computing Square? Second, our faster psCRN for Square with clone polymers still uses leader polymers for synchronization. Is the speed-up possible even without the use of leader polymers? More generally, how can synchronization be achieved in leaderless psCRNs? Are there faster psCRNs, with or without leader polymers? It would be very interesting to know what problems have stable psCRNS that use no leaders, but can use clone polymers. Finally, it would be valuable to have more realistic models of reaction propensities for psCRN models. Underlying the model in this paper is the assumption that the extensible ends of polymers are well-mixed in solution, along with other monomeric molecules in the system. This seems implausible, in practice, since steric hindrance could isolate a long polymer's extensible end from other reactants, for example. In general, subunits that are close by on the polymer would also be close by in solution, which could increase the chances of certain interactions from happening, while decreasing the chances of other ones. Research into this area would at least entail building running simple simulations of how polymers interact with monomers and each other in solution, to build a new model that accounts for these aspects of polymer behaviour. We expect that our contributions of correctness and threading would remain relevant even with an updated model, with the focus being more sophisticated runtime analysis for psCRN programs.

Bibliography

[1] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, Mar. 2006. → page 5

[2] D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. In *Dolev S. (eds) Distributed Computing (DISC), Lecture Notes in Computer Science*, volume 4167, pages 61–75. Springer, Berlin, Heidelberg, 2006. → page 12

[3] D. Angluin, J. Aspnes, and D. Eisenstat. Stably computable predicates are semilinear. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 292–299, New York, NY, USA, 2006. ACM Press. → page 2

[4] C. Bennett. Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532, 1973. → page 4

[5] C. Bennett. The thermodynamics of computation - a review. *International Journal of Theoretical Physics*, 21(12):905–940, 1981. → page 4

[6] L. Cardelli and G. Zavattaro. Turing universality of the biochemical ground form. *Mathematical. Structures in Comp. Sci.*, 20(1):45–73, Feb. 2010. ISSN 0960-1295. → page 5

[7] I. Chatzigiannakis, O. Michail, S. Nikolaou, A. Pavlogiannis, and P. G. Spirakis. Passively mobile communicating machines that use restricted space. In *Proceedings of the 7th ACM ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing*, FOMC '11, pages 6–15, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0779-6. → page 5

[8] H.-L. Chen, R. Cummings, D. Doty, and D. Soloveichik. Speed faults in computation by chemical reaction networks. In F. Kuhn, editor, *Distributed*

Computing, pages 16–30, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-45174-8. → page 10

[9] H.-L. Chen, D. Doty, and D. Soloveichik. Deterministic function computation with chemical reaction networks. *Natural Computing*, 13(4): 517–534, Dec 2014. → pages 1, 2

[10] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952. URL http://www.jstor.org/stable/2236576. → page 31

[11] R. Cummings, D. Doty, and D. Soloveichik. Probability 1 computation with chemical reaction networks. *Natural Computing*, 15(2):245–261, 2014. → pages 2, 4, 11, 12

[12] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, 2000. → page 33

[13] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977. → page 33

[14] H. Jiang, M. Riedel, and K. Parhi. Synchronous sequential computation with molecular reactions. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 836–841, New York, NY, USA, 2011. ACM. → page 5

[15] R. Johnson and E. Winfree. Verifying polymer reaction networks using bisimulation. 2014. URL http://www.dna.caltech.edu/Papers/Polymers2014-VEMDP.pdf. → pages 5, 6

[16] M. R. Lakin and A. Phillips. Modelling, simulating and verifying Turing-powerful strand displacement systems. In L. Cardelli and W. Shih, editors, *DNA Computing and Molecular Programming*, pages 130–144, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. → pages 5, 6, 7, 22, 30

[17] L. Qian, D. Soloveichik, and E. Winfree. Efficient Turing-universal computation with DNA polymers. In *Proceedings of the 16th international conference on DNA computing and molecular programming*, pages 123–140, Berlin, Heidelberg, 2010. Springer-Verlag. → pages 2, 4, 6, 7, 12, 30

[18] D. Soloveichik, M. Cook, E. Winfree, and J. Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, Dec 2008. → pages 4, 10, 12, 30

[19] D. Soloveichik, M. Cook, E. Winfree, and J. Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, 2008. → page 1