UNIVERSITY OF CALIFORNIA
RIVERSIDE

Bridging Gaps in Programmable Laboratories-on-a-chip Workflows and MediSyn: A Modular Pharmaceutical Discovery and Synthesis Framework

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Tyson Lee Loveless

December 2022

Dissertation Committee:

Dr. Philip Brisk, Chairperson
Dr. William H. Grover
Dr. Mohsen Lesani
Dr. Manu Sridharan

The Dissertation of Tyson Lee Loveless is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I've many times heard an appropriation of Thomas Edison's quip toward completing a PhD: "a doctor is 1% inspiration and 99% perspiration" – looking back at my journey, I can't help but disagree. To be clear: this work has been far from a stroll in the park—it has stretched me in every way (except physically: apparently, countless hours of sitting can lead to degeneration of physique)—but to suggest that this is *almost entirely* the result of sheer willpower flagrantly undermines the village of people who have supported me in various ways over the years.

To start, I would like to thank my advisor, Dr. Philip Brisk: your eagerness to color with humor the somber weight that oftentimes accompanies graduate school made the tough times less so. Your patience and encouragement has been treasured. To Dr. William Grover – I can't remember a conversation with you that didn't trigger new insights or spur on helpful questions; thank you for being a constant source of encouragement and inspiration. To Dr. Mohsen Lesani: thank you for the enthusiasm exuded for everything we've worked on together; it was both an encouragement and a driving force toward delivering my best work.

To my wife, Lindsey: your (sometimes tough) love pushed me to persevere through countless difficult seasons, and your patient endurance in listening to my incomprehensible ramblings has surely inspired some of the better ideas contained herein. Thank you for standing next to me through this, and for choosing to spend your life with me.

To my firstborn, Gideon, who inspired me to pursue higher education at the start, and the rest of you, Corban, Asher, Jude, and Ezra: I cannot thank you enough – even though this has likely been as hard, if not harder, for you than it has been for me,

you've patiently and lovingly endured with me, sharing your silliness, joy, and constant eagerness to remind me in all the ways of how glad I am to be your dad.

Thank you, Mom and Dad, for cheering me on over these many years. You've always lifted me up, and your support has not gone unnoticed. To EJ and Hayle, thank you for the formative years I spent with you, and for the ways in which you've endured with me since. Our many sibling rivalries have shaped me in numerous ways, and have given me the drive to always try to be harder/better/faster/stronger.

To Jason, Shelley, and your kiddos – thank you for our cherished friendship. Through leisurely cooking all day, helping me with the grueling demands of a home that is literally falling apart, or laughing through the woes of sub-par internet during a pandemic, you have made this journey far more palatable. Jason, your eagerness to support me has been remarkable; I wouldn't have made it here if it weren't for you.

To The Cameron's: your friend-lord thanks you for your love and support. You've been—despite our best efforts to drive you away—steadfast friends and neighbors throughout this journey; your prayers and company have been life-giving.

To Skyler and Mel: despite missed opportunities when *y'all* lived closer, our burgeoning friendship has been a refreshing spring of encouragement.

To my labmates, friends, and countless others who have lent me your ear on many an occasion: thank you for your advice, encouragement, friendship, support, and overall general willingness to bear with me.

Thank you to the following entities for their generous awards, fellowships and stipends which made this research possible:

- UCR for their Provost Research Fellowship, and
- The Department of Education for their Graduate Assistance in Areas of National Need (GAANN) Fellowship.

Dedication

For Jesus Christ, my guiding light, savior of my soul, and ultimate source of joy. Your provision and direction has given me the strength to persevere in this journey; everything that comes from this belongs to You;

For Lindsey – who selflessly stood by my side, graciously pushed me forward, and helped me grapple with the many challenges graduate school has thrown our way. I couldn't have been blessed with a greater helpmate;

For my boys – Gideon, Corban, Asher, Jude, and Ezra – while patiently enduring with me as I would seemingly disappear for days on end to write or push forward a project, you've witnessed God's unimaginable grace in our lives. Never forget His faithfulness.

ABSTRACT OF THE DISSERTATION

Bridging Gaps in Programmable Laboratories-on-a-chip Workflows
and
MediSyn: A Modular Pharmaceutical Discovery and Synthesis Framework

by

Tyson Lee Loveless

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2022
Dr. Philip Brisk, Chairperson

Life scientists have a need for making their work more efficient, cost-effective, and reproducible. The ongoing reproducibility crisis underscores the need for efforts to improve and transform existing methods, and the normative 10-15 years and $2.6 billion cost to develop new life-saving drugs is harrowing.

This dissertation consists of two parts that aim to partially address these concerns: the first reviews programmable microfluidic labs-on-a-chip (pLoCs), which have been widely promised to solve issues with human error and resource waste when used for biochemical experimentation (assays). Despite touted advantages, existing pLoCs are unwieldy to operate, requiring manual translation of assays to sequences of electrode actuations to control their operation. Progress on high-level languages for pLoCs is encouraging, but back-end compiler support is lacking. This part provides solutions to fill in gaps between existing languages and pLoCs, allowing would-be adopters to translate their existing workflows to utilize these devices. Namely, it details (1) abstractions necessary for

translating and compiling assays featuring time-constrained reactions, (2) optimizations that reduce waste, decrease latency, and—perhaps most importantly—enable targeting the very small surfaces of existing devices, and (3) a strategy for statically compiling and executing assays featuring pre-compiled functions, showcased on a real-world pLoC.

The second part introduces *MediSyn*, a pharmaceutical research framework providing abstractions for building systems for discovering, synthesizing, and verifying safe drugs. *MediSyn* implements a superoptimizing search utilizing a Markov chain Monte Carlo strategy over a candidate space of drugs specified as a probabilistic context-free grammar (PCFG). Back-end modules (for candidate synthesis and evaluation) provide abstractions for connecting to remote cloud labs, local execution on pLoC(s), or manual entry when work is carried out on a benchtop. A proof-of-concept is presented as *PepSyn*, which implements two front-ends: (1) a regex-style domain-specific language (*PepSketch*) that takes inspiration from sketch-based syntax-guided synthesis, and (2) a user-interface that harnesses techniques used in natural language processing (*PepGen*) in the programming-by-example paradigm.

Contents

List of Figures

List of Tables

Part II MediSyn: A Modular Pharmaceutical Discovery and Synthesis Framework

Introduction

Life scientists are burdened with the monumental task of understanding—and, by proxy, caring for—living organisms. Their discoveries often lead to improving the quality and standard of life and have far-reaching applications from healthcare and medicine to agriculture and food production. Nevertheless, much of their work suffers from inefficiencies, astronomical costs, or is difficult to reproduce. The ongoing reproducibility crisis ([14]) reveals that much work is needed to improve, transform, or altogether displace existing methods in biological and chemical sciences, and the normative 10-15 years and upwards cost of $2.6 billion for pharmaceutical researchers to develop new life-saving drugs ([169, 50]) suggests that the *next* pandemic might come before the current one is over, mandating that we simply must do better.

In this dissertation, you will find two loosely related contributions pursuing the disruption and transformation of existing workflows for life scientists concerned with analytical biochemical protocols (or *assays*) and discovering promising new pharmaceuticals.

Part I discusses how programmable microfluidic laboratories-on-a-chip (pLoCs) can overcome some of the aforementioned issues, and presents practical optimizations and additions to workflows necessary for scientists to adopt existing and future devices. The methods are showcased with an end-to-end workflow, where assays specified in a high-

level domain-specific language are compiled to target an existing commercially-available open-hardware pLoC, revealing that sharing of protocols (in the form of completely specified executable assays or pre-compiled binaries) is tenable on these mediums.

Part II introduces *MediSyn*, a modular framework for pharmaceutical researchers to create cyber-physical systems that can automate the discovery, synthesis, and evaluation of safe *de novo* drugs. *MediSyn* aims to both reduce the exorbitant costs and the unreasonable time-to-delivery associated with drug development. *MediSyn*'s utility is evaluated with *PepSyn* a proof-of-concept implementation targeting pharmaceutical peptides. *PepSyn* compares program synthesis-inspired approaches to defining and searching over the space of peptide pharmaceuticals: *PepSketch*, a *SKETCH*-inspired[212] domain-specific-language that specifies a domain of candidate peptides using inlined wildcards in a peptide's primary sequence, and *PepGen*, a front-end in the programming-by-example (PBE) paradigm[107, 82, 188], which constructs a domain of interest using user-provided example drugs and their efficacy against a target pathogen. A superoptimization approach explores the peptide space using a Markov Chain Monte Carlo search [200, 4, 196]; the resulting Markov chains characterize a domain of drugs of interest for further exploration in a wet-lab environment.

Part I

Bridging Gaps: Making pLoC Workflows Practical

Chapter 1

Introduction

The last two decades have witnessed the emergence of software-programmable, integrated microscale machines capable of miniaturizing, automating, and accelerating fluidic biochemical laboratory experimentation (*assays*). So-called programmable *laboratories-on-a-chip* (pLoCs), these devices have promised to revolutionize biochemical experimentation in the life sciences by reducing reagent waste, increasing safety, eliminating human error, and increasing reproducibility. Despite researchers regularly validating these promises, adoption of pLoCs is still in its infancy, partly due to their relative obscurity, and to a greater extent due to their unwieldy operation. Commercially-available pLoCs (OpenDrop, DropBot, etc.) for general-purpose lab use are currently operated manually, or "programmed" through inefficient (and error-prone) processes akin to writing machine code by hand, preventing many would-be adopters from integrating pLoCs into their labs.

While high level programming languages designed for specifying assays for execution on pLoCs have made large strides in recent years, compiler support has lagged behind, and there are currently no general-purpose runtimes capable of driving physical pLoCs that utilize compiled assays written in a high level language. Additionally, though a primary objective of executing assays on pLoCs is to provide reproducible results, there is a present lack of any language or compiler support for implicit timing constraints. This is concerning, as reagents can be *extremely sensitive* to temperature changes, evaporation, and exposure to other reagents, particularly at sub-microliter volumes, leading to inaccuracies in reporting or failure in the worst case.

The rest of this part discusses and presents practical solutions to gaps that exist in the end-to-end workflow of assay specification, compilation, and execution on pLoCs. After detailing necessary background material in Chapter 2, Chapter 3 discusses necessary language and compiler additions that allow scientists to concisely express time constraints and statically ensure the synthesized results will not fail due to missed deadlines. Generalized syntactical constructs are presented that can be added to any language, and a time- and resource-constrained scheduling problem is formalized and solved using a new efficient heuristic and an optimal Integer Linear Programming (ILP) formulation. Methods are demonstrated and evaluated by adding syntax to the BioScript language [178] and compiling contrived time-constrained assay benchmarks, revealing that existing scheduling methods are in general unable to properly schedule assays when timing constraints are present.

Chapter 4 introduces necessary optimizations for successfully compiling assays onto pLoCs with severely constrained spatial resources (a reality of existing devices), exploring various ways of placing microfluidic operations with fluidic dependencies crossing basic

blocks at a global scope. Further optimizations explore trade-offs between operation-level parallelism and execution latency. In addition to accelerating benchmark assays (with an average decrease in latency of 25%), the optimizations provided a 450% increase in compilability when compiling onto severely spatially-constrained architectures.

Chapter 5 provides observations necessary for setting up, running, and tearing down assay protocols specified using (possibly pre-compiled) functions. While the vast majority of real-world assays that have been translated for execution on LoCs are easily specified using straight-line code, scientists could benefit from utilizing functions to organize complex protocols, defining repeatable operations with parameterized reagents, or the ability to disseminate pre-compiled executable protocols. We introduce a *split-technology stack*, which segregates the digital representation of a stack frame from physical droplets requiring on-chip storage that lives across function calls, an idiosyncrasy that traditional computing architectures need not consider. This chapter culminates by demonstrating execution of various protocols on *OpenDrop*, a commercially-available open-source pLoC device.

Chapter 2

Background and Related Work

2.1 Background and Related Work

2.1.1 Analytical Biochemical Protocols (Assays)

An *assay* is a laboratory procedure that aims to assess the activity of a target entity, called the *analyte*; as an over-generalization, we use the term assay to represent a biochemical "algorithm" that will execute on an pLoC. Ideally, the (bio-)chemist of the future will specify an assay using an appropriately designed high level domain-specific programming language (DSL) meant for programmable chemistry. A number of domain-specific programming languages have been proposed for pLoCs [230, 226, 6, 5, 40, 41, 178, 239] (see Section 2.1.4); while most of these languages are tied to specific pLoC technologies, any DSL compatible with DMFBs (see Section 2.1.2) could be used as a front-end for the work presented here.

Many assays are *time-sensitive*, although it may not always be obvious from the description. Reagents can be *extremely sensitive* to temperature changes, evaporation, and exposure to other reagents, especially at sub-microliter volumes [51], and can lead to inaccuracies in reporting or failure in the worst case. Benchtop-scale assays specified either formally or informally without explicit statements of timing constraints can be scheduled in a way that causes problems on a resource-constrained microfluidic format: constrained resources may lead a scheduler to store time-sensitive interactions past their usable lifespan. To mitigate these concerns, Chapter 3 introduces language constructs and compiler support in order to support and enforce these constraints; it formalizes the time- and resource-constrained scheduling problem for assays specified to run on pLoCs, and implements an efficient heuristic and optimal Integer Linear Programming model for solving the problem.

2.1.2 Digital Microfluidic Biochips (DMFBs)

The compiler described in this paper targets a class of pLoCs called *Digital Microfluidic Biochips (DMFBs)*, which manipulate discrete droplets of fluid using electrostatic actuation [130, 168]. DMFBs exploit a physical phenomenon called *electrowetting*, shown in Fig. 2.1: an electrostatic potential applied to a droplet at rest modifies its shape and angle of contact with the surface, causing it to spread. DMFBs exploit this property to facilitate programmable droplet manipulation: droplet transport can be achieved by activating and deactivating adjacent electrodes in sequence, as shown in Fig. 2.2. An optional top "ground electrode" reduces the voltage required to move a droplet and improves the fidelity of on-chip operations.

A DMFB is a programmable 2-dimensional array of individually addressable electrodes (Fig. 2.3) which supports an instruction set consisting of five basic operations: store (hold droplet at position (x, y)), transport (move a droplet from position (x, y) to position (x', y')), merge (combine two droplets), mix (combine two droplets and route them in a rectangular motion), and split (separate a larger droplet into two roughly equal smaller droplets) (Fig. 2.4) [187, 167, 75, 174, 86, 1]. An "executable program" is a sequence of electrode activations supplied by a host PC or microcontroller. A compiler translates a text-based assay specification into an executable program [41, 178]. A DMFB is "reconfigurable" in the sense that each operation can be performed anywhere on the electrode array and any given electrode may contribute to different operations at different points in time during execution. A typical DMFB will integrate non-reconfigurable resources such as I/O reservoirs on its perimeters, as well as heaters [137], sensors [190, 140, 39, 223, 125, 206, 197, 121, 68, 199, 20, 170, 205, 126, 1], optical detectors [218, 135, 136, 236], or online video monitoring [207, 16, 93, 65, 233, 127] into the array itself.

Integration of sensors [190, 140, 39, 223, 125, 206, 197, 121, 68, 199, 20, 170, 205, 126, 1] and online video monitoring [251, 138, 139, 92, 103, 2, 98, 99, 3, 184, 100, 127] allows a host PC or microcontroller connected to a DMFB to obtain online feedback regarding the state of the assay during execution, and facilitates cyber-physical control. At the language design level, this provides targets for control flow: arbitrary computations can be performed on acquired sensory data, including predicates that resolve conditions at run-time [81, 41]. Feedback-control has been applied for precise droplet positioning [206, 170, 20, 126, 16, 65, 93, 233, 127, 1], online error detection and recovery [251, 138,

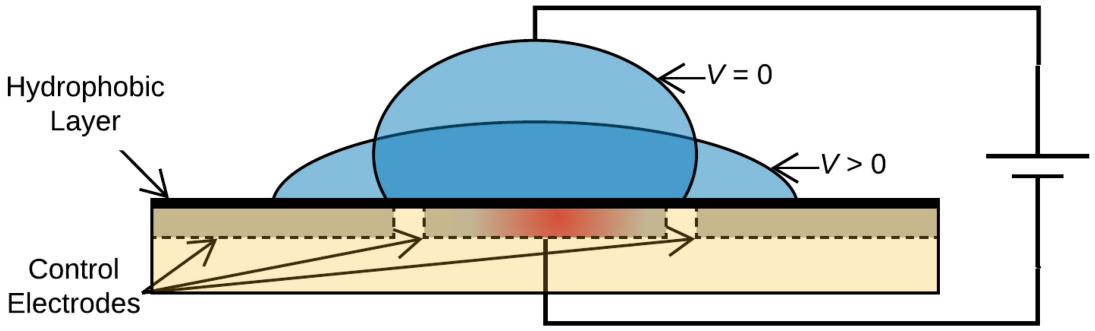


Figure 2.1: The electrowetting effect: applying an electrostatic potential to a droplet modifies its contact angle [130, 168].

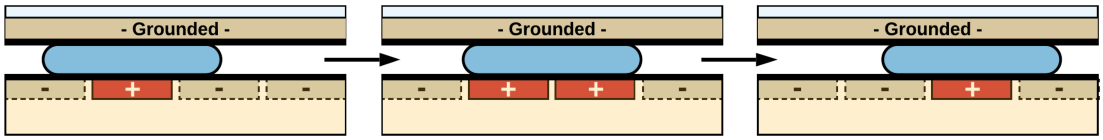


Figure 2.2: Droplet transport is achieved by activating and deactivating electrodes in sequence.

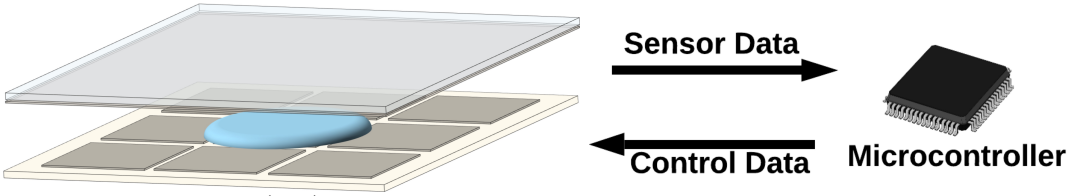


Figure 2.3: A DMFB (left) comprises a 2D array of discrete electrodes, with an optional ground electrode on top. A cyber-physical feedback loop to a microcontroller is enabled by sensory feedback.

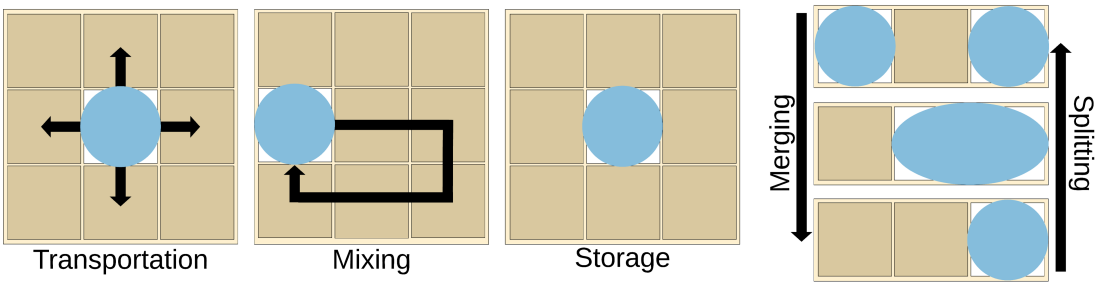


Figure 2.4: DMFB instruction set architecture (ISA).

139, 92, 103, 2, 98, 99, 3, 184, 100, 127], and to provide control flow constructs at the language syntax level [81, 40, 41, 178].

2.1.2.1 `OpenDrop`

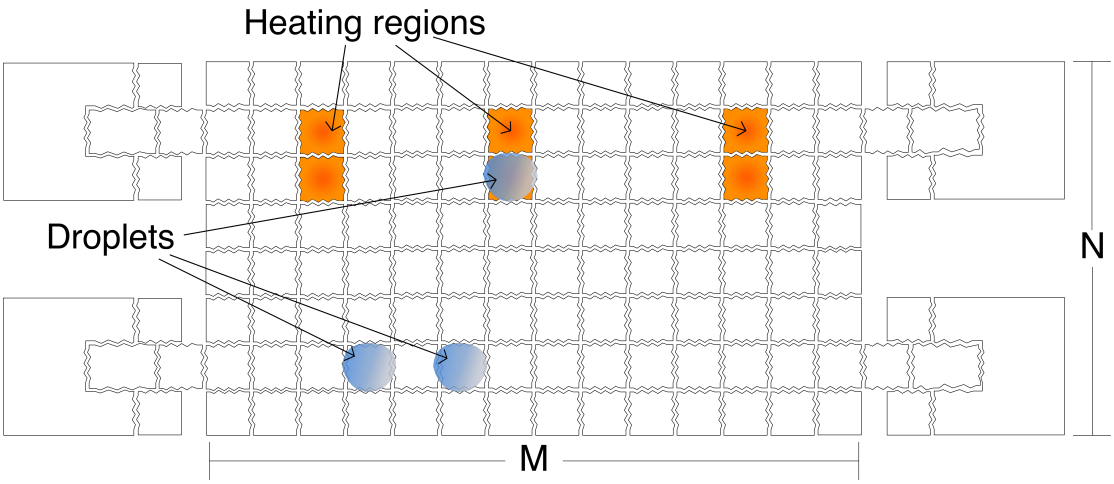
`OpenDrop` (Figure 2.5a) is an open-source (hardware/software) DMFB that features a modular cartridge design. Cartridge designs with integrated heaters are available,[1] and custom cartridges supporting optical detection or varied electrode patterns for application-specific usage have been demonstrated.[2] `OpenDrop` is programmed through an open-source graphical interface through which the user manually turns electrodes on and off over time; an integrated joystick on the device allows for real-time user-control, but can only actuate one electrode at a time. Chapter 5 utilizes `OpenDrop`'s electrode layout for presenting technical concepts, and culminates with proof-of-concept execution utilizing the methods presented throughout this part.

2.1.3 DMFB Compilation

Compilation of assays lacking control flow or timing deadlines is relatively mature. The input is a fluidic variation of a *directed acyclic dependency graph (DAG)*, where vertices represent fluidic operations and edges represent "fluidic dependencies," i.e., an edge between vertices u and v indicates operation u produces a droplet that is consumed by operation v. Figure 2.6 depicts the compilation path for a fluidic DAG, which must solve three interdependent NP-complete sub-problems: scheduling of operations [52, 191, 221, 80, 176, 131], reconfigurable module placement [220, 245, 240, 241, 141, 129, 37, 142, 78], and droplet routing [222, 24, 38, 246, 95, 194, 195, 112, 111].

[1] https://gaudishop.ch/index.php/product/opendrop-v4-cartridge-heated-pads/
[2] http://www.gaudi.ch/OpenDrop/?p=759, http://www.gaudi.ch/OpenDrop/?p=751

(a)



(b)

Figure 2.5: (a) An `OpenDrop` DMFB device with a 8×14 cartridge. (b) The virtual representation of the `OpenDrop`'s electrode arrangement we use in this paper.

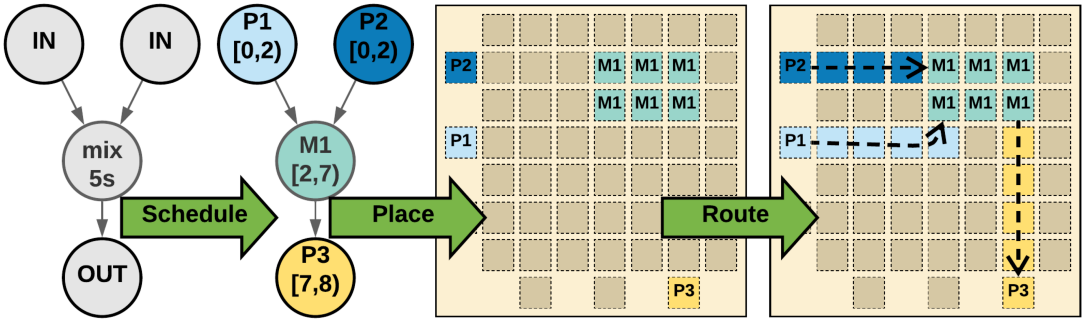


Figure 2.6: A DMFB compiler must determine when and where operations will be performed, as well as how droplets will arrive at their destinations.

The scheduler must determine exact times in which each operation starts and stops, while observing fluidic dependency constraints within the dependency graph and physical resource constraints associated with the target device. All five basic operations can be performed at the same location as e.g., a heater (when off) or a detector; however, heating and detection cannot be performed at any location on-chip. Thus, the compiler must know the precise location of all I/O pads on the device perimeter and both the location and function of all other integrated components. Once the DAG is scheduled, the placer finds physical locations where each fluidic operation will be performed at each time step. Droplet routing is tasked with transporting droplets to appropriate placed locations at the appropriate times without inadvertently colliding with other live droplets. If reagent residues may cause contamination of routing droplets, wash droplets may be introduced by the router to prepare for subsequent routing [96, 249, 242].

While DAG-level compilation is mature, compiling more complex assays represented by a *Control Flow Graph (CFG)* is still an active area of research. Early work in this context targeted online error detection and recovery for the DAG compilation model described above [251, 138, 139, 92, 103, 2, 98, 99, 3, 184, 100, 127]. With appropriate extensions to handle CFGs, these techniques could be integrated into the runtime system that executes assays compiled using the techniques described in this part on a DMFB, but is beyond the scope of this work to design and evaluate such techniques. Two techniques have been recently proposed for compiling CFGs: the first, which is orthogonal to what is proposed here, is to interpret assays online, rather than compile them offline [81, 239]. During execution, the runtime analyzes sensory data to resolve conditionals and then JIT-compiles each basic block, emphasizing compilation speed over solution quality. To the best of our knowledge, prior work has not attempted to JIT-compile an assay on

the granularity of the CFG; any such approach could build on the techniques used here, noting that the runtime overhead of mix operation resizing may be prohibitive. Further, there is a complex interplay between coalescing and module resizing (see Chapter 4), as resizing may affect interferences across the CFG during rescheduling; hence, the combination of these optimizations are not well-suited for online compilation.

The alternative approach, which we build upon in Chapter 4, statically compiles a CFG by using any of the aforementioned scheduling, placement, and routing algorithms on each individual basic blocks (fluidic DAGs), and stitches together the results afterward. We build directly upon the work in [41], which described these techniques; it introduced the hybrid computational-fluidic IR used herein, and demonstrated how to compile a CFG: each basic block could be compiled individually, with additional droplet routes inserted at control flow edges. These routes ensure that each basic block begins with its incoming droplets at the same position regardless of which control path is taken leading into that basic block. As basic blocks are considered in isolation, unnecessary droplet transport operations are introduced, leading to increased reagent residue waste, longer execution times, and possible failure to compile.

Previous microfluidic placement work has taken inspiration from spatial computing: [52, 241, 78] and have adapted algorithms originally introduced for dynamically reconfigurable FPGAs [18] to the microfluidic context. These algorithms, while practical and useful, do not assume that tasks compiled onto a dynamically reconfigurable FPGA communicate, and are thus unable to effectively reduce or eliminate droplet transport latencies when applied to a microfluidic context.

The optimizations presented in Chapter 4 propose a compiler framework that take principles from graph coloring register allocation [33, 32, 28, 70]: fluidic variables whose lifetimes overlap "interfere," and must therefore be placed at non-overlapping positions on the spatial 2-dimensional array to prevent inadvertent mixing and cross-contamination. Operations within basic blocks are defined atomically (i.e., an operation cannot start in one basic block and finish in another); further, [41] splits fluidic variable live ranges at basic block boundaries in accordance with the ϕ- and π-functions of the Static Single Assignment (SSA) [42] and Static Single Information (SSI) [7, 208, 25] Forms, localizing all interfering operations to isolated basic blocks, while ensuring a linearized def-use chain over all variables. While [41] ensures that fully correct CFG compilation can be achieved by considering basic blocks in isolation, and inserting inter-block routes at control edge boundaries, producing a correct "executable program," but, as noted previously this does nothing to mitigate spurious droplet transport operations, and may induce additional wash droplet requirements. By ensuring that fluidic operations o_i and o_j having a non-interfering dependency edge (o_i, o_j) in the CFG are placed at the same location, then otherwise necessary routing is eliminated entirely. This is identical, in principle, to coalescing operations performed during register allocation; moreover, as SSA(SSI) Form may introduce many copy operations at block boundaries, the techniques can have significant effects on routing requirements between blocks.

In short, the strategies presented in Chapter 4 derive (both in principle and literally) techniques from graph coalescing to minimize the number of transport operations that the fluidic SSI Form from [41] inserts into the CFG. Additionally, when droplet transportation requirements persist after placement, the placement mechanisms attempt to minimize overall transport distances required by incorporating a static estimate of how critical

the transport operation is to overall assay execution time — i.e., by placing interfering, but dependent, operations *near* one another, the transport routing can be effectively minimized.

2.1.4 Language Design for Programmable Chemistry

Languages designed for programmable chemistry, including those intended for programming DMFBs, can generally be classified into one of three categories: laboratory automation, ontologies, or domain-specific languages. [198] provides a detailed review of historical language development for programmable chemistry. The following is a brief selection and description of some notable languages:

2.1.4.1 Ontologies

An ontology in synthetic biology and chemistry aims to standardize how scientists discuss and disseminate information. Notable ontologies are the *Synthetic Biology Open Language (SBOL)* [67] and *EXACT* [213]; by describing experiments and models in a common language, scientists are able to concisely define properties and relations between concepts, data, and entities involved in an experiment. Despite their aim, ontologies are unable to directly execute experiments, and thus lack the ability to enforce reproducibility within their design.

2.1.4.2 Laboratory Automation

Lab automation workflows aim to enforce reproducibility and remove human-error by coupling standard inventories with formal/informal operational statements. [114] specifies and composes processes from individual protocols, which are then parallelized and scheduled on available laboratory equipment. In principle, inventories could be

supplemented to include pLoCs programmed using available domain-specific languages. Cloud-based automation provides scientists the ability to remotely execute assays in laboratories controlled by robots over the Internet. Assays are specified using lab- and domain-specific languages, such as *Transcriptic*'s *Autoprotocol* [3] or *Synthace*'s *Antha* [4]. Similar to [114], these processes could be extended to facilitate operation of pLoCs, yet would still require interaction with an appropriate high-level language.

2.1.4.3 Domain-Specific Languages for pLoCs

BioStream [230, 226] targeted a pLoC designed primarily for serial dilution protocols, which pair fluidic mixers with fluidic memories. Now defunct, *BioStream* promised ease of use by abstracting away architecture-specific details from the programmer and including a set of algorithms that would automatically generate the dilution protocols from a set of user-set target concentrations.

Aquacore [6, 5] is a pLoC technology comprised of a collection of microvalve-based components connected to a centralized control bus, and is programmed using the assembly-like *AquaCore Instruction Set (AIS)*. Other high-level languages could be adapted to output AIS with compatibility with Aquacore's components, but this is out of the scope of this part.

Originally an ontology, *BioCoder* [8] was later extended to target pLoCs [150, 81, 40, 41]. While a significant first-of-its-kind proof-of-concept in the direction of providing high-level access to programming LoCs, the syntax was not intuitive, and it lacked any formal semantics.

[3] http://autoprotocol.org
[4] https://docs.antha.com

Puddle [239], while not technically a domain-specific language, is an ongoing effort to implement domain-specific APIs for read-eval-print-loop (REPL) programming of a custom-designed pLoC. *Puddle*'s novelty is in its simplicity of interleaving fluidic operations within a traditional language's existing infrastructure through custom APIs. As an API library, high-order concepts such as arrays, unbounded looping, and functions using fluidic types are supported for free. Despite the relative simplicity of using, e.g., Python as a front-end, *Puddle*'s single-architecture target and choice to embrace complete dynamism at the expense of static guarantees makes reasoning about fluidic types and providing verifiable contracts (e.g., support for timing-constrained protocols) untenable.

BioScript [178] serves as a long-term replacement for *BioCoder*. It features a chemical safety type system that warns programmers when potentially dangerous chemical reactions may take place in a provided program, and formalizes its semantics for provably correct operation. *BioScript* further shines in its syntactical constructs, where very-high-level operations are specified in a cookbook style, enabling life scientists, who are not typically trained in the idiosyncrasies of typical programming languages, to effortlessly and concisely specify assays for use on pLoCs.

Work presented in this part leverages and builds upon *BioScript* for all proof-of-concept implementations. Table 2.1 lists the operations *BioScript* supports and the corresponding syntax, categorized by microfluidic technology complexity. The core of the language consists of generic fluidic operations common to all LoCs, including declarations of fluidic variables, mixing of fluids, and storage. Notably, to reduce the programming burden and to keep the language small, certain constructs are omitted from the syntax that can be directly inferred by the compiler or execution engine (transportation, storage). *BioScript* enables programming control-flow constructs for pLoCs having integrated

sensors with conditional branches and `while` loops, as well as DMFB-specific operations for reading and writing to integrated modules (detect, heat) and droplet splitting. The detect instruction abstracts the reading from the hardware implementation, and can be used for various read tasks: e.g., measuring temperature, mass, or fluorescence, etc. While declaration of (non-recursive) functions is supported, *BioScript's* compiler inlines all function calls due to the execution of functions on pLoCs being non-trivial (Chapter 5 addresses this).

Table 2.1: Operations supported by *BioScript*

Target	**Feature**	**Syntax**
Core	Material Decl	`manifest A`
	Input	`a = dispense 5 units of A`
	Output	`drain ab`
	Mix	`ab = mix a with 10 units of B for 10s`
	Transport	`/* inferred through routing */`
	Store	`/* inferred through scheduling */`
	Repeat	`repeat n times { ... }`
Control Flow	Branch	`if (...) { ... } else { ... }`
	Loop	`while (...) { ... }`
DFMB	Detect	`detect temperature on ab`
	Heat	`heat ab at 30c for 10s`
	Split	`x = split ab into 2`

Figure 2.7 depicts a simple assay written in *BioScript*. The *BioScript* syntax allows the programmer to specify the latency of assay operations (e.g., "`... for 1m`"), but not timing constraints on the intermediate products (fluids) that are generated and later consumed (Chapter 3 addresses this limitation). *BioScript* provides the programmer with abstractions for acquiring sensory data from the DMFB, processing that data, and making control flow decisions based on the result of the computation.

```
1       /* inputs: sample, reagent */
2       analyte = mix sample with reagent for 1m
3       heat analyte at 80c for 2m
4       result = detect analyte for 30s
5       drain analyte
```

Figure 2.7: A simple assay written in *BioScript*.

2.1.5 Mixing Modules

Mixing fluids together is a common operation in any assay; the latency of mixing two fluids depends on the number of electrodes that have been allocated to perform the mixing and also the routing path that the droplet takes within the mixer [179] (see Table 2.2). While larger mixers yield lower latency, they reduce the availability of spatial parallelism on-chip. Several prior papers for single basic block compilation effectively co-optimized scheduling and placement to account for different mixing latencies and their associated resource consumption [240, 241, 141]. In contrast, the compiler additions described in Chapter 4 includes a feedback loop that adjusts the size of different mixing operations in order to optimize performance by exploring trade-offs between increasing operation-level parallelism and/or reducing individual operation latencies. Although not discussed, these techniques could be utilized in order to meet assay deadlines, as well (see Chapter 3).

Table 2.2: Mixing module dimensions and their latencies [179].

Mixing Dimensions	Mixing time (sec)
2×2	9.95
2×3	6.61
1×4	4.6
2×4	2.9

Chapter 3

Language and Compiler Support for Time-Constrained Chemistry

3.1 Introduction

Today's programming languages and compilers support *timed operations* (e.g., `mix` or `heat` for s seconds) but **do not** support *timing constraints* on the intermediate fluids that are produced and subsequently consumed during assay execution; for example, it may be necessary that a fluid *generated* at time t must be *consumed* before time $t + \Delta$ to prevent spoilage. Existing compilers aim to minimize assay execution time [177], but do not enforce timing constraints. When available on-chip resources are scant, the compiler may temporarily store a time-sensitive intermediate product beyond its usable lifespan.

To address these concerns, this chapter makes the following contributions:

- It introduces syntactical annotations that can be added to any microfluidic programming language to allow specification of timing constraints for fluids; the annotations are added to the `BioScript` language [178] as a proof-of-concept.

- It reformulates the resource-constrained scheduling problem for compilers targeting *Digital Microfluidic Biochips (DMFBs)* and presents a new heuristic and an optimal *Integer Linear Programming (ILP)* formulation to solve this problem.

- It presents two benchmark suites to evaluate a scheduler's ability to solve the updated scheduling problem, and compares existing scheduling methods against those presented here, demonstrating their effectiveness

The remainder of this chapter is organized as follows: Section 3.2 describes the motivation for timing constraints, and introduces syntactical annotations to associate them with operations, culminating with a proof-of-concept implementation in `BioScript`. Section 3.3 specifies the scheduling problem for DMFBs and introduces scheduling constraints; Section 3.4 presents heuristic and optimal algorithms to solve the scheduling problem; Sections 3.5 and 3.6 summarizes the benchmarks used in our study and presents an experimental evaluation of our proposed methods; finally, Section 3.7 concludes the chapter and outlines potential directions for future research.

3.2 Timing Constraint Annotations

This section introduces and characterizes six types of timing constraints on the usage of intermediate fluids that the designers of an assay may want to include in the assay's specification. To express these constraints, syntactic constructs are presented in the form

of annotations. As a proof-of-concept, the annotations are added to the syntax of the `BioScript` language.

Table 3.1: Time constraint variants on the intermediate fluid between operations u and v, which is generated when u completes at time t; syntax associates a constraint with u, applying it to all (u, v) dependencies

Constraint	**Description**	**New Syntax**
SLE Δ	v must begin no later than $t + \Delta$	@use.in Δ
SGE Δ	v must begin no earlier than $t + \Delta$	@use.after Δ
SEQ Δ	v must begin at $t + \Delta$	@use.at Δ
FLE Δ	v must complete no later than $t + \Delta$	@finish.in Δ
FGE Δ	v must complete no earlier than $t + \Delta$	@finish.after Δ
FEQ Δ	v must complete at $t + \Delta$	@finish.at Δ

3.2.1 Timing Constraints

Assays that feature time-sensitive reagents must include timing constraints to guarantee correctness. The designer of such an assay must be able to specify these constraints unambiguously in a manner that is both readable by humans and executable or interpretable by machines. Table 3.1 summarizes six timing constraints that can be applied to an intermediate fluid *produced* by operation u and *consumed* by operation v; these constraints are associated with the edge (u, v) in the associated DAG.

Table 3.2: Fluidic operations and their associated latencies

Operation	**Latency (in t time-steps)**
input	2
output	0
split, *merge*	2
heat, *sense*	programmer-specified
mix	programmer-specified†

† – Ref [179] associates a mixer's *dimensions* with concrete latencies for homogenization; Ref [133] adjusts specified latencies based on the chosen module size.

The constraints in Table 3.1 are categorized based on how they relate to the consumption of the intermediate fluid; in each constraint Δ is a non-negative value that represents the amount of time specified in the constraint. Constraints starting with S (start) links Δ to the start of the operation v consuming the droplet, while those starting with F (finish) are associated with the completion of operation v. *LE*, *GE*, and *EQ* correspond to the $\leq, \geq$ and $=$ comparison operators.

The full set of constraints *can* be expressed by either S or F groupings. For example, an SLE constraint could enforce an FLE constraint by incorporating the latency of v into the value of Δ.

The new syntax in the final column of Table 3.1 attaches an annotation to an operation u which generates a droplet d; the associated constraint is then applied to edge (u, v) where v is an operation that consumes d. Split operations may be inserted after u if multiple operations will consume the fluid produced by u.

```
1   /* inputs: a, b, c, d, e */
2   @use.in 30s
3   ab = mix 1 units of a with 1 units of b for 15s
4   @finish.at 37s
5   cd = mix 1 units of c with 1 units of d for 10s
6
7   t_e = dispense 2 units of e
8   temp_e = split t_e into 2
9
10  @finish.in 15s
11  heat temp_e[0] at 30c for 15s
12
13  @finish.after 10s
14  heat cd at 90c for 35s
15
16  @use.at 5s
17  cde = mix cd with temp_e[0] for 5s
18
19  @use.after 5s
20  abcde = mix ab with cde for 10s
21
22  abcde = mix abcde with temp_e[1] for 5s
23
24  result = detect sensor on abcde for 5s
25
26  dispose abcde
```

Figure 3.1: A `BioScript` assay utilizing all six timing constraint variants.

3.2.2 `BioScript` Example

The annotations introduced in the preceding section can be added to any language; we add them to the `BioScript` language [178] as a proof of concept. `BioScript`'s type system ensures that no fluidic variable is used more than once, enforcing the semantic that variable uses are destructive. Figure 3.1 shows a toy example assay, written in `BioScript` and featuring timing annotations: to attach a time constraint to the intermediate fluid (u, v), an annotation is inserted in the program immediately before operation u.

As described in Section 3.2.1, a timing constraint constrains the duration of time between a pair of operations – e.g., the `@use.in 30s` annotation on line 2 in Fig. 3.1 indicates there is a `SLE` constraint on the droplet referred to by ab with $\Delta = 30s$. The annotation associates the constraint with the definition of ab on line 2 and the use of ab on line 20, specifying that operation $abcde = \ldots$ on line 20 must *start* within 30 seconds of generating the droplet. *It is the responsibility of the compiler–in particular, of the scheduler–to ensure that this constraint is satisfied.*

3.3 Scheduling Problem

This section reformulates the DMFB scheduling problem to support timing constraints. Using this updated formulation, the following section will present new constraint-aware scheduling algorithms that can be integrated into a compiler. Table 3.3 lists the notation used throughout this section; Table 3.4 lists the constraints that characterize a problem instance.

The input is a DAG $G = (V, E)$ and a description of the relevant physical parameters of the DMFB architecture; details about the latter are deferred until Section 3.3.3. Each operation $u \in V$ has a latency $L[u] \in \mathbb{Z}_+$ (Table 3.2). The scheduler computes the starting time of each operation $S[u]$. The objective is to minimize the latency of the schedule, i.e.,

$$Objective : \min \left\{ \max_{u \in V} F[u] \right\} \tag{3.1}$$

where $F[u]$ is the finishing time of operation u:

$$F[u] = S[u] + L[u] \tag{3.2}$$

3.3.1 Precedence Constraints

Precedence constraints are fundamental to scheduling. An edge $(u, v) \in E$ means that operation u produces a droplet that operation v will consume; v cannot commence until u completes, i.e.:

$$S[v] \geq F[u], \forall (u, v) \in E. \tag{3.3}$$

3.3.2 Fluidic Identifiers (Types)

Scheduling is built on top of a set of identifiers, which we call *types* (not to be confused with *type systems* [178]). Each operation $u \in V$ has a *type* denoted $T[u]$. Likewise, each physical resource in a DMFB can execute assay operations of at least one type.

Each input reservoir on the perimeter of a chip can supply one specific fluid; each output reservoir collects one specific fluid. We treat each uniquely named fluid as its own type. The sets of m and n input and output types are $\mathfrak{I} = \{i_1, \ldots, i_m\}$ and $\mathfrak{O} = \{o_1, \ldots, o_n\}$.

The electrode array on the DMFB surface performs a set of *reconfigurable* operation types: $\mathfrak{R} = \{split, store, mix, merge\}$ (Fig. 2.4); droplet transport is implicit, and is not included as a type. Operations such as *heat* and *sense* that require external modules can be treated as unique types as well.

$\mathfrak{T} = \mathfrak{R} \cup \mathfrak{I} \cup \mathfrak{O} \cup \{heat, sense\}$ is the set of types. $\mathfrak{T}$ can be extended, for example, if a DMFB integrates multiple sensor types.

3.3.3 DMFB Architecture

Next, we describe our representation of the DMFB architecture. The number of reservoirs (on the perimeter of the chip) of a given input type i_f or output type o_f are denoted as N_{i_f} and N_{o_f}, respectively, where f is the type of the fluid. To ensure that the scheduling problem is tractable, we employ a *virtual topology* (Fig. 3.2), which partitions the DMFB into a set of N *work modules* based on its dimensions. Absent the virtual topology, the scheduler must effectively solve placement (itself NP-complete) to determine if a schedule is legal; the virtual topology enables schedule legality to be specified via integer linear constraints.

All work modules can execute *reconfigurable* operations of type $\mathfrak{R}$. Work modules overlapping regions of the chip featuring integrated sensors or heaters can additionally perform operations of type *sense* and *heat* respectively; the number of work modules having these capabilities are denoted $N_{sense} \leq N$ and $N_{heat} \leq N$, respectively. A work module can store up to k droplets, depending on its size; at most $N \times k$ droplets may be stored at the same time on the chip.

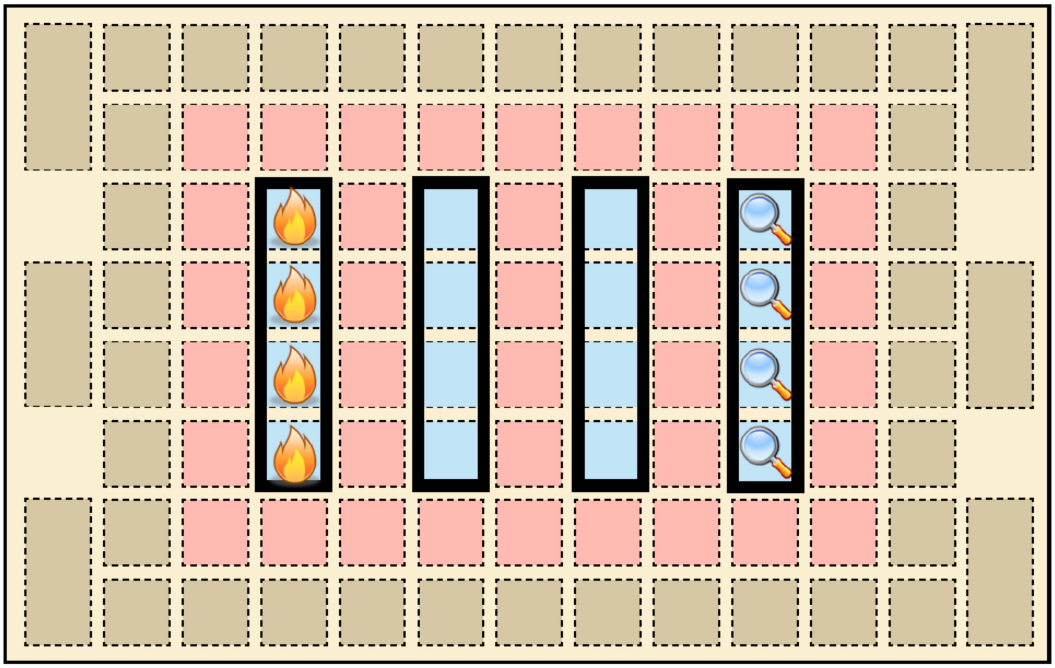


Figure 3.2: Virtual topology (VT): to ensure placing and routing feasibility, DMFB resources are partitioned into a set of *work modules* capable of performing all instruction in Fig. 2.4, with interference-free streets reserved for droplet transport. The VT for this 11x8 DMFB exposes 4 work modules, with one capable of heating and one capable of sensing. Interference regions completely surround each work module to prevent inadvertent merging of droplets.

3.3.4 Resource Constraints

Let $X[j,t]$ be the number operations of type $j \in \mathfrak{T}$ scheduled at time t. The following *resource constraints* must be satisfied at each time-step t:

$$X[i,t] \leq N_i \qquad \forall i \in \mathfrak{I} \tag{3.4}$$

$$X[o,t] \leq N_o \qquad \forall o \in \mathfrak{O} \tag{3.5}$$

$$X[heat,t] \leq N_{heat} \tag{3.6}$$

$$X[sense,t] \leq N_{sense} \tag{3.7}$$

$$X[j,t] \leq N * k - 1 \qquad \forall j \in \mathfrak{R} \cup \{heat, sense\} \tag{3.8}$$

Equations (3.4) to (3.7) ensure that the number of scheduled operations at time t does not exceed the number of available resources that can execute those operations; Eq. (3.8) limits droplet storage to prevent deadlock [77].

3.3.5 Timing Constraints

Let $\mathfrak{C}$ be the set of all timing constraint types listed in Table 3.1 and let $E_t \subseteq E$ be the set of time-constrained edges. Each edge $(u,v) \in E_t$ is labeled with a constraint pair $\{C[u,v], D[u,v]\}$, where $C[u,v] \in \mathfrak{C}$ is the constraint type and $D[u,v] \in \mathbb{Z}_{\geq 0}$ is the associated duration. Let $SLE, SGE \ldots$ be the subsets of edges in E_t that have the

Table 3.3: Notation

	Notation/Variable	Description
Architecture	N	# of work modules the VT partitions the DMFB into
	$N_{heat} \leq N$	# of work modules capable of heating
	$N_{sense} \leq N$	# of work modules capable of sensing
	$N_{i_f}, i_f \in \mathfrak{I}$	# of input reservoirs with type i_f
	$N_{o_f}, o_f \in \mathfrak{O}$	# of output reservoirs with type o_f
	k	# of droplets a work module can store
Operations and Types	$G = (V, E)$	DAG: V is the set of operations; E is the set of fluidic dependency edges between operations
	$\mathfrak{R} = \{split, store, mix, merge\}$	Set of reconfigurable operation types
	$\mathfrak{I} = \{i_1, \ldots, i_m\}$	Set of input types
	$\mathfrak{O} = \{o_1, \ldots, o_n\}$	Set of output types
	$\mathfrak{T} = \mathfrak{R} \cup \mathfrak{I} \cup \mathfrak{O} \cup \{heat, sense\}$	Set of all operation types
	$T[v] \in \mathfrak{T}, v \in V$	Operation type for operation v
	$L[v] \in \mathbb{Z}_+, v \in V$	Latency of operation v
	$S[v] \in \mathbb{Z}_{\geq 0}, v \in V$	Scheduled start time for operation v
	$F[v] \in \mathbb{Z}_+, v \in V$	Scheduled finish time for operation v
	$X[j, t], j \in \mathfrak{T}$	# of scheduled operations of type j at time t
Timing	$E_t \subseteq E$	Subset of fluidic dependency edges having timing constraints
	$\mathfrak{C} = \{SLE, SGE, SEQ, FLE, FGE, FEQ\}$	Set of time constraint types
	$C[u, v] \in \mathfrak{C}, (u, v) \in E_t$	Time constraint type for edge (u, v)
	$D[u, v] \in \mathbb{Z}_{\geq 0}, (u, v) \in E_t$	Time constraint duration for edge (u, v)
	$SLE, SGE, \ldots$	The subsets of edges $\in E_t$ having the matching constraint type from $\mathfrak{C}$

Table 3.4: DMFB Scheduling Problem Constraints

	Constraint	Description
Prec.	$F[u] = S[u] + L[u]$	Operation u's finish time is its start time + its latency
	$S[v] \geq F[u], \forall (u,v) \in E$	Operation v cannot start until operation u completes
Resource	$X[i_f, t] \leq N_{i_f}$, $\forall i_f \in \mathfrak{I}$	No more than N_{i_f} input operations of type i_f can be scheduled simultaneously
	$X[o_f, t] \leq N_{o_f}$, $\forall o_f \in \mathfrak{O}$	No more than N_{o_f} output operations of type o_f can be scheduled simultaneously
	$X[heat, t] \leq N_{heat}$	No more than N_{heat} heat operations can be scheduled simultaneously
	$X[sense, t] \leq N_{sense}$	No more than N_{sense} sense operations can be scheduled simultaneously
	$X[j, t] \leq N * k - 1$, $\forall j \in \mathfrak{R} \cup \{heat, sense\}$	One less than the max capacity of all modules can be scheduled simultaneously
Timing	$S[v] - F[u] \leq D[u,v], \forall (u,v) \in SLE$	Operation v must start no later than $D[u,v]$ after operation u completes.
	$S[v] - F[u] \geq D[u,v], \forall (u,v) \in SGE$	Operation v must start no earlier than $D[u,v]$ after operation u completes.
	$S[v] - F[u] = D[u,v], \forall (u,v) \in SEQ$	Operation v must start exactly $D[u,v]$ after operation u completes.
	$F[v] - F[u] \leq D[u,v], \forall (u,v) \in FLE$	Operation v must complete no later than $D[u,v]$ after operation u completes.
	$F[v] - F[u] \geq D[u,v], \forall (u,v) \in FGE$	Operation v must complete no earlier than $D[u,v]$ after operation u completes.
	$F[v] - F[u] = D[u,v], \forall (u,v) \in FEQ$	Operation v must complete exactly $D[u,v]$ after operation u completes.

associated time constraints. The timing constraints are as follows:

$$S[v] - F[u] \leq D[u, v], \forall (u, v) \in SLE \tag{3.9}$$

$$S[v] - F[u] \geq D[u, v], \forall (u, v) \in SGE \tag{3.10}$$

$$S[v] - F[u] = D[u, v], \forall (u, v) \in SEQ \tag{3.11}$$

$$F[v] - F[u] \leq D[u, v], \forall (u, v) \in FLE \tag{3.12}$$

$$F[v] - F[u] \geq D[u, v], \forall (u, v) \in FGE \tag{3.13}$$

$$F[v] - F[u] = D[u, v], \forall (u, v) \in FEQ \tag{3.14}$$

3.4 Scheduling Algorithms

As noted earlier, the DMFB scheduling problem comprises a DAG $G = (V, E)$ specifying the assay to perform, a subset of edges $E_t \subseteq E$ that feature timing constraints, and a set of parameters characterizing the DMFB architecture (see Table 3.3). The scheduler must compute the start times $S[u] \forall u \in V$ such that precedence, resource, and timing constraints are satisfied (see Table 3.4).

Prior work on DMFB scheduling decomposes the schedule on the granularity of time-steps, typically 1 second [221]; assay operations are integer multiples of the time-step. Droplet transport times are assumed to be orders of magnitude faster than assay operations, and are assumed to be zero during scheduling; transport times are computed directly by the droplet router [222], which is beyond the scope of this paper.

This section presents two approaches to our updated variant of the scheduling problem. The first is a heuristic, *Relative Interval Scheduling*, that converges quickly but is not guaranteed to find legal or optimal solutions; the second is an *Integer Linear Program*

will compute an optimal schedule, if one exists, but runs in exponential worst-case time unless it is eventually proven that $P = NP$.

3.4.1 Relative Interval Scheduling

This section presents *Relative Interval Scheduling* (RIS), an efficient heuristic. Limited pseudocode for RIS is available in Appendix C. RIS proceeds in two phases: the first phase satisfies precedence and timing constraints, and the second satisfies resource constraints. The scheduler fails if either phase cannot satisfy all constraints.

3.4.1.1 Phase 1 – Satisfying Precedence and Timing Constraints

The first phase of RIS produces an *implicit schedule*, which satisfies precedence (Section 3.3.1) and timing (Section 3.3.5) constraints, but not resource constraints (Section 3.3.4). This phase inserts storage operations into the DAG to ensure $S[v] = F[u], \forall (u, v) \in E$, ensuring that each droplet is either used or stored immediately after it is produced.

For each edge $(u, v) \in E_t$ with constraint type $C[u, v]$ and duration $D[u, v]$ (see Section 3.3.5), we consider three cases:

1 – Infeasible Edges: If $C[u, v] \in \{FLE, FEQ\} \wedge D[u, v] < L[v]$, then the constraint cannot be satisfied; no legal schedule exists.

2 – Storage Node Insertion: If $S[v] = F[u]$ violates a timing constraint, then a storage operation s of appropriate duration is inserted into the DAG between u and v.

3 – Storage Windows: Let δ be the maximum value that can satisfy the constraint should $S[v] = F[u] + \delta$. To model this possibility, we associate a *storage window*

$W[u, v] = \delta$ with edge (u, v) indicating that we can store the droplet produced by u for up to δ time-steps while satisfying the timing constraint. Additionally, we set $W[u, v] = \infty$ for each unconstrained edge $(u, v) \in E \setminus E_t$.

Storage Window Expansion: Storage operations can be inserted along DAG edges where storage windows exist. Consider an edge $(u, v) \in E$ with storage window $W[u, v] = \delta > 0$. We can *expand* $W[u, v]$ by inserting a storage operation s with latency $\lambda \leq \delta$, and then reducing $W[u, v]$ by λ. If $W[u, v]$ remains positive, then subsequent expansions may be possible; with these limits, storage window expansion cannot induce new timing constraint violations.

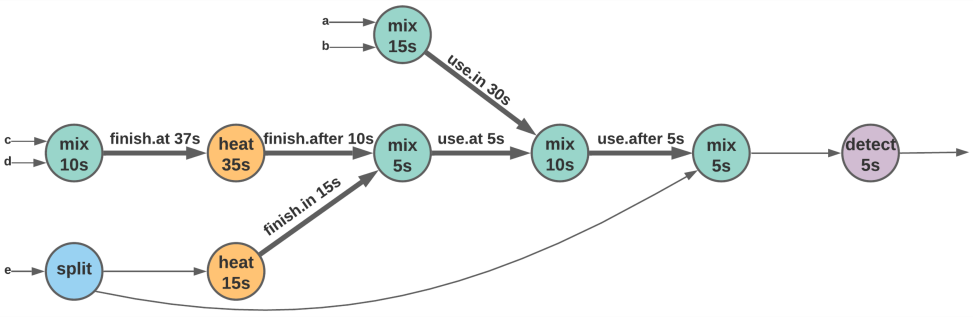
Path Length Equalization: A path P in G is a sequence of vertices that are connected by edges. The latency of P, denoted $L[P]$, is the sum of the latencies of the vertices on the path. Two paths P_1 and P_2 in a DAG are *divergent-convergent* if they share the same initial vertex u and terminal vertex v, and no other vertices. Without loss of generality, if $L[P_1] < L[P_2]$, then one or more storage nodes totaling $L[P_2] - L[P_1]$ time-steps will need to be added to P_1; this ensures that all droplets consumed by v will be present at time-step $S[v]$. This can be accomplished by applying storage window expansion on vertices in P_1, stopping when $L[P_1] = L[P_2]$. If it is not possible to do so, then a legal schedule does not exist. Equalizing the lengths of all divergent-convergent path pairs ensures that $S[v] = F[u], \forall (u, v) \in E$. We integrate path-length equalization with a call to an algorithm that computes a latency-constrained schedule of the DAG, such as as *As Soon As Possible (ASAP)* or *As Late As Possible (ALAP)* scheduling [153]; details are omitted to conserve space. We abstract this process as a function $\Omega(G)$ which returns an implicit schedule of operations in G, starting at time 0.

3.4.1.2 Relative Interval Forest

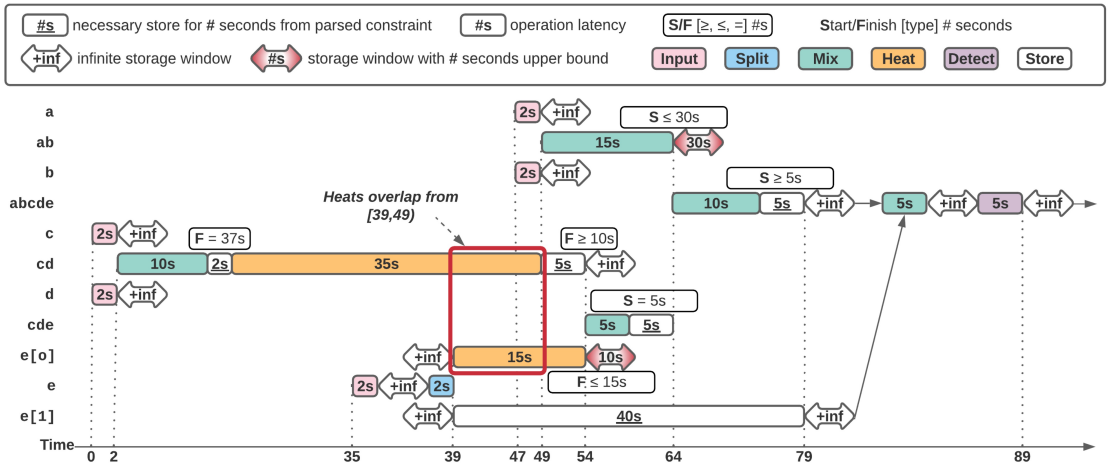
The *relative interval forest* is a data structure that encodes the implicit schedule corresponding to $\Omega(G)$ in a way that provides an efficient mechanism to test for resource constraint violations at any time-step of the schedule.

Droplet Lifetime: The lifetime of droplet d can be characterized by a path $P_d = \langle u_1, u_2, \ldots, u_m \rangle$ where u_1 is the operation that *produces* d and u_m is the operation that consumes d. The operations that produce new droplets are inputs, mixes, merges, and the immediate successors of splits; the operations that consume droplets are outputs, splits, and the immediate predecessors of mixes or merges. Sensing and heating operations do not consume or produce new droplets; a sensing operation produces data from a droplet, while a heating operation changes a physical property of the droplet.

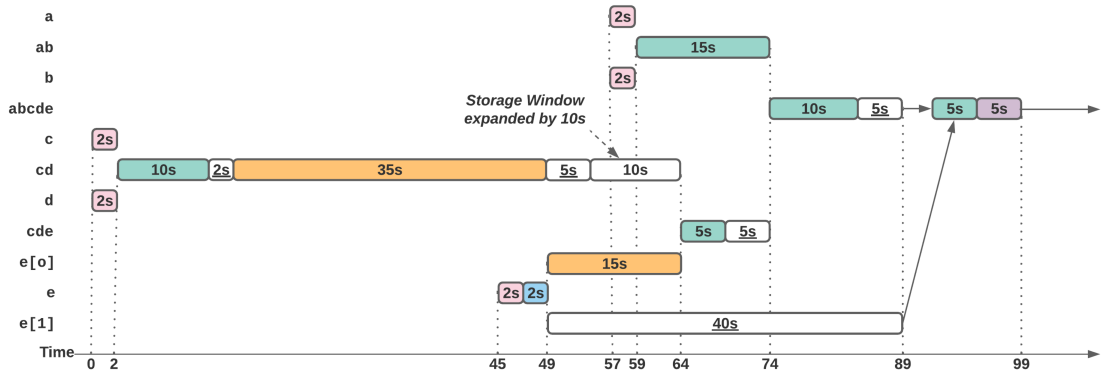
Relative Interval Tree: Path length equalization ensures that $S[u_{i+1}] = F[u_i], 1 \leq i \leq m-1$. In other words, the lifetimes of the operations that comprise the lifetime of d are a set of contiguous intervals $[S[u_1], F[u_1])[F[u_1], F[u_2]) \ldots [F[u_{m-1}], F[u_m])$ with no gaps between them. To evaluate resource constraints, the scheduler may ask if an operation of type j is processing droplet d at time t. The answer is 'yes' if there exists an operation $u_i \in P_d$ such that $S[u_i] \leq t < F[u_i] \wedge T[u_i] = j$ and 'no' otherwise. This query can be answered in $\mathcal{O}(m)$ time using a linear search of the intervals, or in $\mathcal{O}(\log m)$ time by representing the intervals with a balanced binary search tree, which we call a *relative interval tree*. We employ the latter, omitting details to conserve space.



(a) The DAG from Fig. 3.1



(b) Relative Interval Forest: droplet lifetimes are encoded as distinct interval trees within the forest



(c) Phase II: storage windows are expanded to satisfy resource constraints

Figure 3.3: Relative Interval Scheduling (RIS): (a) is parsed by RIS in Phase I to impose a schedule satisfying all timing constraints; given the architecture in Fig. 3.2, the Relative Interval Forest (b) corresponding to the schedule found in Phase I exposes a resource violation from [39, 49). During Phase II, RIS expands the storage window occurring at [54,+inf), resolving this violation.

A relative interval forest consists of a set of relative interval trees for all the variables in a DAG. A *query* $Q[j,t]$ on a relative interval forest returns the set of operations of type j scheduled to execute at time t, i.e.,

$$Q[j,t] = \{u \in V |\ S[u] \leq t < F[u] \wedge T[u] = j\}. \tag{3.15}$$

By querying the relative interval forest, it is straightforward to determine whether any of the resource constraints listed in Table 3.4 are violated at time t.

3.4.1.3 Phase 2 – Satisfying Resource Constraints

Phase 2 of RIS tries to resolve any resource constraint violations present in the implicit schedule $\Omega(G)$. It first resolves all resource constraint violations in each connected component $\gamma \in G$, as described next, then checks for violations when all γ are considered. Violations existing across connected components are trivially resolved by shifting components' schedules in relation to each other; in the worst case, components are scheduled in a way where none of them overlap, as timing constraints and fluidic dependencies are absent between them. $\Omega(G)$ returns when all constraints are satisfied.

Resolving Resource Violations: Without loss of generality, suppose that one of Eqs. (3.4) to (3.7) is not satisfied in a connected component. Let $\langle t_i, t_{i+1}, \ldots t_{i+m} \rangle$ be the maximum-length contiguous sequence of time-steps where resource constraints for type j are not satisfied, i.e., $X[j,t_\alpha] > N_j, i \leq \alpha \leq i+m$. We say that a resource violation of duration m exists between the set of droplets Θ that correspond to operations returned by queries $Q[j,t_\alpha], i \leq \alpha \leq i+m$.

The basic strategy is to expand storage windows between some of the operations in the respective lifetimes of some of the droplets in Θ, adjusting the schedule in a manner that

satisfies one or more resource constraints (i.e., reducing the value of $X[j, t_\alpha]$); however, doing so may introduce new resource violations in other parts of the DAG. Whenever a storage window is expanded, RIS recomputes the relative interval forest, iteratively lengthening the schedule until all resource constraints are satisfied. For each distinct pair of droplets $\{d_1, d_2\} \in \Theta$, we expand storage windows along particular paths in the DAG to offset their relative positions by m; there are three cases to consider:

1 – Convergent (or Divergent) Paths: If d_1 and d_2 are on paths that converge (or diverge), we expand storage windows that exist *after*(*prior* to) the operations involved in the violation.

2 – Divergent-Convergent Paths: If d_1 and d_2 are on a pair of divergent-convergent paths, we expand storage windows by an equal amount on opposite sides of the offending operations.

3 – Descendant-Ancestor Paths: If d_1 and d_2 have descendants with a common ancestor $\hat{u_a}$ that is not an ancestor of the operations that produce d_1 and/or d_2, then we expand either of the paths between $\hat{u_a}$ and the associated descendant.

If a violation of Eq. (3.8) exists within any connected component, where all droplets are related by the first two cases described above, then RIS aborts; in this case, expanding any storage windows would not reduce the number of simultaneously scheduled droplets below the threshold.

3.4.1.4 RIS Example

Fig. 3.3a depicts the DAG corresponding to the `BioScript` assay shown in Fig. 3.1. Phase 1 of RIS inserts storage operations and storage windows in the DAG, leading to the construction of the relative interval forest on $\Omega(G)$ shown in Fig. 3.3b. For example, the $finish.after$ constraint with $\delta = D[u, v] = 10s$ on the edge (u, v) between the *heat* (u) of variable *cd* and subsequent *mix* (v) of *cd* and *e* requires that $F[v] \geq F[u] + \delta$ (Section 3.2.1). As $L[v] = 5 \ngeq \delta$, phase one inserts a storage operation s into G with a latency of $\delta - L[v] = 5$ as described in Section 3.4.1.1. Additionally, as the constraint allows v to finish at *any* time later than $F[u] + \delta$, storage window $W[s, v] = \infty$ is associated on (s, v). Phase 2 of RIS then finds the resource violation of the overlapping *heat* operations over the interval $[39, 49)$ shown in Fig. 3.3b and expands $W[s, v]$ inserted in phase 1 to resolve this constraint (Fig. 3.3c). In this state, the forest reveals that no resource violations exist; hence, $\Omega(G)$ is a legal schedule of G on the given architecture.

3.4.2 Integer Linear Programming Formulation

We present an *Integer Linear Programming (ILP)* formulation of the DMFB scheduling problem with timing constraints; an ILP solver can produce optimal solutions, but will not be able to scale to large instances. The formulation derives from prior work ([177]) which does not support timing constraints.

The ILP scheduler first computes an upper bound B on the length of the schedule using any efficient (and presumably suboptimal) heuristic [221, 177]. For each $u \in V$ and $1 \leq t \leq B$, we introduce a binary variable $x_{u,t}$ which is set to 1 if u is scheduled to start at time t and 0 otherwise (Eq. (3.16)); the solver ensures that each operation is scheduled to start at exactly one time-step (Eq. (3.17)) and determines the start time $S[u]$ of each

operation (Eq. (3.18)):

$$x_{u,t} = \begin{cases} 1, & \text{if operation } u \text{ starts at time-step } t \\ 0, & \text{otherwise} \end{cases} \tag{3.16}$$

$$\sum_{t=1}^{B} x_{u,t} = 1, \forall u \in V \tag{3.17}$$

$$S[u] = t | x_{u,t} = 1 \tag{3.18}$$

The scheduling objective, operation finish times, and precedence constraints are obtained directly from Eqs. (3.1) to (3.3).

Resource constraints from Eqs. (3.4) to (3.8) are obtained directly after first counting the number of operations of each type that are scheduled at each timestep; we compute each count as follows:

$$X[\tau, t] = \sum_{u \in v | T[u] = \tau} \sum_{j = max\{1, t - L[u]\}}^{t} x_{u,j}, \forall \tau \in \mathfrak{T} \tag{3.19}$$

Storage operations are inferred from the schedule. For each edge $(u, v) \in E$, variable $z_{u,v,t}$ is set to 1 if the corresponding droplet is stored during time-step $t, S[v] > t > F[u]$ and 0 otherwise.

$$z_{u,v,t} = \sum_{k=1}^{t-L(u)} x_{u,k} - \sum_{k=0}^{t} x_{v,k} \qquad \forall (u, v) \in E \tag{3.20}$$

$$X[store, t] = \sum_{(u,v) \in E} z_{u,v,t} \qquad 1 \leq t \leq B \tag{3.21}$$

Timing constraints described previously in Eqs. (3.9) to (3.14) are modeled directly for all $(u, v) \in E_t$.

We observed that the ILP-based scheduler led to an inordinate number of seemingly unnecessary storage operations, which increased the amount of work being done by the

placer and router downstream. To compensate, we introduced a secondary scheduling objective to minimize the amount of allocated storage:

$$\text{Minimize} : \sum X[store, t], 1 \leq t \leq B \tag{3.22}$$

3.5 Benchmarks

To the best of our knowledge, benchmarks that presently exist for evaluation of microfluidic compilers and synthesis tools do not feature timing constraints akin to those introduced in this paper. We created two sets of benchmarks by adding timing constraints to existing benchmarks, all of which are DAGs. The original (unmodified) benchmarks have been widely used in prior papers that evaluate DMFB scheduling [177] without timing constraints.[1]

SLE-only consists of DAGs annotated with SLE constraints, serving to limit the amount of time that a volatile reagent could be stored before spoilage occurs; all SLE constraints were set with $\Delta = 0$ to maximally stress the schedulers. *SLE-only* is further separated into three sections: the first (unnamed) section contains *ELISA*, an immunoassay that detects various opiates in a sample, and the *Polymerase Chain Reaction (PCR)*, which amplifies DNA through replication. The *Multiplexed* section contains a multiplexed version of PCR (with 4 targets to replicate), as well as 5 variants of an *in vitro* diagnostics assay, where all combinations of the number of samples (s) and reagents (r) are evaluated. Lastly, the *ProteinSplit* assays in the *Split-Dilutes* section employ high fan-out trees to dilute a sample to varying concentrations.

[1] Listings of the benchmarks used here are available in Appendix B.2.

Mixed category DAGs were synthetically generated to stress-test the scheduler's ability to satisfy different types of timing constraints. The two versions of all_six utilize all six timing constraints introduced in this paper, while all_eq, all_finish and all_start use only the types of constraints that match their names. The last benchmark has an *infeasible* set of constraints; no legal schedule exists.

3.6 Evaluation

3.6.1 Setup

All experiments were performed on a mobile PC with an Intel® Core™ i9 processor, with 1TB SSD storage and 32GB DDR4 memory running macOS 10.15.6®. All assays were written in `BioScript` [178] using the constraint annotations introduced in Section 3.2. Each assay was compiled and passed to an open source cycle-accurate DMFB synthesis simulation framework [79] for evaluation.

We compare against four baseline schedulers available within the framework that do not account for timing constraints: **List Scheduling (LS)** [221], **Force-directed List Scheduling (FDLS)** [177], **Path Scheduling (PS)** [177], and an **ILP**-based scheduler [221, 177], which we rewrote to use the Gurobi optimizer[2]. We implemented **Relative Interval Scheduling (RIS)** within the same framework and extended the ILP scheduler to include timing constraints (**ILP-T**).

For *SLE-only*, we target a 15 × 19 DMFB architecture featuring input ports with a 2-second latency, output ports with zero latency, and 5 heating and sensing regions

[2]https://www.gurobi.com

Table 3.5: Scheduling Results and Execution Time for Each of the Scheduling Algorithms Described in Section 3.5

Suite		Benchmark	Scheduled Time (seconds)						Execution Time (milliseconds)					
			LS	**FDLS**	**PS**	**RIS**	**ILP**	**ILP-T**	**LS**	**FDLS**	**PS**	**RIS**	**ILP**	**ILP-T**
SLE-only		ELISA	Fail (1)	Fail (1)	Fail (1)	**7,600**	Fail (1)	**7,600**	-	-	-	**48**	-	333,233
		PCR	Fail (1)	Fail (1)	Fail (1)	**697**	Fail (1)	**697**	-	-	-	**4**	-	1,220
	Multiplexed	M-PCR	**1,042**	**1,042**	**1,042**	**1,042**	**1,042**	**1,042**	**14**	11,582	**14**	22	133,436	104,498
		InVitro 2s,2r	**15**	**15**	**15**	17	**15**	**15**	2	3	**1**	3	5,131	5,158
		InVitro 2s,3r	19	19	19	19	**18**	**18**	**3**	6	**3**	6	5,210	5,304
		InVitro 3s,3r	Fail (5)	Fail (4)	Fail (3)	24	Fail (1)	**20**	-	-	-	**17**	-	6,005
		InVitro 3s,4r	Fail (9)	Fail (7)	Fail (1)	27	Fail (3)	**23**	-	-	-	**36**	-	7,546
		InVitro 4s,4r	Fail (10)	Fail (7)	Fail (2)	32	Fail (1)	**28**	-	-	-	**77**	-	10,520
	Split-Dilutes	ProteinSplit1	Fail (6)	Fail (6)	64	**55**	Fail (2)	**55**	-	-	**3**	23	-	6,057
		ProteinSplit2	Fail (12)	Fail (12)	91	Fail (R)	Fail (7)	**75**	-	-	**3**	-	-	31,902
		ProteinSplit3	Fail (24)	Fail (24)	145	Fail (R)	Fail (20)	**115**	-	-	**12**	-	-	322,721
		ProteinSplit4	Fail (48)	Fail (48)	**253**	Fail (R)	Fail (56)	*Fail (T)*	-	-	**6**	-	-	-
		ProteinSplit5	Fail (96)	Fail (96)	**562**	Fail (R)	*Fail (T)*	*Fail (T)*	-	-	**14**	-	-	-
Mixed		all_six	Fail (6)	Fail (6)	Fail (6)	**100**	Fail (5)	**100**	-	-	-	**5**	-	14,692
		all_six_2	Fail (4)	Fail (4)	Fail (4)	**237**	Fail (4)	**237**	-	-	-	**2**	-	11,881
		all_eq	Fail (7)	Fail (7)	Fail (8)	**64**	Fail (7)	**64**	-	-	-	**4**	-	5,860
		all_finish	Fail (2)	Fail (2)	Fail (2)	**38**	Fail (2)	**38**	-	-	-	**1**	-	4,578
		all_start	Fail (2)	Fail (2)	Fail (2)	**64**	Fail (2)	**64**	-	-	-	**2**	-	4,352
		infeasible	Fail (1)	Fail (1)	Fail (1)	**Abort**	Fail (1)	**Abort**	-	-	-	**1**	-	150

Fail:	(T) ILP-T timed out without improving upon or confirming optimality of seeded solution		
	(#) number of timing constraints violated	(R)	RIS failed to overcome resource constraints
bold	best rest	underline	optimal schedule
Abort:	scheduler reported infeasibility and aborted	-	time not reported due to failure

available. For *Mixed*, we target the same DMFB with a single heating region, similar to the scheduling example with resource constraint violation in Fig. 3.3.

Table 3.5 reports the length of each schedule (in seconds) along with the execution time of each algorithm. For each benchmark, the shortest obtained schedule that satisfies all constraints is reported in bold; optimal solutions, as verified by the ILP solver, are underlined. Table 3.5 also reports scheduling failures:

- The baseline heuristics and unmodified ILP may find schedules that satisfy resource and precedence constraints, but not timing constraints; we report the number of timing constraint violations as $Fail(\#)$
- RIS may satisfy precedence and timing constraints, but not resource constraints; when this occurs, we report $Fail(R)$
- We give ILP and ILP-T a 4-hour timeout; if a legal schedule is not found within 4 hours, we report $Fail(T)$

We only report the execution time of successful scheduling runs; if a scheduler is able to determine that no legal schedule can be found that satisfies all timing constraints, it aborts and reports "infeasibility" which we consider to be an optimal result.

3.6.2 Simulation Results: Schedule Length

LS, **FDLS**, and **ILP** found legal schedules for 3 of the 19 benchmarks, but failed to satisfy at least one timing constraint for the others. On average, ILP generated schedules with 23% and 22% fewer timing constraint violations than LS or FDLS. Results for LS

and FDLS only differed for the 3 largest multiplexed *In vitro* benchmarks, with FDLS having 24% fewer violations for them.

PS successfully scheduled 8 of the 19 benchmarks; it was more successful than FS, FDLS, and ILP, especially in the *Split-Dilutes* subcategory, noting that PS was optimized for DAGs with high fan-out [177]. ILP-T confirmed that PS' schedules for ProteinSplit1-3 were suboptimal, but timed out for ProteinSplit4-5.

RIS succeeded in 15 of the 19 benchmarks, 10 of which were confirmed as optimal. Notably, RIS produced a shorter schedule than PS for ProteinSplit1; however, its four failures were for the larger ProteinSplit2-5 DAGs, which are all trees with high fan-out. The failures occurred as a byproduct of the process for choosing storage windows to expand when resource violations arise;determining better expansion orders is left open for future work.

ILP-T reported the optimal time for 17 of the 19 benchmarks; the two failures are due to timeout after 4 hours. In 5 of the 17 successful cases, ILP-T reported shorter schedules than the best result obtained among the heuristics.

3.6.3 Execution Time

The four heuristics ran orders of magnitude faster than ILP or ILP-T. Among the heuristics, RIS was a bit slower than LS or PS. The runtime of RIS was comparable to FDLS for all but one benchmark (M-PCR), where FDLS ran orders of magnitude slower (but still much faster than ILP or ILP-T). For the three benchmarks where ILP and ILP-T found legal solutions, their runtimes were comparable for two, while ILP ran a bit slower than ILP-T for the third (once again, M-PCR).

If a legal schedule that satisfies timing constraints is desired, a good strategy might first try scheduling using RIS, followed by scheduling using ILP-T. The good-quality schedules produced by RIS will provide a reasonable lower-bound on the schedules that can be achieved.

3.7 Conclusion

This chapter identified and provided solutions to an important problem for writing and executing assays on programmable microfluidic biochips; the key insight is that most real world assays have timing-sensitive reactions — whether listed explicitly or implied — which are unable to be specified or enforced using existing languages and compilers. Any language and compiler could adopt the methods presented here to enable fine-tuned control over timing-constrained operations at the granularity of any straight-line program with no branches or branch targets (i.e., a basic block). Future work should investigate precise methods to schedule programs satisfying time constraints across multiple execution paths.

Chapter 4

Practical Compiler Optimizations

4.1 Introduction

The emergence of laboratory-on-a-chip technologies, enabled by technological advances in microfabrication coupled with scientific understanding of microfluidics, have resulted in many experimental laboratory procedures being miniaturized, accelerated, and automated. While the bulk of microfluidic devices are essentially Application Specific Integrated Circuits (ASICs), several programmable LoCs (pLoCs) have been demonstrated [187, 230, 6, 105, 63, 5].

While recent work on programming languages for pLoCs is promising [178], gaps still exist in supporting existing architectures through compilation of programmed protocols; namely, the severely constrained spatial resources of existing pLoCs lack supporting compiler optimizations that are able to reduce latency requirements, increase operational parallelism, or even successfully synthesize a result. To address these needs, this chapter presents compiler optimizations that exploit the parallelism provided by the target

platform to execute as many concurrent chemical operations as possible, when practical, or enable compilation through module resizing when space is limited. Section 4.2 reviews the workflow for specifying and executing assays on pLoCs. Section 4.3 presents an updated formulation of the microfluidic compilation problems (scheduling, placement, and routing) in the context of assays featuring control-flow, as opposed to assays that can be represented by a single basic block. Of note, the placement problem formulation (Section 4.3.3) borrows ideas from graph coloring register allocation and spatial/data flow compilation; Section 4.3.3.1 details global placement as an optimization problem that is solved using an evolutionary heuristic, while Section 4.3.3.2 presents a unified global placement approach where a modified interference graph is placed, rather than individually scheduled operations. Section 4.4 reviews algorithms implemented. Compiler optimizations are empirically evaluated via simulation using a set of benchmark applications obtained from the scientific literature; discussion of benchmarks and evaluation is found in Section 4.5. Finally, Section 4.6 concludes the chapter and outlines directions for future work.

4.2 Overview

A basic pLoC workflow for executing assays (depicted in Fig. 4.1), consists of three parts: a front-end language compiler, a device-specific code generator, and a runtime environment. The assay is specified in an appropriate domain-specific language such as *BioCoder* [41] or *BioScript* [178][1], that seamlessly interleaves fluidic operations with computation. The target is a cyber-physical DMFB (Fig. 2.3) which provides sensory feedback to the runtime software that manages the device. This enables the programmer to specify assays featuring arbitrary control flow: the assay obtains sensory feedback from the device and performs computations on the acquired data; the result of the

[1]We utilize *BioScript* for all proof-of-concept examples

computation can be used as a condition which determines which fluidic operations to execute next.

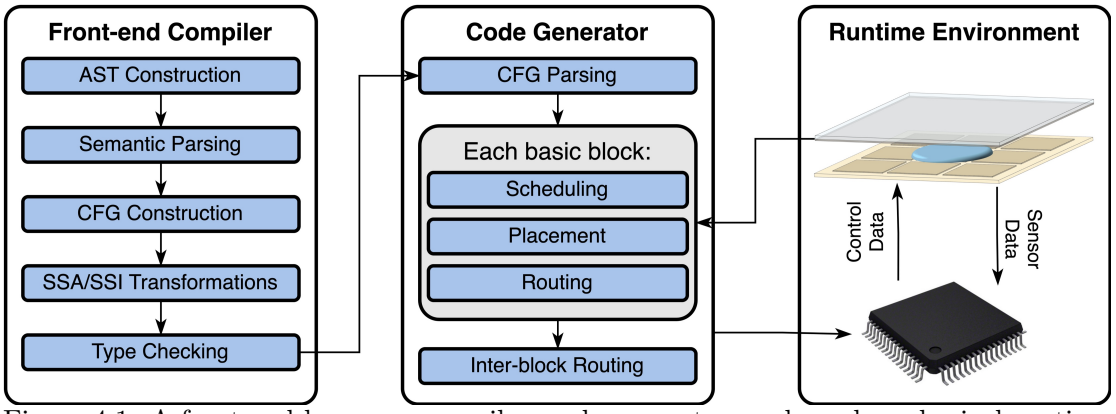


Figure 4.1: A front-end language compiler, code generator, and a cyber-physical runtime form a pLoC workflow.

Our input language supports function calls, but does not support unbounded recursion. The compiler's preprocessor inlines all function calls, which converts the assay to one procedure. The input language restricts all fluidic variables to be scalars; it does not support fluidic arrays. We hope to relax these assumptions in the future.

Figure 4.2a depicts an assay specified in the *BioScript* language [178]. After semantic parsing, we convert the assay to a hybrid computational-fluidic *intermediate representation (IR)* [41], as shown in Fig. 4.2b. This IR represents the assay as a *Control Flow Graph (CFG)*. Next, we convert both fluidic and computational variables to *Static Single Information (SSI) Form* [7, 208, 25], whereby each definition of a variable dominates each use, and each use of a variable post-dominates its definition, effectively linearizing def-use chains. Each basic block is represented as a hybrid fluidic/data dependence graph. Figures 4.2c and 4.2d respectively show the BioScript specification and hybrid-IR

converted to SSI Form: in this case, π- and ϕ- functions[2] are inserted for one fluidic variable.

Once CFG construction and SSI transformation is complete, the code generator then schedules, places, and routes each basic block in the CFG onto an abstract representation of a provided architecture. A static code generator [41] inserts inter-block routes at compile-time, while an online interpreter [81] JIT-compiles each block on demand (and must communicate directly with the runtime environment. The cyber-physical runtime is composed of the pLoC and a connected microcontroller, which processes sensory feedback produced by the pLoC, enabling e.g., control flow decisions to be made at run-time, or dynamic error detection and recovery [251, 138, 139, 92, 103, 2, 98, 99, 3, 184, 100, 127].

4.3 An Optimizing Compiler for Cyber-Physical DMFBs

Compilation begins with a typeable program, whose CFG, including fluidic variables, has been converted to SSI Form, as described above. To conserve space, we assume that the reader is familiar with SSI's ϕ- and π-functions, which respectively split variable live ranges at branch convergence and divergence points [42, 7, 208, 25].

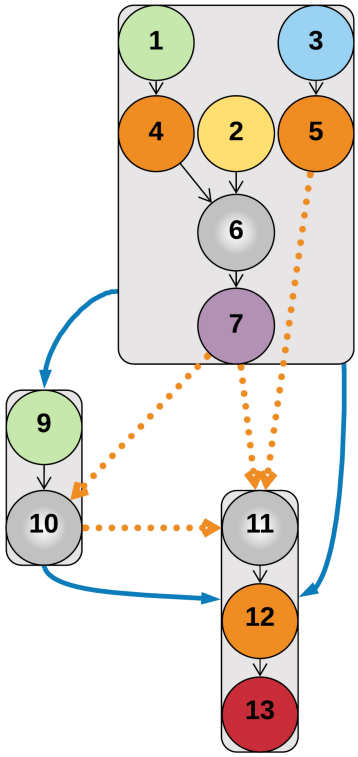
4.3.1 Scheduling

The first step is to schedule assay operations. Each basic block is scheduled individually. The scheduler ensures that each operation starts and finishes within the basic block containing it to ensure atomicity. Referring back to Table 2.2, the scheduler assumes 2×2 mixers with 9.95s latencies; this assumption is later relaxed during Rescheduling

[2] SSI's π-function (sometimes σ-function) defines a split set for a variable at the end of some basic blocks where control flow follows, allowing a unique identifier for each conditional usage of the variable in a similar way that a ϕ-function provides a single definition point for each variable.

```
1  dispense a
2  dispense b
3  dispense c
4  heat a
5  heat c
6  d = mix a with b
7  detect d
8  if (...)
9    dispense a
10   d = mix a with d
11 d = mix c with d
12 heat d
13 drain d
```

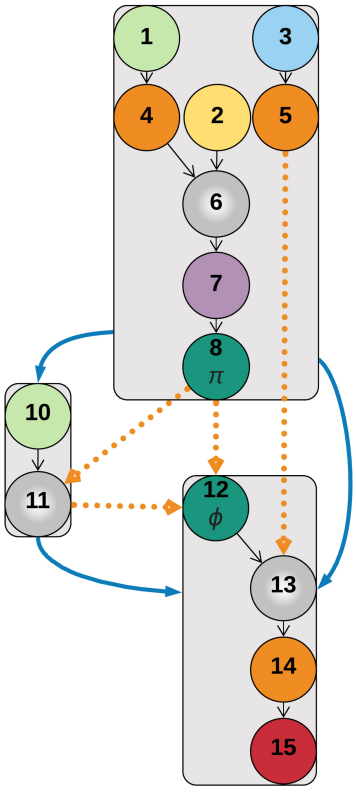
(a)



(b)

```
1  dispense a0
2  dispense b0
3  dispense c0
4  a1 = heat a0
5  c1 = heat c0
6  d1 = mix a1 with b0
7  detect d1
8  d2, d3 = π(d1)
9  if (...)
10   dispense a2
11   d4 = mix a2 with d2
12 d5 = φ(d3, d4)
13 d6 = mix c1 with d5
14 d7 = heat d6
15 drain d7
```

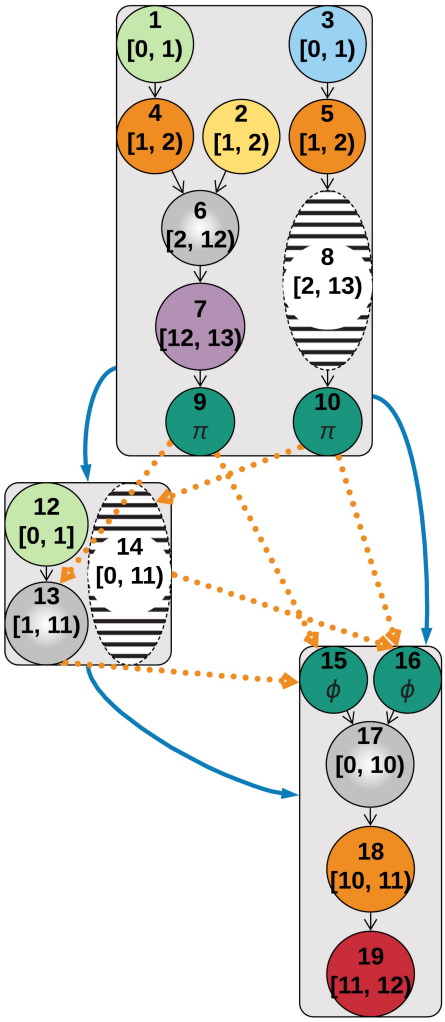
(c)



(d)

Figure 4.2: A simple assay written in BioScript (a) and the associated CFG (b) can be augmented with SSI form's ϕ and π nodes (c and d).

```
1  dispense a0
2  dispense b0
3  dispense c0
4  a1 = heat a0
5  c1 = heat c0
6  d1 = mix a1 with b0
7  detect d1
8  c2 = store c1
9  d2, d3 = π(d1)
10 c3, c4 = π(c2)
11 if (...)
12   dispense a2
13   d4 = mix a2 with d2
14   c5 = store c3
15 d5 = φ(d3, d4)
16 c6 = φ(c4, c5
17 d6 = mix c6 with d5
18 d7 = heat d6
19 drain d7
```



(a) (b)

Figure 4.3: Our scheduler adds implicit store operations (a) and updates SSI form to generate a schedule (b) that captures the linear def-use chain that SSI form provides.

(Section 4.3.3.4). O'Neal et al. [177] present the problem formulation and survey many scheduling heuristics that have been published to date.

The compiler infers droplet storage operations from the schedule and inserts them into the IR. The IR treats storage as an explicit operation that uses (and consumes) its input and defines a new output droplet. This may necessitate the insertion of additional π- and ϕ- functions to maintain SSI Form, as shown in Figs. 4.3a and 4.3b. This representation enables the placer (Section 4.3.3) to treat droplet storage the same as all other scheduled assay operations.

The scheduler enforces resource constraints that conservatively over-approximate placement. To simplify the discussion, we omit resource constraints involving I/O operations (see Chapter 3 for more detail on resource-constrained scheduling). The scheduler partitions the DMFB into a *virtual topology (VT)* of N *reconfigurable modules* (Fig. 4.4) based on a provided module size, providing deadlock free routing space around the work modules. The VT provides several benefits to placement and routing (Sections 4.3.3 and 4.3.4); most important, a legal placement and route is *guaranteed* when using the VT and supporting placement/routing algorithms. At any point in the schedule, a *reconfigurable module* can perform one mix, split, or merge operation, or can store up to k droplets, depending on its size. Any module that features an integrated heater or sensor can perform a heating or sensing operation as well; let the number of such modules be N_{heat} and N_{sense} respectively. Let $r_j(p)$ be the number of operations of type $j \in \{mix, split, merge, store, heat, sense\}$ scheduled at program point p. A legal schedule must satisfy the following constraints for each program point p:

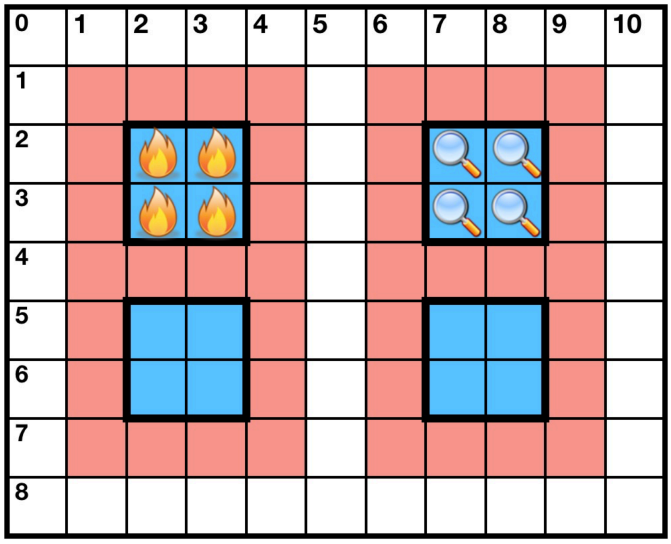


Figure 4.4: Virtual Topology: a DMFB partitioned into a 2×2 array of work modules exposed to the scheduler, where one module has a heater and one has a sensor. The topology is arranged to provide deadlock free routing around the modules.

$$r_{heat}(p) \leq N_{heat} \tag{4.1}$$

$$r_{sense}(p) \leq N_{sense} \tag{4.2}$$

$$\begin{aligned} r_{mix}(p) + r_{split}(p) + r_{merge}(p) + \left\lceil \frac{r_{store}(p)}{k} \right\rceil \\ + r_{heat}(p) + r_{sense}(p) \leq N \end{aligned} \tag{4.3}$$

Scheduling failures may occur and are unavoidable in the general case, even if the problem is solved optimally. If scheduling fails, the only option is to switch to a larger DFMB target, or rewrite the assay. During compilation, switching to larger and faster mixers Table 2.2 increases the likelihood of failure, which is one reason why we default to the smallest, slowest mixer for the initial scheduling step. Failures due to module sizes are addressed in Section 4.3.3.4.

4.3.2 Interference Graph

After scheduling, the compiler is able to construct a graph characterizing how operations relate to one another, either with interference (i.e., they should not cross paths), or with affinity (i.e., there is some dependency between operations). This section formally defines the *interference graph* and describes how it is constructed.

4.3.2.1 Definitions and Properties

Let $G = (V, E, A)$ be the *interference graph* [33, 32, 28]: V is the set of assay operations, which have already been scheduled; thus, the lifetime of each operation in V, which is contained wholly within each basic block, can be derived from the schedule. E is the set of interference edges, and A is set of affinity edges that represent fluid transfers between operations.

Let $adj[v_i]$ and $aff[v_i]$ denote the sets of interference and affinity neighbors of $v_i \in V$; additionally, let $adj^*[v_i] = adj[v_i] \cup \{v_i\}$ and $aff^*[v_i] = aff[v_i] \cup \{v_i\}$.

Each vertex is labeled with a *type*, denoted

$$type[v_i] \in \{mix, split, merge, store, heat, sense\}.$$

As shorthand, and albeit a slight abuse of notation, we define a *meta-type*, *reconfig*, as the union of types *mix*, *merge*, *split*, or *store*.

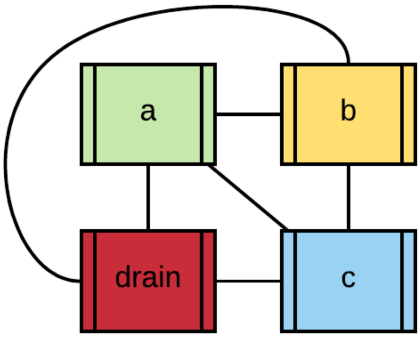
The set of interference or affinity neighbors of type t are respectively denoted $adj_t[v_i] = \{v_j \in adj[v_i] \mid type[v_j] = t\}$ and $aff_t[v_i] = \{v_j \in aff[v_i] \mid type[v_j] = t\}$; $adj_t{}^*[v_i]$ and $aff_t{}^*[v_i]$ are defined analogously to $adj^*[v_i]$ and $aff^*[v_i]$.

4.3.2.2 Construction

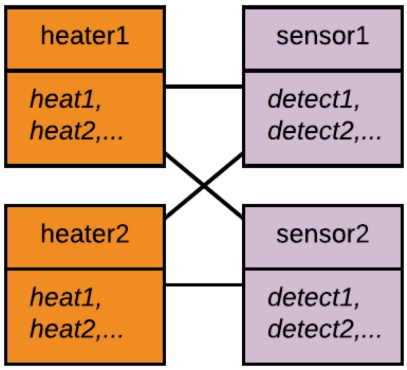
An interference edge is placed between two operations whose lifetimes overlap; in other words, for each pair of fluidic variables or operations $v_i, v_j \in V$, interference edge $(v_i, v_j) \in E$ exists if and only if the lifetimes of v_i and v_j overlap; in other words, v_i and v_j must be placed at different locations on the surface of the DMFB. Let $q_i = (x_i, y_i)$ denote the location at which operation or fluidic variable v_i is placed (for non-unit-size operations, such as 3×3 mixers, the location can be defined anywhere, e.g., upper-left corner, center, etc., as long as the definition is applied consistently). Two operations (and/or stored variables) are placed "legally" if they do not overlap and there is at least one row of unused electrodes between them. For a more precise definition, refer to [220, 41]; details are omitted to conserve space.

Affinity edges arise from fluidic dependencies in the IR, including those arising between fluidic variables used and defined by the ϕ- and π-functions inserted during SSI construction [41, 178]. Conceptually, for each pair of non-interfering fluidic variables or operations $v_i, v_k \in V$, an affinity edge $v_i, v_k \in A$ exists if a droplet must be transported between the locations where v_i and v_k are placed; thus, it is possible to eliminate a droplet transport operation by placing v_i and v_k at the same location. This occurs in three situations: operation v_i produces a fluid that is used by operation v_k (or vice-versa) as discussed earlier; there exists a ϕ-function $d_{j,k} \leftarrow \phi_j(\ldots, d_{i,j}, \ldots)$; or there exists a π-function $(\ldots, d_{j,k}, \ldots) \leftarrow \pi_j(d_{i,j})$.

Affinity edges can only be inserted between "compatible" operations. For example, a mix operation is compatible with a heat because a mix operation can be scheduled on a DMFB module that includes an integrated heater (presumably turned off). On the



(a) I/O interferences



(b) External module interferences

Figure 4.5: *I/O & Module Interferences:* All I/O reservoirs (a) are universal nodes; their subgraph forms a clique. The subgraph of external modules (b) is a complete multipartite graph, with each module type comprising a part.

other hand, heat and sense operations are incompatible: to date no DMFB devices has integrated a heater and sensor at the same on-chip location.

The interference graph includes a complete multipartite gadget (Fig. 4.5b) to make resource-related incompatibilities explicit. I/O operations bound to the same reservoir cannot interfere, while I/O operations bound to different reservoirs explicitly interfere.

Without loss of generality, a sensing operation cannot be bound to a region of a DMFB that features an integrated heater, and vice-versa.

Figure 4.8a shows the interference graph corresponding to the assay in Fig. 4.2 after scheduling and storage insertion, and assuming that the target DMFB has at least two heaters. Instructions $1, 2, 3, 9$, and 13 are statically bound to I/O reservoirs. Operation 5 (heat c) overlaps with operations $4, 6$, and 7; droplet c is stored after operation 7 (detect d) completes. To conserve space, the interference graph omits the interference edges that belong to the gadget in *resource-interferences* between operation 7 (detect d) and the three heat operations ($4, 5$, and 12). *Fluidic dependencies* result in affinity edges: (v_4, v_6), (v_6, v_7), (v_7, v_{10}), (v_5, v_{11}), (v_7, v_{11}), (v_{10}, v_{11}), and (v_{11}, v_{12}).

4.3.3 Placing a CFG

The next step, which is a novel contribution of this work, is to perform "global" placement in a manner that is cognizant of the CFG, as opposed to prior work [41], which limited the scope of placement to individual basic blocks. As noted in Section 2.1.3, a fluidic dependency (v_i, v_j) represents a droplet $d_{i,j}$ that is produced by v_i and consumed by v_j, which necessitates fluid transport; however, if v_i and v_j are placed at the same location, or at least nearby, the transport operation can be eliminated or shortened. When compiling a CFG, this observation generalizes to dependencies that cross basic block boundaries.

In SSI Form, droplets that are stored across basic block boundaries are represented explicitly by ϕ- and π-functions; prior work has introduced techniques to translate out of fluidic SSI Form [41], but did not attempt to minimize droplet transport latencies while

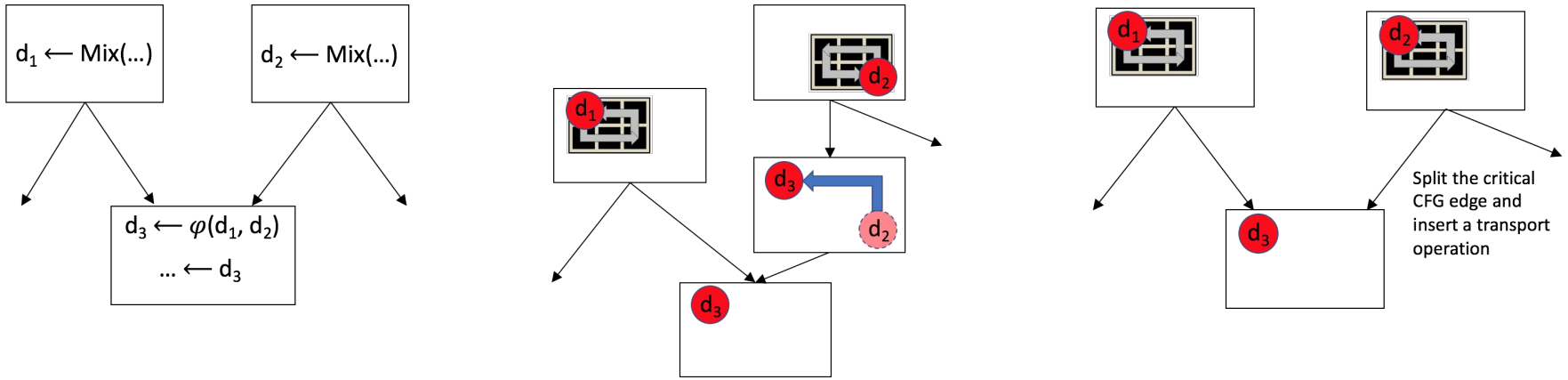


Figure 4.6: An SSA/SSI program fragment (a), shown with a single ϕ-function: $d_3 \leftarrow \phi(d_1, d_2)$. An example of placement that has not been globally optimized from the perspective of the CFG (b): the operation that produces droplet d_1 is placed at the same location as the operation that consumes droplet d_3 (after renaming via the ϕ-function), eliminating the droplet transport operation if the path on the left is taken; however, the operation that produces droplet d_2 on the right is *not* placed at the same location as the operation that consumes d_3, necessitating the introduction of a new basic block (via critical edge splitting [217]) that contains the transport operation. Globally optimized placement (c): the operations that produce droplets d_1 and d_2 are placed at the same location as the operation that consumes droplet d_3, thereby eliminating the need to insert additional droplet transport operations.

doing so; Figure 4.6 illustrates the different results that may occur depending on whether the placer accounts for transport operations that occur due to ϕ- and π-functions.

The global placement problem shares many principle similarities to graph coloring register allocation [33, 32, 70, 28], however there are significant and subtle differences. The first is that global placement must account for both scheduled operations and fluidic variables that are stored on-chip, as both compete for space on the surface of the DMFB. The second is that operations and variables must be placed on a 2D surface, as opposed to being allocated to registers; this requires an extension to the graph coloring model, as simply assigning distinct integer "colors" to two concurrent mixing operations is insufficient to describe where they are placed. The third and final difference is that there is no off-chip fluidic memory, which means that "spilling" is not allowed; if a legal placement cannot be found, then code generation fails, and a larger DMFB is needed.

4.3.3.1 Global Placement as an Optimization Problem

These observations allow us to define global placement as a constrained optimization problem. Let $G = (V, E, A)$ be the fluidic interference graph and $Q = \{q_i | v_i \in V\}$ be the set of locations at which each operation or fluidic variable is placed. A global placement solution is legal if the placement for each interference edge (v_i, v_j) is legal. The objective is to minimize an estimate of the total distance traveled by each droplet, e.g.:

$T = \sum_{(v_i,v_j)\in E} D'(v_i, v_j)$

It is straightforward to generalize this objective, e.g., to add additional weight terms to favor shorter estimated transport distances in deep loop nests.

As an example, Figure 4.7a shows a scheduled basic block. Figure 4.7b shows the resulting interference graph, which features three interference edges and one affinity edge (v_1, v_2). Figure 4.7c shows a legal, but unoptimized placement, the places v_1 and v_2 at different locations, thereby incurring the overhead of droplet transport. Figure 4.7d shows a legal and optimized placement, where v_1 and v_2 are placed at the same location, thereby eliminating the need to transport the droplet.

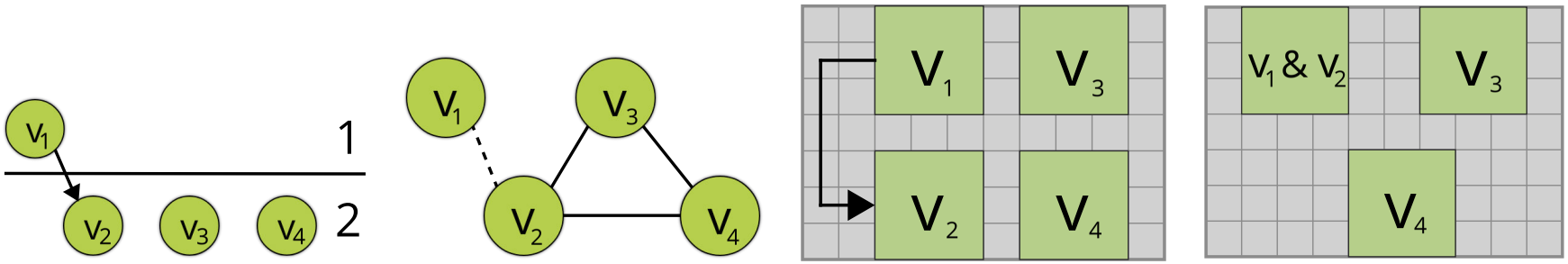
To solve the constrained global optimization placement problem (Section 4.3.3.1) in practice, we used *NSGA-II* [47], a publicly available genetic algorithm[3]. As an iterative improvement heuristic, NSGA-II produces locally optimal solutions, although the running time and overall solution quality depend on a number of user-specified parameters. An initial feasible solution is obtained using an efficient heuristic [78]. The solution is encoded as binary variables of the form (x_i, y_i, v_i, s_i), where v_i and s_i are the orientation and size of operation or stored variable v_i; the number of bits required for x_i and y_i vary, based on the maximum dimension of the architecture.

4.3.3.2 Adapting Graph Coalescing for CFG Placement

Alternatively, we can adapt the principles of graph coloring register allocation directly to the placement problem. Here, we describe how coalescing the interference graph can work toward a unified global placement solution, whereby (possibly coalesced) vertices are placed, rather than discrete operations.

Coalescing merges non-interfering affinity-related vertices in the interference graph to ensure that the corresponding operations are placed at the same on-chip location: this eliminates the need to transport droplets, which can reduce the burden on placement and

[3]Source code available at: `https://www.iitk.ac.in/kangal/codes.shtml`



(a) (b) (c) (d)

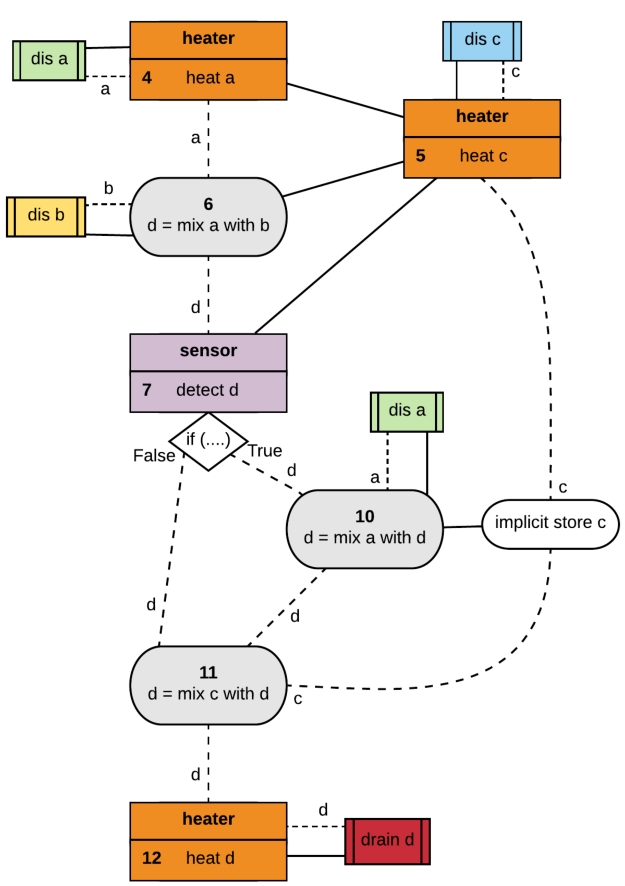
Figure 4.7: A scheduled basic block (a); the resulting interference graph corresponding to the schedule, with one affinity edge (b); an unoptimized placement (c): since there is an affinity edge between v_1 an v_2, placing them at different locations necessitates droplet transport; and an optimized placement (d): since there is an affinity edge between v_1 an v_2, placing them at the same location eliminates the need to transport a droplet.

routing (Sections 4.3.3 and 4.3.4), two NP-complete problems. Coalescing is implemented as an affinity edge contraction operation [28, 70, 123, 109]: given an affinity edge $(v_i, v_j) \in A$ where $(v_i, v_j) \notin E$, vertices v_i and v_j are merged to form new vertex v_{ij} having interference and affinity neighbor sets $adj[v_{ij}] = (adj[v_i] \cup adj[v_j])$ and $aff[v_{ij}] = (aff[v_i] \cup aff[v_j]) \setminus \{v_i, v_j\}$. Figure 4.8a shows the interference graph derived from the scheduled CFG shown in Fig. 4.3b; Figs. 4.8b and 4.8c show two possible coalescing outcomes. In this example, Fig. 4.8c has coalesced more affinity edges than Fig. 4.8b. This, in turn, reduces the workload of the placer (Section 4.3.3) and router (Section 4.3.4) downstream.

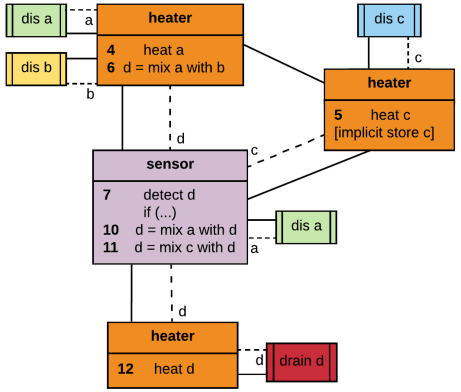
Coalescing here differs from register allocation in one key respect. Consider the example shown in Fig. 4.9a: when coalescing (v_i, v_j) into v_{ij}, traditional mechanisms discard (v_i, v_k), which may result in extended routes (Fig. 4.9b). We instead maintain the affinity, allowing routes to be optimized by placing operations *near* each other (Fig. 4.9c).

When reconfigurable operations of different dimensions are coalesced, the coalesced vertex is given the minimum rectangular dimension that can accommodate its constitutions (see Fig. 4.10a). The type of a coalesced vertex has the most restrictive among $\{reconfig, heat, sense\}$, as shown in Fig. 4.10b.

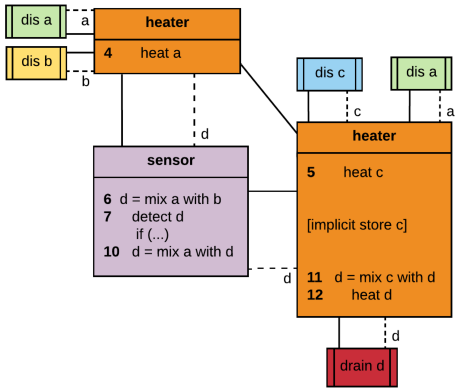
Next, we describe two important subroutines, followed by a description of two coalescing heuristics adapted for our constraints. In the discussion that follows, we talk about interference graph "vertices" rather than assay operations.



(a)



(b) {4,6} & {7,10,11}



(c) {6,7,10} & {5,11,12}

Figure 4.8: **(a)** The interference graph for the Assay in Figure 4.2a: solid lines are interferences, while dotted lines are affinities between nodes. Note that all interferences in Figure 4.5 are present but not depicted; **(b)** and **(c)** show that a coalescing solution is not unique.

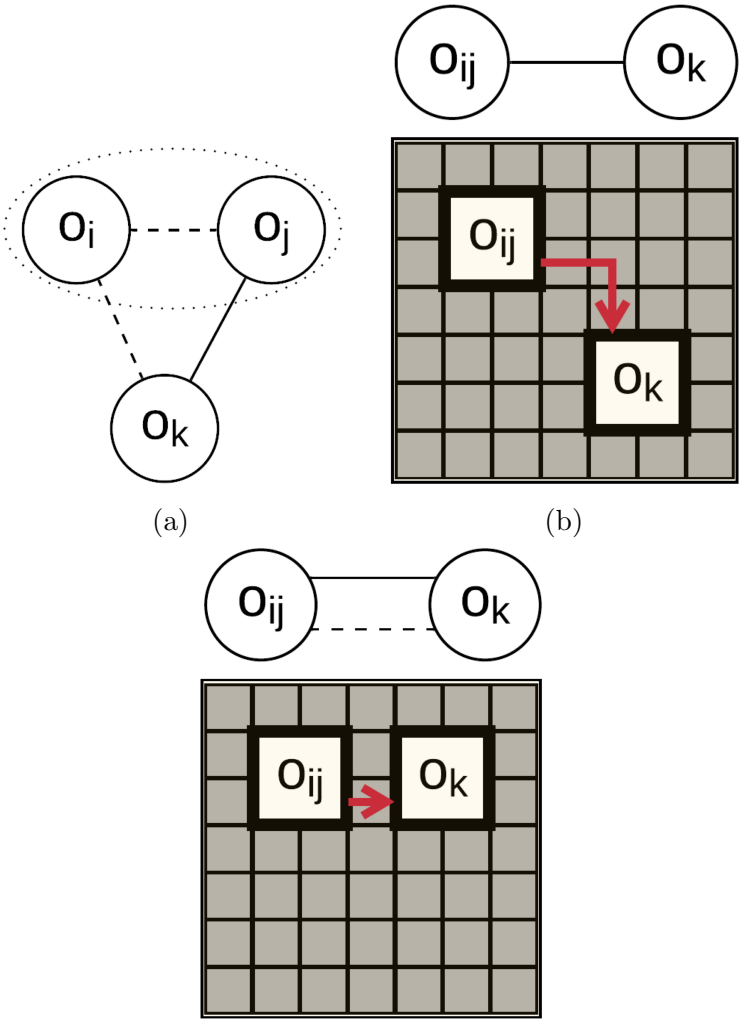


Figure 4.9: **(a)** v_i has affinity with v_j and v_k, while v_j and v_k interfere; traditional coalescing does not maintain the affinity edge when coalescing v_{ij}, which may result in extended routes **(b)**; by keeping the edge, we can optimize routes by placing dependent operations *near* each other **(c)**.

Simplification is a subroutine commonly used during register allocation, which we here adapt for our purposes. Any vertex that trivially satisfies the scheduling resource constraints above, but is *not* affinity-adjacent to any other vertices can be removed from the graph: the rationale is that a legal placement for the simplified vertex can *always* be found regardless of where all of its neighboring vertices are placed. Removing simplified vertices from the graph creates opportunities for new coalescing while also rendering other vertices simplifiable. Following repeated rounds of simplification, all vertices in the remaining graph can be placed. Simplified vertices can then be placed by processing them in reverse order of their removal.

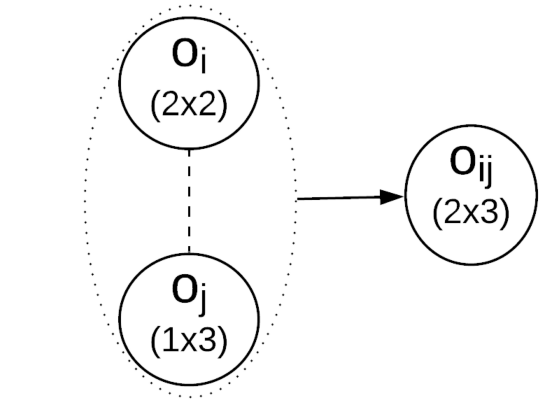
Conservative Coalescing Coalescing is *conservative* if the coalesced vertex v_{ij} and its interference neighbors satisfy the scheduler's resource constraints (Eqs. (4.1) to (4.3)), i.e.:

$$|adj_{heat}{}^*[v_{ij}]| \leq N_{heat} \tag{4.4}$$

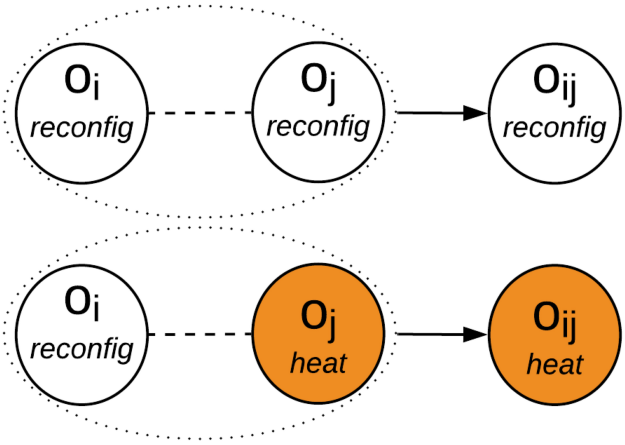
$$|adj_{sense}{}^*[v_{ij}]| \leq N_{sense} \tag{4.5}$$

$$\begin{aligned} |adj_{mix}{}^*[v_{ij}]| + |adj_{split}{}^*[v_{ij}]| + |adj_{merge}{}^*[v_{ij}]| \\ + \left\lceil \frac{|adj_{store}{}^*[v_{ij}]|}{k} \right\rceil + |adj_{heat}{}^*[v_{ij}]| \\ + |adj_{sense}{}^*[v_{ij}]| \leq N \end{aligned} \tag{4.6}$$

Coalescing Strategy *Iterated Coalescing*, depicted in Fig. 4.11, is adapted from iterated register coalescing [70], but without spilling. The iterated coalescer simplifies the interference graph until it is not possible to do so any further. It then applies conservative coalescing; if coalescing occurs, further simplification is performed; otherwise, a low-degree vertex with at least one incident affinity edge is "frozen" i.e., the coalescer gives up



(a)



(b)

Figure 4.10: The rectangular dimensions of a coalesced vertex are the minimum dimensions that can accommodate its constituent parts **(a)**; a coalesced vertex takes on the type of its most restrictive module **(b)**.

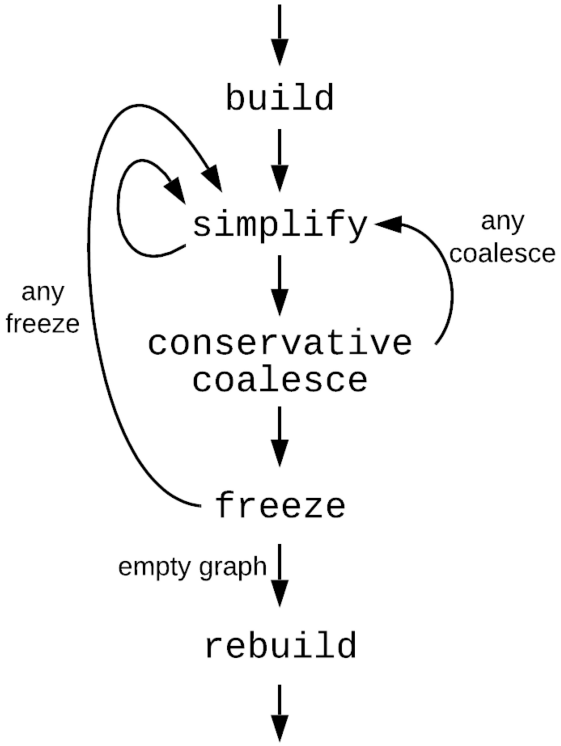


Figure 4.11: **Phase ordering of Iterated Coalescing** [70]

hope of coalescing its incident affinity edges, thereby allowing the vertex to be simplified. Iterated coalescing terminates when all vertices have been removed via simplification. The graph is then rebuilt and passed to the placer. Conservatism is guaranteed by the observation that the initial interference graph, simplification process, and conservative coalescing strategy ensure that the scheduler's resource constraints are satisfied at each step of the heuristic.

4.3.3.3 Optimized CFG Placement

While the optimization approach in Section 4.3.3.1 worked at the granularity of individual basic blocks, and iteratively adjusted each basic block to try to find a converging (locally) optimal global placement solution, the updated model shown in Fig. 4.12 is able to utilize existing placement methods with adjustments for placing (possibly coalesced) graph vertices, rather than discrete operations.

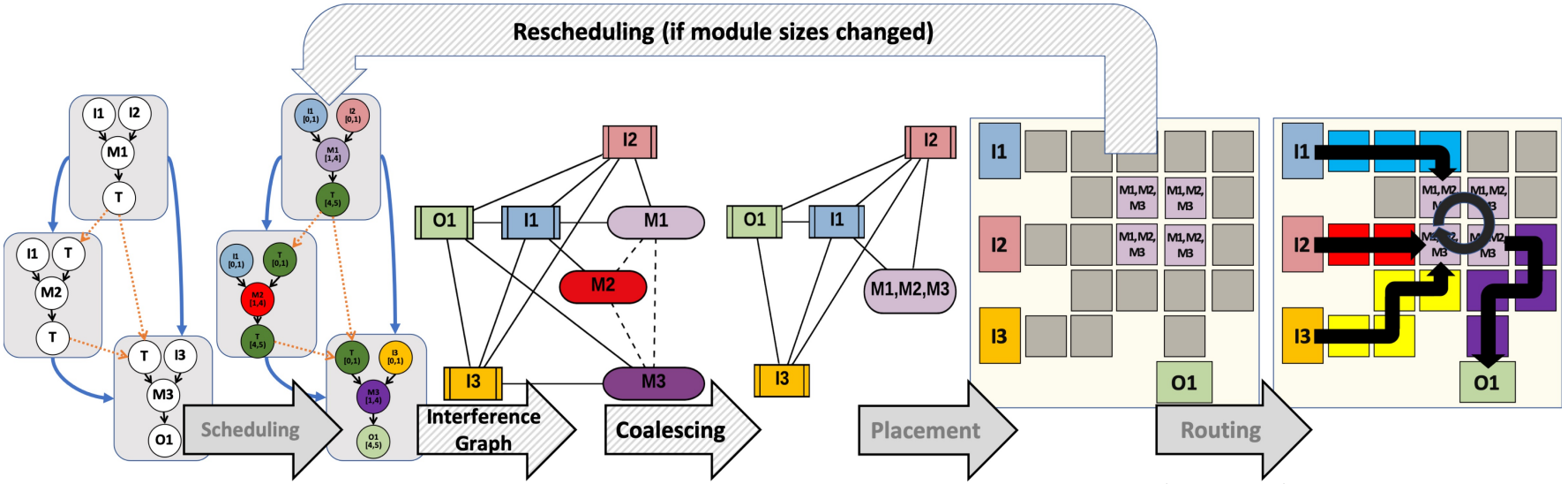


Figure 4.12: Overview of our DMFB compiler. The front-end compiles an assay specification to a CFG (not shown). The back-end converts the CFG to an executable format. The "Interference Graph", "Coalescing", and "Rescheduling" arrows are novel additions to traditional 3-step synthesis (e.g., [79])

As before, the placer determines the location on-chip where each assay operation will execute. A legal placement satisfies the constraint that operations v_i and v_j are placed at non-overlapping positions for each interference edge $(v_i, v_j) \in E$. In addition to supporting the new NSGA-II placement algorithm, the updated version of our compiler updates two distinct placement strategies that have been published elsewhere: *Virtual Topology with Left-Edge Binder (VT-LEB)* [78] and *Keep All Maximal Empty Rectangles (KAMER)* [18]. The *Virtual Topology* aspect of VT-LEB refers to the partitioning of the DMFB that the scheduler performs as described in Section 4.3.1 (see Fig. 4.4). Prior work implemented these heuristics in a manner similar to linear scan register allocation [185]: starting with a scheduled basic block, the placer scans each program point in sequential order: operations scheduled to complete at the previous time-step are removed from the current placement, and operations scheduled to begin at the subsequent time-step are added to the placement.

Our compiler uses modified versions of NSGA-II, VT-LEB and KAMER to perform placement on a coalesced interference graph rather than a scheduled CFG; vertices are processed one-by-one in a worklist sorted by the earliest time step.

When coalescing is performed, affinity relationships between interfering vertices may still exist, indicating exactly which vertices should be placed near each other; hence, after placing v_i, if $\mathit{aff}[v_i] \neq \emptyset$, we recursively process affinity neighbors prior to returning to the sorted order (see Fig. 4.9c).

Let $adj^{<}[v_i]$ be the set of v_i's interference neighbors that precede v_i in the computed order. Placement proceeds in a greedy fashion: operation v_i can be placed at any position that does not overlap the position(s) where operations in $adj^{<}[v_i]$ have been placed. All

vertices that have been coalesced with/into v_i are placed at the same location. The resulting placement is guaranteed to be legal as it ensures that v_i's position never overlaps that of any vertices in $adj[v_i]$. VT-LEB guarantees that a legal placement can be found because it ensures that all placement decisions adhere to scheduling resource constraints.

4.3.3.4 Mix Operation Resizing and Rescheduling

The rescheduling loop in Fig. 4.12 enables the compiler to adjust the size of mixing operations (Table 2.2) during placement to reduce assay execution time, enable valid placement results, or increase available operation-level parallelism. The availability of space to accommodate larger mixing operations is not known until placement; on the other hand, the benefits of adjusting the latency of a mixing operation cannot be ascertained without rescheduling, and the updated schedule may change which fluidic variable live ranges overlap, thereby rendering the interference graph invalid. This observation necessitates the rescheduling loop.

The compiler uses a *local search*, which converges to a locally optimal solution, to adjust mixing operation sizes. When placing an interference graph, the first mixing operation or coalesced vertex o_i that contains at least one mix operation invokes Algorithm 1 to select an appropriate mixer size. The heuristic relies on two subroutines:

1. `MaxParallel` applies Dilworth's Theorem [49] to compute the *width*, i.e., the maximum number of operations that *could* be scheduled concurrently, of the basic block that contains o_i; if o_i contains multiple coalesced vertices, `MaxParallel` returns the maximum width of among all the basic blocks containing them.

2. `CanFit` computes the number of mixing modules of size s that can fit on a given DMFB architecture. Referring back to Fig. 4.4, `CanFit` is effectively the same subroutine that a scheduler would use to determine the resource constraints of the target chip.

The heuristic first checks if o_i's scheduled module size `CanFit` `MaxParallel` operations. If more parallelism is available than what is currently scheduled, it checks if *smaller* modules `CanFit` more than those currently scheduled, and continues until it finds a module size that `CanFit` up to `MaxParallel` operations.

The heuristic will *increase* o_i's size in two cases:

1. If the chip `CanFit` strictly fewer than `MaxParallel` operations, and the heuristic is unable to increase the number of operations the chip `CanFit` by decreasing o_i's module size, then it *increases* o_i's size as long as it does not further reduce the number of operations that the chip `CanFit`.

2. If o_i's scheduled module size `CanFit` `MaxParallel` operations, then the heuristic *increases* o_i's size to the largest point where `MaxParallel` operations `CanFit` on the chip.

When the size of a mixing operation is updated, the size of any other mixing operations that are coalesced with it are updated as well. If a mixing operation is updated during placement, its latency is scaled as per Eq. (4.7) and the compiler loops back to scheduling:

$$t' = t * latency_{new}/latency_{old} \tag{4.7}$$

For example, if the compiler changes a 10 second `mix` operation's given work module from a 2×3 to a 2×4 module, then the compiler computes the new latency as $t' = 10 * 2.9/6.1 \approx 4.76$ seconds. The compiler rounds the new latency up to the next millisecond. The termination criteria to continue on to droplet routing is either (1) module sizes are not updated during placement, so rescheduling is unnecessary, or (2) the loop taken during a rescheduling loop failed during scheduling or placement. In the case of (2), we revert to the best legal schedule and placement found. The interested reader can find an example of module resizing in Appendix A.

Algorithm 1 Resizing Heuristic

```
1:  function ChooseModuleSize(Block(s) b, Vertex o_i)
2:      current ← o_i.size
3:      choice ← current
4:      max ← MaxParallel(b)
5:      currNum ← CanFit(current)
6:      updated ← False
7:      if max > currNum then
8:          chosenNum ← currNum
9:          smaller ← current
10:         while smaller ≠ smallest do
11:             smaller ← decrease(current)
12:             check ← CanFit(smaller)
13:             if check > chosenNum then
14:                 choice ← smaller
15:                 updated ← True
16:                 chosenNum ← check
17:                 if chosenNum ≥ max then break
18:     if updated = False then
19:         larger ← decrease(current)
20:         check ← CanFit(larger)
21:         while check = currNum OR check ≥ max do
22:             choice ← larger
23:             if larger = largest then break
24:             larger ← increase(choice)
25:             check ← CanFit(larger)
26:     o_i.size ← choice
```

4.3.4 Droplet Routing

Once a legal placement solution is obtained, each droplet must be routed from its source to its destination; many papers published in the past 15 years have described routing algorithms, and in principle any can be used [222, 24, 38, 246, 95, 194, 195, 112, 111]. The most advanced routers also integrate washing operations to eliminate cross-contamination [97, 250, 243]. The only additional requirement is that droplet routes must be inserted at basic block boundaries; our compiler implements these routes as part of SSI elimination [41].

4.4 Implementation

4.4.1 Overview

Our compiler targets an open-source cycle-accurate DMFB simulator [77, 79]; we modified a back-end that can statically compile CFGs [41], and rely on the simulator to report execution time. The simulator is primarily used for performance characterization under idealized (i.e., fault-free) operating conditions.

As the simulator does not have access to physical sensors, it generates pseudo-random numbers, constrained within realistic values, to represent sensor readings that are then passed to the execution engine when confronted with a *detect* instruction. We used a collection of benchmarks specified using the *BioScript* language, which is compatible with the framework's static compilation model [178].

Scheduling is performed block-by-block using a modified List Scheduling algorithm [221, 78]. We construct the interference graph as detailed in Section 4.3.2 prior to

performing placement, using our novel NSGA-II algorithm or modified versions of VT-LEB [78] or KAMER [18]. Finally, all methods use a greedy, yet effective, droplet router [194, 78].

The NSGA-II heuristic, in its basic form, does not utilize coalescing or module resizing; the basic 3-step synthesis method (Fig. 2.6) is instead followed. We configured NSGA-II to use a population size of 100 and to run for 250 iterations. The initial feasible solution is implanted in the initial population, and additional solutions are generated via mutation and crossover operations. After each generation, encoded solutions that NSGA-II discovers via evolution are extracted and examined for legality; the objective is computed for legal solutions. The constraints and objectives are returned to NSGA-II for the subsequent evolution. We maintain a copy of the best solution(s) found thus far. After the final generation, we return the best legal placement solution that was discovered.

4.4.2 Modification of Placement Algorithms for Placing Interference Graphs

The updates to the synthesis model for utilizing a (possibly coalesced) interference graph for placement is evaluated with all three aforementioned placers, but as coalescing is abstracted away from placement, any existing placement heuristic could be easily modified to operate on a coalesced interference graph. Our coalescing model employs the three placement strategies described in Section 4.3.3. The modifications required on these algorithms to place a coalesced interference graph are summarized here:

Virtual Topology with Left-Edge Binder (VT-LEB) [78]: We extended VT-LEB to accept an interference graph as input and imposed the constraint that two operations whose corresponding vertices are adjacent cannot be bound to the same work module, which allowed our implementation of VT-LEB to be compatible with our proposed global coalescing strategy: our updated VT-LEB binds vertices, rather than operations, to available work modules; when a binding decision is made, all the operations that have been coalesced into the interference graph vertex are bound to the same work module. We added or modified 23 lines to the VT-LEB code to allow for VT-LEB to bind a coalesced interference graph.

Keep All Maximal Empty Rectangles (KAMER) [18]: KAMER performs free placement of operations for each time step in a given block one-by-one in an iterative manner as follows:

1. the DMFB array is segmented into a set of maximum empty rectangles (MERs) – the largest rectangles that do not have any placed operations

2. the operation o_i being considered for placement then checks each MER to see if it contains the appropriate resources required and has ample room to fit o_i's dimensions.

3. the MER with the smallest area that fits o_i is chosen for placement, and a module is assigned to the location with o_i's scheduled time assigned to it

It is straightforward to modify KAMER to place an interference graph: when a vertex is placed, all the operations that have been coalesced into that vertex are placed at the same location. The size of the placed location of an individual operation or a coalesced

operation is equal to the size of the largest operation that has been coalesced into the vertex. KAMER's underlying algorithmic details (how to select free space, and how to update the data structure that represents free space) remain the same. We added or modified 31 lines to the KAMER code to allow KAMER to place a coalesced interference graph.

NSGA-II: Coalescing does not impact the NSGA-II problem formulation in the slightest: when a vertex is placed, the space and location allocated is equal to that of the largest operation that has been coalesced into the vertex. As described in Section 4.3.3.1, NSGA-II can optimize the placement of interfering vertices that are also affinity-adjacent. Since the vertices interfere, coalescing is not possible and the corresponding operations cannot be placed at the same location; however, droplet routing paths can still be shortened by placing them *near* one another. We added or modified 18 lines to our NSGA-II interface to enable these changes.

4.4.3 Modification of Placement Algorithms for Mix Module Resizing

While rescheduling is abstracted away from placement, the resizing operations, by necessity, must be performed during placement, which necessitates a more substantial revamp of existing methods. Our current implementation is only compatible with placers that place operations one-at-a-time in a greedy fashion.

As the work module size in a virtual topology is fixed (Fig. 4.4), and the VT-LEB relies on the virtual topology for binding, it is not possible to resize mixing operations with the LEB without larger adjustments to the underlying resource scheduler, which is outside the scope of this work. Additionally, as the NSGA-II placer's current implementation relies on an initial seed into the population of a placement given from LEB, we did not

implement resizing for the NSGA-II algorithm. Ostensibly, resizing with NSGA-II should be possible; however, the heuristic used would be different. Specifically, as discussed in [101, 46], optimizing for more than 2 objectives with NSGA-II not only results in significantly longer computation, but also in poorer results. To encode NSGA-II with the dual optimization criteria of parallelizing operations and reducing individual module latencies on top of the current goal of reducing distances between affinity-adjacent vertices would then likely not give much-improved results. We therefore leave for future research the implementation of resizing optimizations using a different multi-objective evolutionary algorithm that is well-suited for three or more optimization objectives.

As KAMER binds regions of the chip freely, we allow KAMER to dynamically adjust the sizes of a module prior to it being bound using the method described in Section 4.3.3.4. The application of Dilworth's Theorem to find the maximum width of an assay is accomplished through building a bipartite graph B from the block (where each part of $B = V$, with edges from (u', v'') existing wherever $u < v$ in the block), finding a maximum matching M on B, and using M to partition the block into a minimal set of w chains, the maximum number of parallel operations.

When an operation's size as given from the scheduler is adjusted when placing a module, we reschedule the assay using the updated module size. As the number of module sizes we choose from is limited (Table 2.2), and the heuristic we use converges as a local search (Algorithm 1), the number of times we might reschedule an assay is minimal. Module resizing required more extensive modifications to the algorithm; around an additional 450 lines of code.

4.5 Evaluation

Table 4.1: Compile time and simulated execution times

Benchmark	**Compilation Time (sec)**	**Execution Engine Time (m:s:ms)**	**Execution Time**
Broad Spectrum Opiate [12, 146, 108]	0.011	0:18:55	0:23:21
Ciprofloxacin [108]	0.023	101:31:80	128:54:32
Diazepam [91]	0.024	96:48:13	121:01:39
Dilution [91]	0.014	0:21:05	0:26:33
Fentanyl [146]	0.018	126:32:40	158:10:80
Full Morphine [91]	0.048	127:16:78	159:06:17
Glucose Detection [6]	0.012	0:23:77	0:29:73
Heroine [91]	0.020	126:32:40	158:10:80
Image Probe Synthesis [6]	0.015	8:38:96	10:47:50
Morphine [91]	0.018	126:32:40	158:10:80
Oxycodone [12]	0.026	126:32:40	158:10:80
PCR [6]	0.032	11:16:12	14:36:29
Cancer-detection [210]	0.016	1920:08:01	N/A

Even though we support physical chips, the expense associated with their use is prohibitive for evaluation purposes; hence, we evaluate our compiler through simulation-based empirical studies on known real-world assays specified for execution on DMFBs. Specifically, we aim to evaluate the impact of global placement on compilation and assay execution time, along with the ability to successfully compile onto very-small architectures. We compare the traditional 3-step synthesis method against our NSGA-II approach, as well as the updated model with coalescing and mix operation resizing. All reported averages use the geometric mean over the ratios of each benchmark to avoid providing too much weight to longer- or shorter-running benchmarks [64].

4.5.1 Experimental Setup

Experiments were performed on a 2.7 GHz Intel®; Core™ i7 processor, 8GB RAM, machine running macOS®. We compare directly against a previously published compiler [41] using an identical 15×19 DMFB architecture. We also report results on 15×15, 12×12 and 8×8 DMFBs to evaluate the impact of our mix operation resizing heuristic. Reducing the amount of on-chip area and the number of heating and sensing models allows exploration of trade offs between parallelism and operation latency.

4.5.2 Benchmarks

Our evaluation uses a set of DMFB benchmarks that were previously used to evaluate the compiler that we use as a baseline for comparison. Ref. [41] specified them using a variant of the BioCoder language, which is now deprecated; Ref. [178] translated these into the *BioScript* language; listings of the benchmarks are given in Appendix B.1. Each of these benchmarks were obtained by reading the scientific literature on DMFBs and extracting specifications of assays that were used in practice by other research groups.

4.5.3 Compiler Configurations

The baseline DMFB compiler we compare against ([41]) does not employ coalescing or mix operation resizing; it compiles a CFG one basic block at a time using the standard VT-LEB algorithm for placement ([78]), eschewing optimizations across basic block boundaries. The NSGA-II placer does not employ coalescing, but does attempt to maximize the number of affinity-adjacent operations that are placed at the same location; it also aims to place affinity-adjacent operations that cannot be placed at the same location nearby one another to reduce the length of droplet routing paths. The Virtual

Topology with Left-Edge Binder (VT-LEB) and KAMER placers are evaluated for their ability to place a coalesced interference graph, and KAMER is additionally compared while module resizing is enabled.

4.5.4 Results and Analysis

Table 4.2 compares simulated assay execution times previously reported for the baseline compiler [41] to the five configurations of the compiler presented here: NSGA-II (N), NSGA-II plus coalescing (NC), VT-LEB placement plus coalescing (VC), KAMER placement plus coalescing (KC), and KAMER placement plus both coalescing and mix operation resizing (KCR). On average, VC, KC, and KCR reduce assay execution time by 1.1%, 1.2%, and 25.0% respectively. These results are not surprising, as assay execution time is known to be dominated by schedule latency, not droplet routing time [222]; as optimizations, coalescing aims to reduce droplet routing overhead while mix operation resizing can lead to shorter schedules. We observed that convergence typically occurs after 2 iterations of rescheduling when resizing is enabled.

The improvements reported for VC and KC over Ref. [178] indicate situations where coalescing turns out to be more effective than the NSGA-II placer; however, NSGA-II may discover different (and possibly better) solutions if the random number seed and other configuration parameters are varied. Future work may extend the NSGA-II placer to utilize a coalesced interference graph; the amount of work required to extend the NSGA-II placer with resizing capabilities (which would entail re-scheduling and re-placing at every perturbation) is prohibitive, so we did not evaluate this option.

The compiler described in Ref. [41] utilizes the same placer as VC, sans coalescing. Adding coalescing capabilities yielded marginal improvements, due to the fact that droplet routing does not dominate total assay execution time.

Mix operation resizing had a more profound impact on total assay execution time than coalescing. Furthermore, Fig. 4.13 depicts an observed linear correlation between the amount of time an assay is specified for mixing and the percentage decrease we expect to achieve via resizing across DMFBs of varying size. At the smallest size, 8×8 (Figure 4.13d), resizing allows us to compile several assays that failed to compile successfully without this optimization turned on. Through inspection, we determined that our resizing heuristic was able to avail the minimum required parallelism for these assays by using a 1×4 module size; the default 2×2 mixer did not provide enough room for a legal schedule.

Table 4.3 provides details into how coalescing impacts the placer's workload and droplet routing time. On average, coalescing reduces the number of operations that are placed by 77%; this, in turn, reduces the amount of work that needs to be done during both placement and routing. In terms of overall performance impact, the VC and KC placers reduced droplet routing times by 9.4% and 8.6% compared to the baseline.

4.6 Conclusion

This chapter described the framework of an optimizing compiler that extends the scope of programmable LoC compilation optimizations to the granularity of the CFG. The key innovations were twofold: the formulation of the placement problem for CFGs that shares

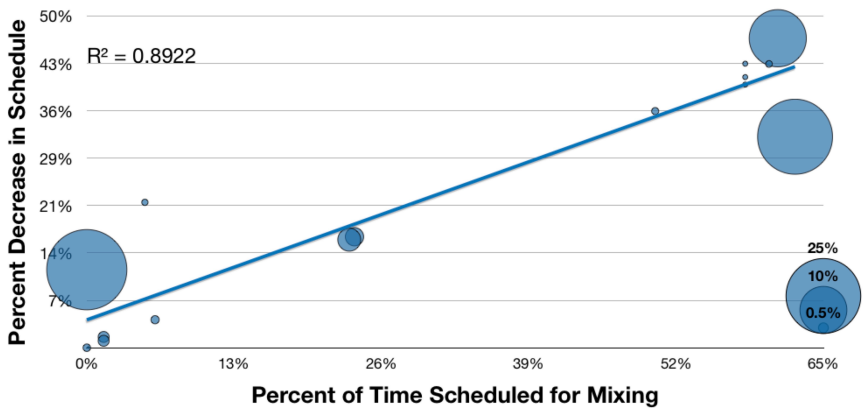
Table 4.2: Impact of coalescing, choice of placement heuristic, and mix operation resizing on total assay execution time.

Assay	**Total Execution Time (m:s.ms)**				
	Baseline	**NSGA-II + Coalesce**	**VT-LEB + Coalesce**	**KAMER + Coalesce**	**KAMER + Coalesce + Resize**
BroadSpectrumOpiate	00:18.550	00:17.900	00:18.200	00:17.810	00:16.180
CancerDetection	1920:08.010	1920:06.020	1920:06.000	1920:05.810	1919:24.000
Ciprofloxacin	101:31.800	100:37.170	100:37.100	100:36.910	100:29.950
Diazepam	96:48.130	96:50.000	96:50.180	96:49.760	96:14.970
Dilution	21:05.000	20:40.980	20:43.000	20:41.000	06:25.470
Fentanyl et al.	126:32.400	126:36.000	126:24.540	126:24.330	72:20.600
FullMorphine	157:21.540	157:20.400	157:21.500	157:19.890	122:52.780
GlucoseDetection	00:23.770	00:23.220	00:23.590	00:23.730	00:16.730
ImageProbeSynth	08:38.960	08:22.860	08:22.860	08:22.780	06:58.860
OpiateDetection_N	252:50.400	252:42.000	252:50.100	252:47.500	144:36.540
OpiateDetection_P_H	227:04.000	226:55.000	227:03.700	227:01.800	137:01.140
OpiateDetection_P_M	353:20.700	353:19.980	353:20.200	353:17.100	209:13.700
PCRDropletReplacement	40:44.000	39:16.960	39:17.890	39:17.120	32:55.170
ProbabilisticPCR_early	07:21.000	07:12.400	07:12.420	07:12.390	07:05.430
ProbabilisticPCR_full	11:19.000	11:10.550	11:10.600	11:10.550	11:03.610
PCR	11:43.000	11:27.350	11:27.370	11:27.380	11:27.370
Average Decrease:		*1.1%*	*1.1%*	*1.2%*	*25.0%*

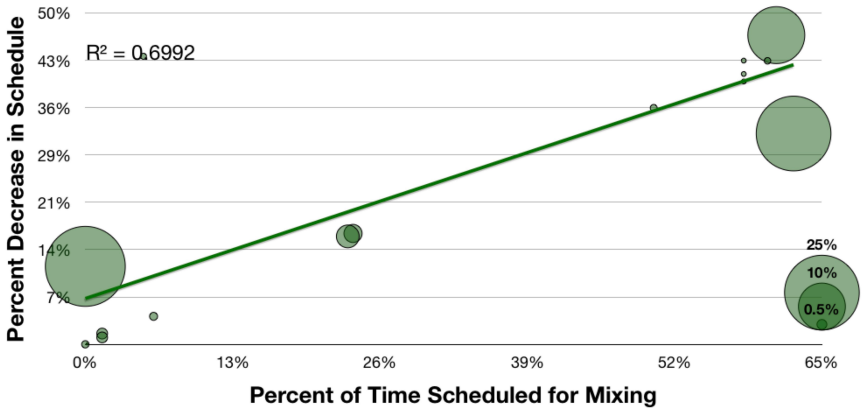
N - negative P - positive H - heroin M - morphine

Table 4.3: Impact of coalescing on placement effort and droplet routing time.

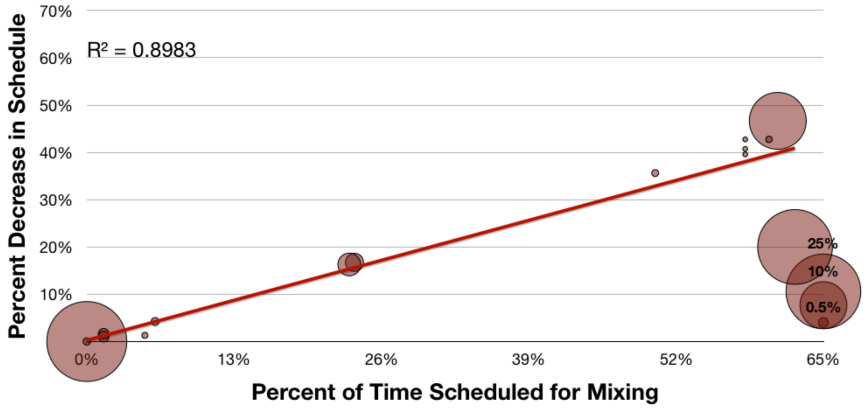
Assay	# Modules Placed		Droplet Route Time (s.ms)			
	Baseline	**Coalesced**	**Baseline**	**NSGA-II + Coalesce**	**VT-LEB + Coalesce**	**KAMER + Coalesce**
BroadSpectrumOpiate	5	2	00.740	00.880	00.740	00.910
CancerDetection	11	4	00.820	00.710	00.680	01.000
Ciprofloxacin	11	3	02.390	02.060	02.450	02.220
Diazepam	13	2	02.600	02.920	02.670	03.950
Dilution	11	2	00.710	01.150	00.750	01.150
Fentanyl et al.	11	3	02.670	02.140	02.770	02.540
FullMorphine	19	8	05.840	08.050	05.760	08.420
GlucoseDetection	10	5	01.470	01.630	01.250	01.800
ImageProbeSynth	9	1	00.770	00.650	00.530	00.780
OpiateDetection_N	49	4	06.060	05.620	05.030	07.230
OpiateDetection_P_H	49	4	05.820	05.400	04.770	06.870
OpiateDetection_P_M	49	4	08.470	07.680	06.890	10.030
PCRDropletReplacement	4	2	00.510	00.230	00.510	00.350
ProbabilisticPCR_early	10	2	07.610	03.530	03.920	05.420
ProbabilisticPCR_full	8	2	00.630	00.340	00.530	00.390
PCR	8	2	00.870	00.400	00.770	00.550
Average Decrease:		*77.1%*		*10.3%*	*9.4%*	*8.6%*



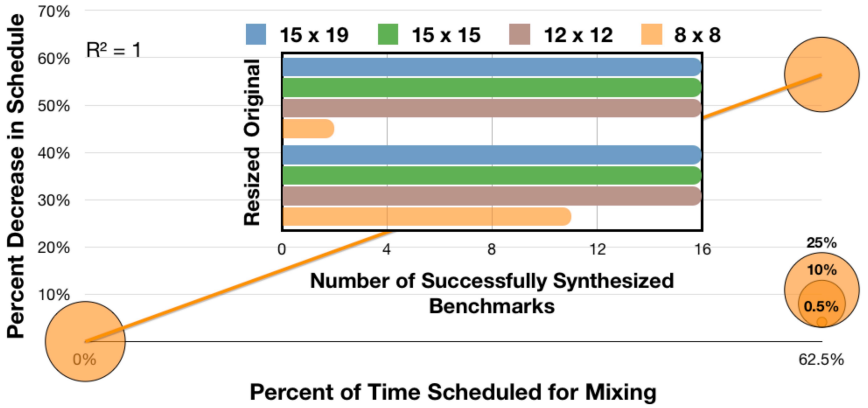
(a) 15×19 chip



(b) 15×15 chip



(c) 12×12 chip



(d) 8×8 chip is hamstrung without resizing

Figure 4.13: Resizing mix operations: we observe a linear correlation between the ratio of time spent mixing and the expected percent decrease in an assay's total schedule. The size of the bubbles indicate the ratio of time scheduled for I/O operations. Without resizing, the 8×8 chip can only synthesize 2 of the 18 benchmarks. With resizing enabled, we are able to successfully synthesize 11 out of the 18.

many principle similarities to register allocation [33, 32], and a mix operation resizing and rescheduling loop. We presented a novel heuristic to solve the placement problem in isolation, as well as a generalized update to the placement problem enabling the adaption of a register coalescing technique [28, 70]; the coalescing technique provides an easy path for existing placement methods to place CFG-level interference graphs, rather than individual operations within a basic block, which reduces the burden of placement and routing during synthesis and eliminates or reduces otherwise-spurious droplet routes. The resizing and rescheduling operations effectively reduce scheduled latency while increasing instruction-level parallelism or enabling synthesis onto architectures that are otherwise useless due to their restricted size. While there is certainly room to investigate more effective heuristics that solve the various problems within the compiler, we believe that the general back-end framework presented here represents the correct way to model the constituent optimization problems that must be solved, along with their interactions. Moreover, we believe that the most important topics for future investigation start at the programming language level; for example, determining how to support function calls, fluidic arrays, and fluidic SIMD operations; additionally, there is need to port BioScript (and/or other similar languages) to a variety of pLoC targets in addition to DMFBs.

Chapter 5

Compiling Functions onto pLoCs

5.1 Introduction

This chapter describes the implementation for compiling function calls onto pLoCs, which can be realized on a diverse array of devices, ranging from those that are microfluidic in nature [1, 76, 86, 167, 174, 187] to pipetting robots [143, 23, 149] and cloud laboratories [106, 35, 157].

While virtually all programming languages provide the programmer with syntactic constructs to specify functions/procedure calls, prior work on programming language and compiler design for programmable chemistry has glossed over the issue, either providing support exclusively for functions that can be inlined [178] or managing everything dynamically with a runtime interpreter [239]. This chapter provides a thorough treatment of how to implement functions statically in the context of one specific microfluidic technology that shares many principles with spatial computing architectures; this treatment is general enough to support functions that cannot be inlined, such as recursive functions

and functions that are provided as pre-compiled binaries without source code or an intermediate representation.

Supporting functions in this context does not create novel algorithmic challenges that need to be solved, but involves a number of observations about how a system that manipulates fluids, rather than bits, must behave. To the best of our knowledge, these observations have not been disseminated previously. Just as an example, a chemical program that calls a function may have defined fluidic variables that are not used by the function but are live across the call; this necessitates the creation of a stack for fluid storage which must be maintained on a microfluidic architecture that does not have a clearly delineated memory subsystem or the possibility of off-chip storage. These observations inform both the implementation of functions at the compiler level and the extent to which a programmer can expect to write fluidic programs that make extensive use of nested function calls.

This chapter makes the following contributions:

- It distinguishes the unique complexities involved in compiling and executing functions on reconfigurable spatial architectures with extreme memory restrictions and possibly requiring device support for externally-attached modules (i.e., heaters or sensors).

- It introduces a cyber-physical paradigm to model both data and fluidic values we call a *split-technology stack*, enabling bookkeeping and control of live fluidic values outside the scope of a function call during its execution.

- It provides solutions for static compilation of chemical languages containing function calls from source code or library (pre-compiled) function specifications.

- It implements our solutions in an open-source compiler for DMFBs, and provide back-end support for `OpenDrop`, an open-hardware DMFB device.

The remainder of this chapter is organized as follows: Section 5.2 briefly presents the various technological issues we must overcome to support functions these devices. Section 5.3 presents an overview of how to address various difficulties when compiling functions onto DMFBs. Finally, we present proof-of-concept evaluation in Section 5.4 using both cycle-accurate DMFB simulation and by direct execution using the open-hardware `OpenDrop` device before concluding the chapter in Section 5.5 with directions for future work.

5.2 Technology Issues

A DMFB has considerable more in common with spatial computing architectures (e.g., FPGAs) than CPUs; even so, important technology differences persist, and they inform relevant aspects of this work. Semiconductor-based computing systems naturally separate the physical resources used for logic (transistors), data transport (wires), and on-chip storage (flip-flops, register files, caches); in contrast, a DMFB employs (groups) of electrodes for all operations, noting that operations such as sensing or heating nonetheless require a droplet be held in-place on a specific electrode. Second, most computing systems employ some form of external storage such as EEPROM, DRAM, hard disk, or flash memory, with greater density (e.g., bits stored per unit area/volume) than on-chip storage; in contrast, DMFBs presently do not have external storage[1], and, even if they did, there would be no density advantage to storing $1\mu L$ of fluid on- or off-chip.

[1]This may change as liquid-handling robots [143, 23, 149] evolve; to date, external storage with fluidic read/write functionality has not been demonstrated.

5.3 Fluidic Functions

Functions are ubiquitous in modern programming languages. As languages for DMFBs evolve, and programs grow in size and complexity, it makes sense to include functions as a first-class syntactic construct. Our focus here is how a compiler targeting a DMFB can implement functions; syntactic considerations and performance overhead are secondary concerns.

Why Not Inline? `BioScript`'s syntax allows programmers to write non-recursive functions, all of which were inlined by the compiler [178]. While aggressive inlining can negatively impact instruction cache performance for CPUs [151], such concerns are irrelevant to DMFBs. One justification against inlining is to support *recursion*. While we are unaware of any fluidic algorithms that are naturally expressed recursively, we prefer not to make decisions that restrict expressiveness of programs, given the ubiquity of recursive programming in computing. A second justification is to support *pre-compiled binaries* that can be linked (statically or dynamically) without providing direct access to the source code or intermediate representation. While it is unclear if there is a practical use case for pre-compiled binaries in microfluidic programming today, we prefer to eschew engineering decisions that would preclude this possibility in the future.

5.3.1 Function Definition

A function f is a mapping $f : \mathcal{D}^p \rightarrow \mathcal{D}^q$, where $p, q \in \mathbb{Z}^{\geq 0}$ and $\mathcal{D}^p$ and $\mathcal{D}^q$ are ordered sets containing p and q droplets respectively[2]. Let $\mathcal{D}_{in} = \langle \mathcal{I}_1, \mathcal{I}_2, \ldots \mathcal{I}_p \rangle \in \mathcal{D}^p$ be the set of input droplets (parameters) and $\mathcal{D}_{out} = \langle \mathcal{O}_1, \mathcal{O}_2, \ldots \mathcal{O}_q \rangle \in \mathcal{D}^q$ be the set of output droplets produced by f. To simplify our initial presentation of key concepts, we will assume that function f only applies the five operations shown in Figure 2.4; we will add support for specialized operations, such as heating, and I/O operations that access reservoirs on the DMFB's perimeters, later in Sections 5.3.6 and 5.3.7.

5.3.2 Function Placement

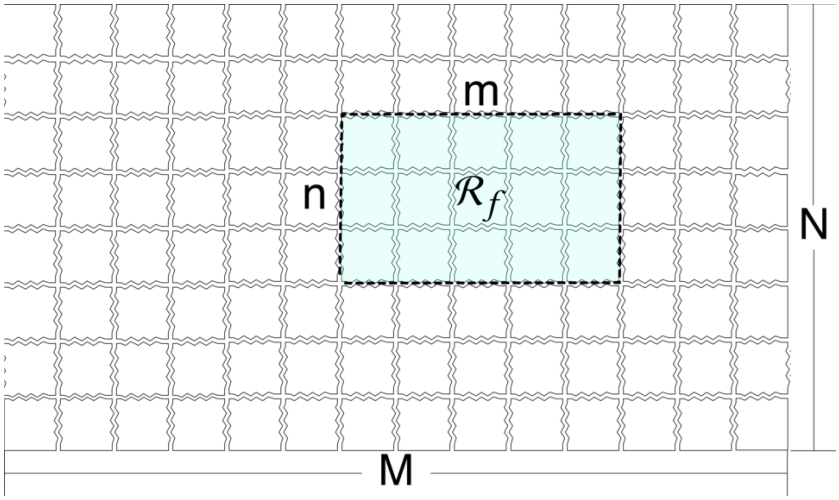


Figure 5.1: A compiled function f requires an $m \times n$ region $\mathcal{R}_f$. When compiling a caller of f, the placer reserves an unoccupied region on the $M \times N$ DMFB for $\mathcal{R}_f$ to use.

A DMFB is defined to be an $M \times N$ electrode grid with I/O reservoirs on the perimeter. A function f can be compiled onto an $m \times n$ rectangular sub-region, denoted $\mathcal{R}_f$, such that $m \leq M$ and $n \leq N$; existing DMFB compilation algorithms mentioned in Section 5.2 can be adapted for this task. When compiling the caller, the placer allocates an unoccupied $m \times n$ sub-region of the DMFB and places $\mathcal{R}_f$ there (Fig. 5.1).

[2]Groups of droplets (e.g., arrays, structs, unions, etc.) can be handled analogously, but are omitted for brevity.

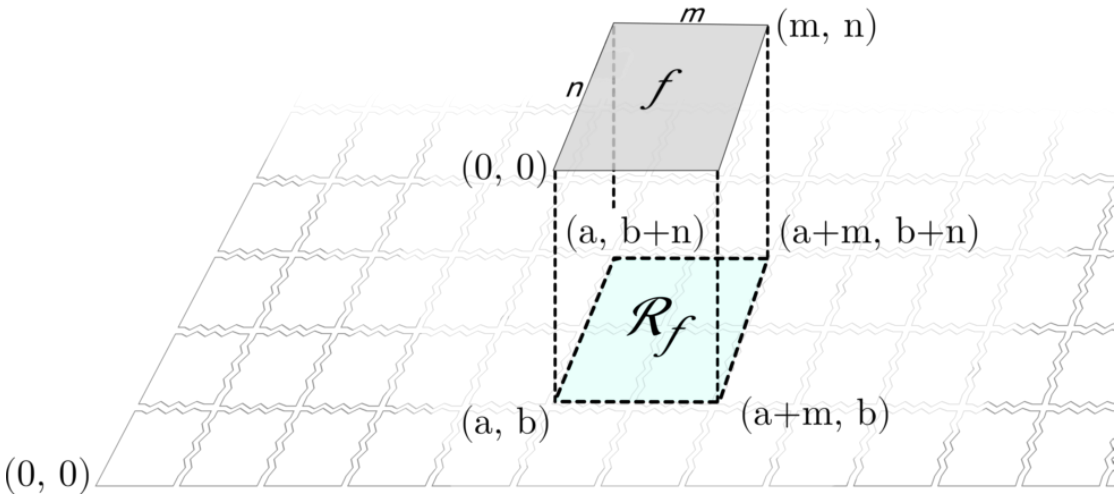


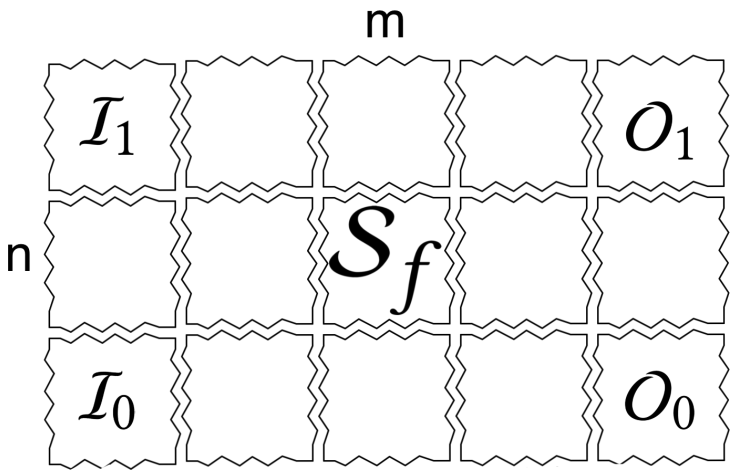
Figure 5.2: A function's virtual space is mapped to the chip's physical space.

5.3.3 Coordinate Spaces

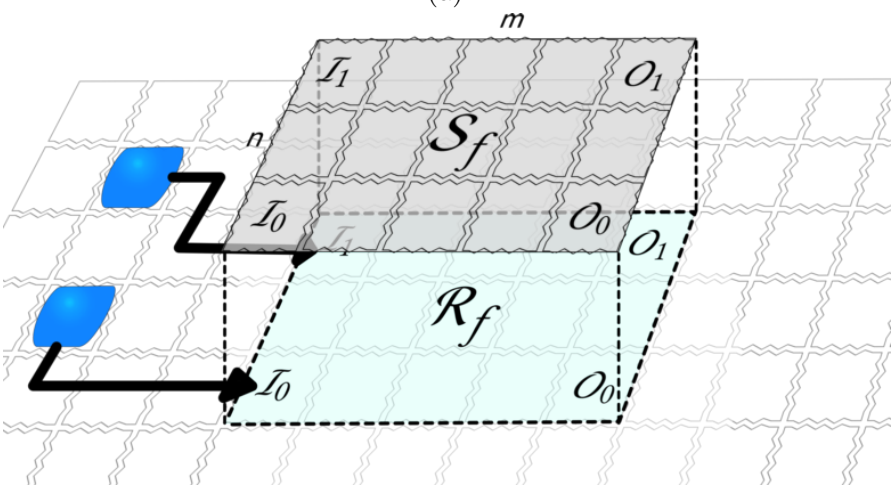
Function f can be compiled without knowing precisely where it will be placed on a DMFB. Once compiled, f can be invoked at different call sites in the program and f can be placed at a different physical location on the DMFB at each call site. To support relocation, f is compiled using a *virtual electrode coordinate space*, whose origin and dimensions are distinct from those of the *physical electrode coordinate space* corresponding to a real-world DMFB target. Once f has been placed, each electrode activation in the virtual electrode coordinate space must be translated to the physical electrode coordinate space. As shown in Fig. 5.2, if $\mathcal{R}_f$'s origin is placed at coordinate (a, b), the electrode at virtual coordinate (x, y) translates to physical coordinate $(x + a, y + b)$.

5.3.4 The Physical Function Prototype

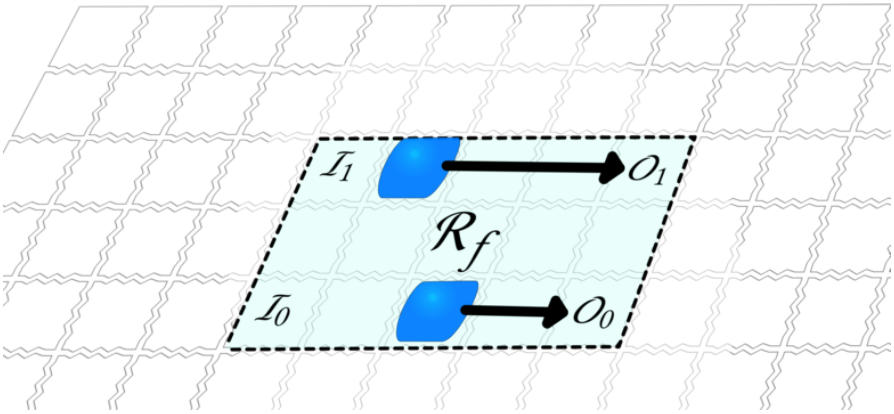
Let $\mathcal{C}_{in}$ and $\mathcal{C}_{out}$ respectively denote the initial and final positions of the input parameter droplets in $\mathcal{D}_{in}$ and output droplets $\mathcal{D}_{out}$. $\mathcal{C}_{in}$ and $\mathcal{C}_{out}$ are determined when f is compiled, and typically will be locations on the perimeter of the $m \times n$ virtual electrode



(a)



(b)



(c)

Figure 5.3: (a) The locations of all input and output droplets within a function prototype $\mathcal{S}_f$; (b) when $\mathcal{R}_f$ is determined, the translated virtual coordinates are used for routing input droplets prior to invoking f; (c) f routes all output droplets to virtual output coordinates prior to terminating.

coordinate space. This information is provided as metadata, which we refer to as f's *Physical Function Prototype*[3]: $\mathcal{S}_f = (m, n, p, q, \mathcal{C}_{in}, \mathcal{C}_{out})$ (Fig. 5.3a). We will subsequently augment the physical function prototype with additional metadata in Sections 5.3.6 and 5.3.7 as we relax some of our simplifying assumptions.

After placing $\mathcal{R}_f$, the caller translates each virtual coordinate $(x_i, y_i) \in \mathcal{C}_{in}$ to physical coordinate $(x_i + a, y_i + b)$ and routes the corresponding droplet there (Fig. 5.3b). Once all input droplets are routed, the caller cedes control to f. Prior to termination, f must route each output droplet to virtual coordinate $(x_j, y_j) \in \mathcal{C}_{out}$, which translates to physical coordinate $(x_j + a, y_j + b)$ (Fig. 5.3c). When f terminates, the caller knows the coordinates of all output droplets, and can route them to appropriate locations for subsequent processing.

5.3.5 Stack Management for Fluidic Variables

In traditional programming, variables that are live across a function called are pushed onto a stack prior to function invocation, and popped from the stack when the function terminates. The stack (Fig. 5.4a, right) is implemented in memory, and contains bookkeeping information (e.g., return address, stack and frame pointers), stack-allocated variables, along with variables that are live across the call. Each function only sees the portion of the stack allocated to its frame.

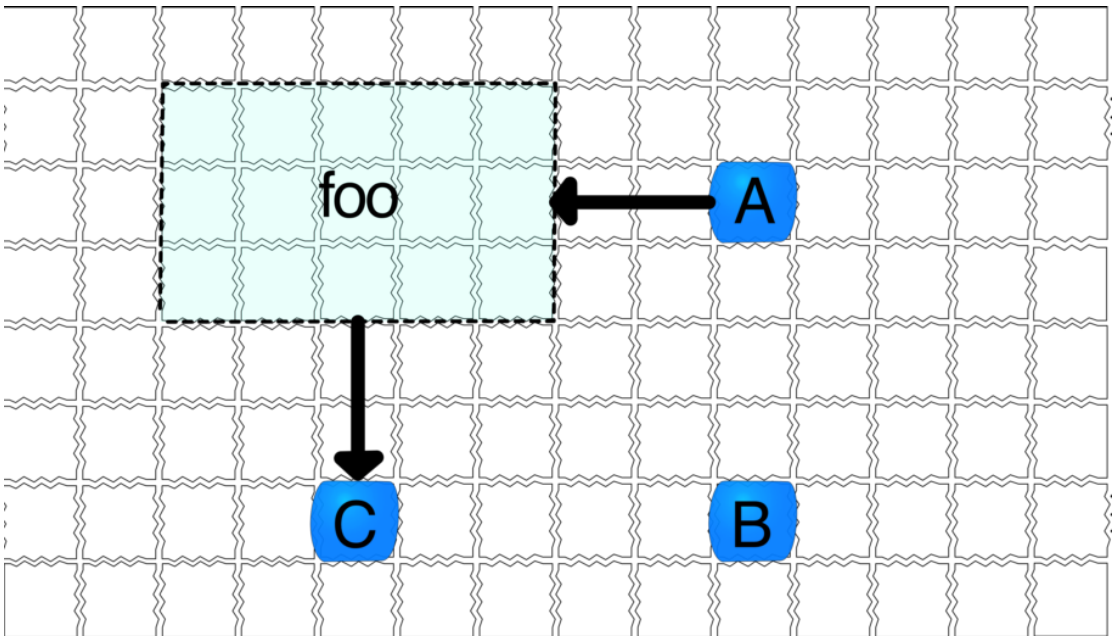
In a fluidic program, droplets that are live across a call to function f must be stored on-chip, and outside of the region $\mathcal{R}_f$ where f is placed, while any computational variables

[3]In traditional computer programming, a function's prototype specifies its type signature. Fluidic types [178] can and should be included in a fluidic function's prototype, for example, to facilitate program analyses, and type inference, but are not needed for function placement.

```
1   function foo(float, droplet);
2
3   main {
4       A = dispense ...
5       B = dispense ...
6
7       x = detect fluorescence on A
8       y = detect fluorescence on B
9
10      C = foo(x, A)
11      dispose C
12  }
```

Stack
x
y
...

(a) A traditional computing platform manages data variables using a stack.



(b) The fluid "B" must be stored on *the droplet stack*, i.e., any unused surface of the DMFB outside of function `foo`'s region for execution.

Figure 5.4: (a) A simple chemical program that features dispense statements (for fluidic values) and detect statements (for data values); data variables are stored on a traditional stack. For interleaving fluidic and data variables, we introduce a *split-technology* stack, where data variables are traditionally stored, and fluidic variables are stored on the surface DMFB prior to invoking a function (b).

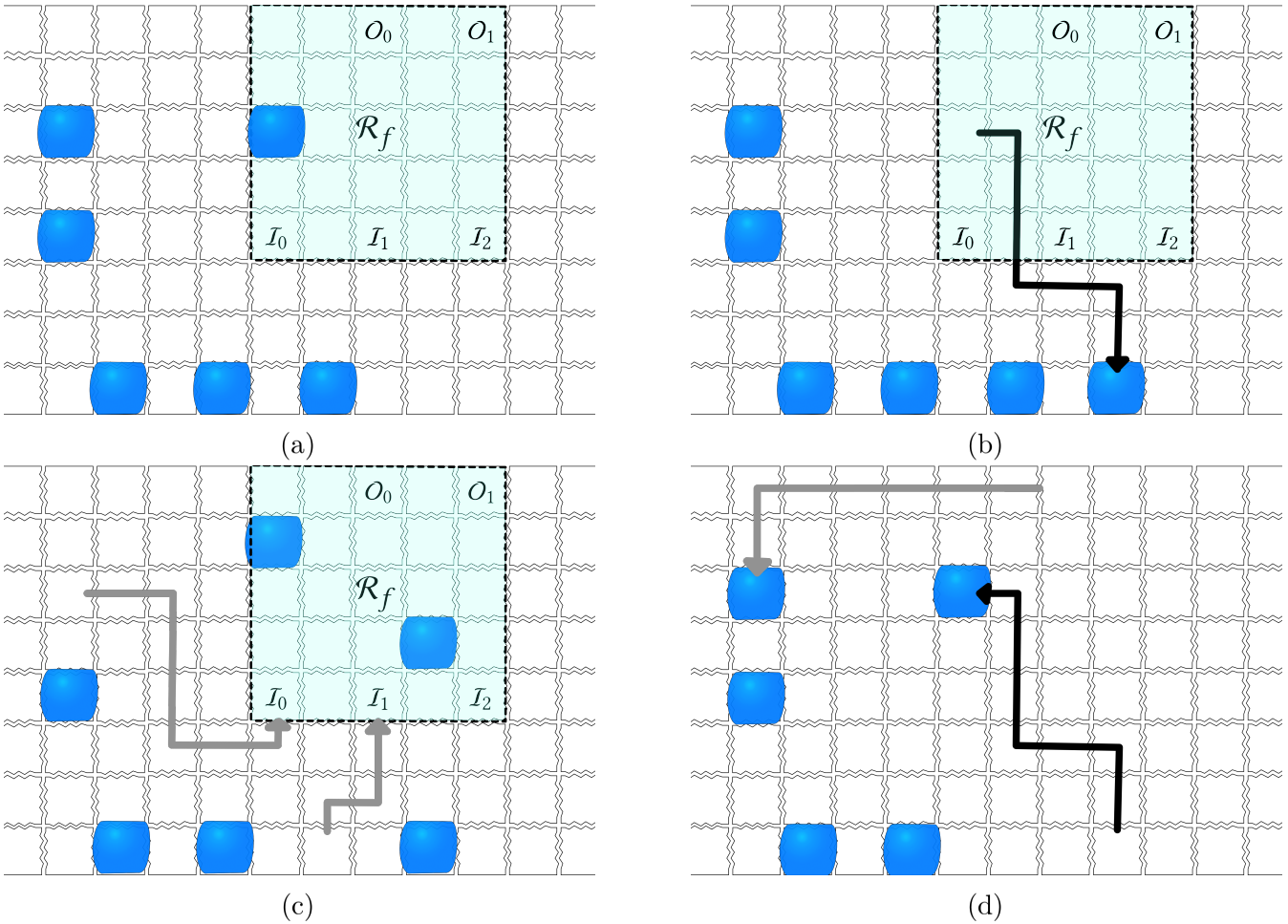


Figure 5.5: When placing $\mathcal{R}_f$ for a function call to f, existing droplets may interfere (a). In order to successfully place $\mathcal{R}_f$, all interfering droplets are routed to the droplet stack outside $\mathcal{R}_f$'s region (b). As f executes (possibly consuming droplets as it does so), live droplets across the call are maintained in the droplet stack (c). Finally, as f returns, droplets that are output by f are routed to their next location, and any droplets that were moved out of the way to place $\mathcal{R}_f$ are returned to their original locations (d).

live across the call can be pushed onto the CPU's stack. We refer to this paradigm — where data values are maintained by a CPU controller and fluidic values are maintained on the DMFB as *split-technology stack* (Fig. 5.4). The droplet stack (Fig. 5.4b) can be implemented using any unused region outside of $\mathcal{R}_f$; it does not need to be contiguous and droplets do not need to be stored in any specific order on-chip; this is to avoid lengthy routing paths that may occur if the droplet stack was pre-allocated to a specific region of the chip.

Let $d \notin \mathcal{D}_{in}$ be a droplet that is live at f's call site. The placer *pushes* d onto the droplet stack by routing it to a position outside of $\mathcal{R}_f$, as shown in Figs. 5.5a and 5.5b; this ensures that d does not inadvertently mix with droplets created by or used locally by f within $\mathcal{R}_f$. If d resides outside of $\mathcal{R}_f$, then the push operation is implicit, as routing is unnecessary. As a performance optimization, pushing d can be performed concurrently with input parameter droplet routing, as discussed in the preceding section. When f terminates execution, the corresponding *pop* operation is implicit, as the callee has immediate access to d wherever d resides.

5.3.6 External Devices

Additional care needs to be taken when compiling functions that feature operations such as heating or optical detection that are performed by external devices that are located at pre-specified locations on (or beneath) the DMFB surface. For brevity, we consider heaters exclusively, recognizing that the same principle applies to other types of devices. Heaters are often larger than one electrode; for example, `OpenDrop` offers a cartridge featuring three 2×1 heaters (Fig. 2.5b). To compile a function that features one or more

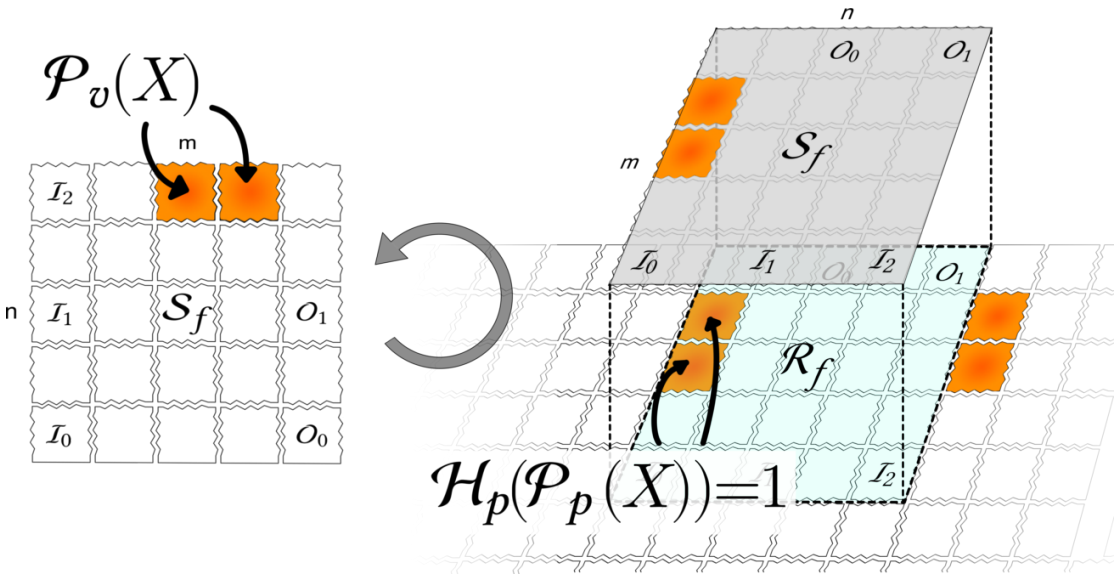


Figure 5.6: The virtual coordinates of placed heat operations within a function f's prototype are specified when the function is compiled (left). When placing f, the placer must orient and place $\mathcal{R}_f$ in the physical coordinate system such that $\mathcal{S}_f$'s virtually-bound heat operations align with heaters in the physical coordinate system (right).

heating operations, $\mathcal{R}_f$ must contain at least one *virtual heater*, and, $\mathcal{R}_f$ must be placed in such a manner that its virtual heater aligns to a *physical heater* on the DMFB.

Let X be the set of operations in a fluidic program. For a given coordinate system, mapping $\mathcal{P} : X \rightarrow \mathbb{N}^2$ that specifies the placement of each operation, and mapping $\mathcal{H} : \mathbb{N}^2 \rightarrow \mathbb{B}$ establishes the positions of the heaters, i.e., $\mathcal{H}(x, y) = 1$ if the DMFB features a heater at coordinate (x, y), and 0 otherwise). If h is a heating operation placed at coordinate $\mathcal{P}(h) = (x_h, y_h)$, the placer must ensure that $\mathcal{H}(\mathcal{P}(h)) = 1$.

We define $\mathcal{H}_p$ and $\mathcal{P}_p$ for the physical coordinate system and $\mathcal{H}_v$ and $\mathcal{P}_v$ for the virtual coordinate system $\mathcal{R}_f$. When f is compiled prior to distribution, each heating operation h must be placed at position $\mathcal{P}_v(h) = (x_h, y_h)$ satisfying $\mathcal{H}_v(\mathcal{P}_v(h)) = 1$ in the virtual coordinate system. When a program that calls f is compiled, the placer must satisfy the constraint $\mathcal{H}_p(\mathcal{P}_p(h)) = 1$ (Fig. 5.6). To accommodate this constraint, the physical

function prototype (Section 5.3.4) is extended to include the location of heaters within the virtual coordinate system (Fig. 5.6, left); notation is omitted for brevity.

5.3.7 Droplet I/O

When function f is generated as a pre-compiled binary, the compiler cannot know the precise locations of the physical I/O ports on the perimeter of the DMFB, and where $\mathcal{R}_f$ will be placed on the DMFB relative to those I/O ports. This creates a number of subtle challenges when compiling functions that perform I/O operations that require access to the ports. When f is compiled offline, the compiler can select an appropriate location on the perimeter of $\mathcal{R}_f$ for droplet entry and exit, and can compute routing paths from the entry point to the droplet's initial use point, and from the droplet's final use point to the exit respectively; the compiler cannot determine a routing pathway from/to the appropriate I/O reservoir, nor can it know the precise latency of droplet transportation.

Fig. 5.8 illustrates the preceding issues: within f, the droplet routing pathway from input coordinate $\mathcal{I}_0$ to output coordinate $\mathcal{O}_0$ within the virtual electrode coordinate space is known, and can be translated to the physical electrode coordinate space, as described in section Section 5.3.3. In contrast, the pathways from the input reservoir to $\mathcal{I}_0$ and from $\mathcal{O}_0$ to the output reservoir are not known when f is compiled; they are only known when a subsequent program that calls f is compiled.

The initial position of each input droplet and the final position of each output droplet on the perimeter of $\mathcal{R}_f$ can be added to the physical function prototype $\mathcal{S}_f$; the simplest option is to re-use the coordinates for input parameter and output droplets ($\mathcal{C}_{in}$ and $\mathcal{C}_{out}$, defined in Section 5.3.4), although there is no requirement to do so. Droplet routes

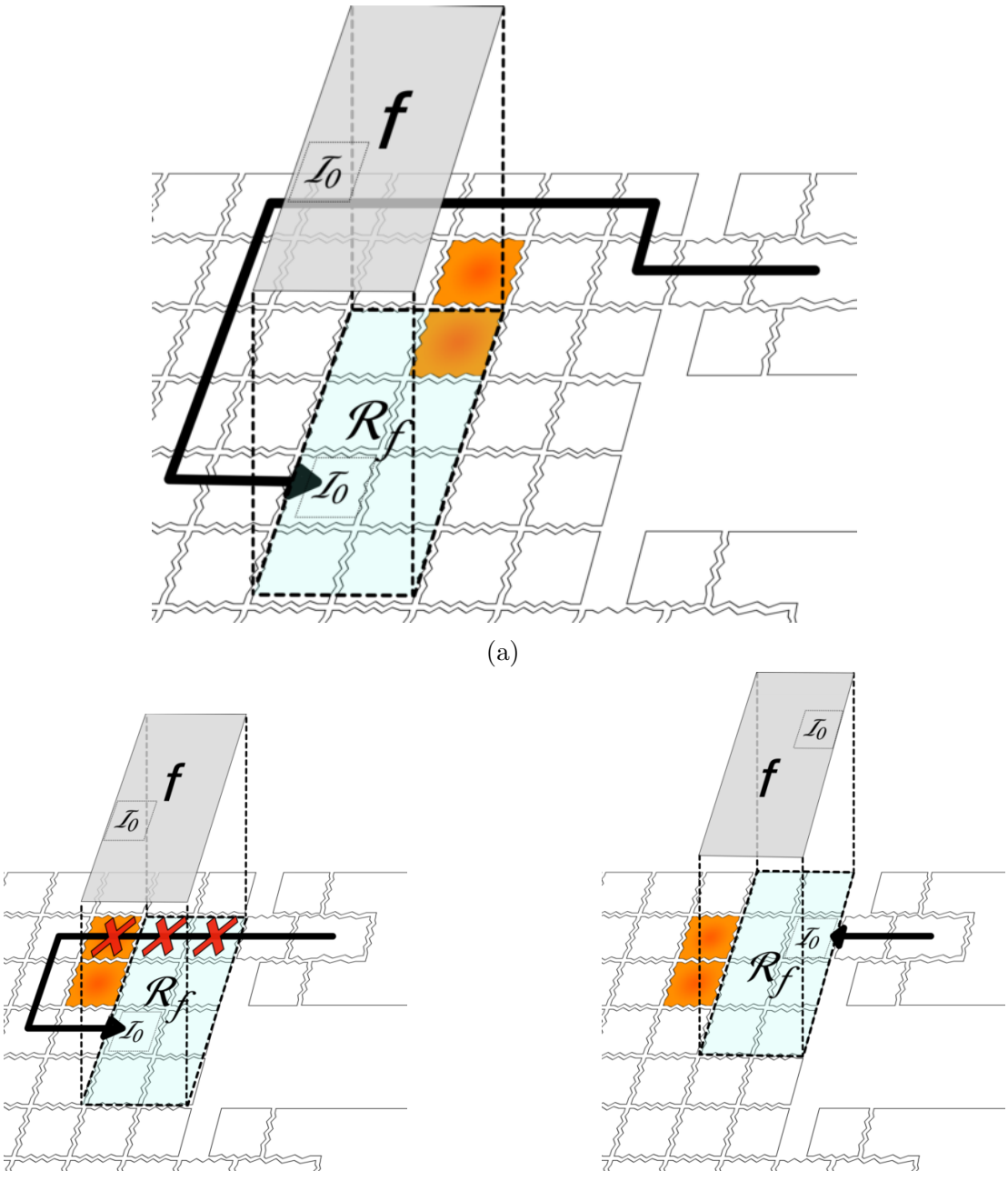


Figure 5.7: (a) A legal placement of $\mathcal{R}_f$ requires enough room around $\mathcal{R}_f$ so as to avoid inadvertent mixing of fluids within $\mathcal{R}_f$ and any routing of droplets external to $\mathcal{R}_f$. (b) Blocking a required reservoir results in failure to route, unless the reservoir has a direct path to the appropriate input at $\mathcal{R}_f$ (c)

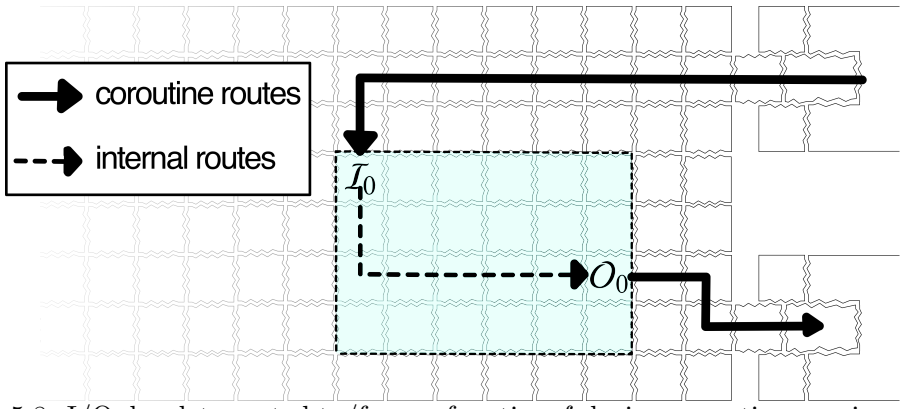


Figure 5.8: I/O droplets routed to/from a function f during execution require a coroutine external to f's control; when droplets arrive at f's boundary, their control is ceded to f.

within $\mathcal{R}_f$ can be computed when the function is compiled; the routes between I/O reservoirs and the initial/final position of each droplet, as per $\mathcal{S}_f$, can only be determined when the function that calls f is compiled.

Precisely how to implement the route between I/O reservoirs and the perimeter of $\mathcal{R}_f$ is unclear. Clearly, the caller must ensure that the placement at the call site does not block any of the required routes (Fig. 5.7). One possibility is to compile f with a *stub* that represents the partial route, and have the caller complete the stub. Another is to implement the route using a *coroutine* that executes concurrently with f (Fig. 5.8). Neither the stub nor the coroutine mechanism would be exposed to the programmer, as droplet routes are implicitly determined from dependencies between operations that define and use droplets. We oped for the co-routine approach in our implementation. It is important here to recognize that I/O operations within a function may depend on control flow conditions that can only be resolved dynamically, so it is not possible, in the general case, to statically determine precisely when I/O operations will be issued.

If the droplet route must satisfy timing constraints [134], then f can communicate these constraints to the caller/compiler by including them as non-executable metadata with

the pre-compiled binary. Without loss of generality, consider an input droplet whose maximum allowable routing time is T. Let t_f be the time spent routing the droplet from the perimeter of $\mathcal{R}_f$ to the location within $\mathcal{R}_f$ where the droplet will be used. Then the compiler must be able to route the droplet from the input reservoir to the perimeter of $\mathcal{R}_f$ within time $T - t_f$; otherwise compilation will fail.

An additional requirement is that $\mathcal{R}_f$ cannot be placed in a manner that blocks all possible routing pathways between I/O droplets and the physical I/O reservoirs on the perimeter of the chip. Figure 5.7a depicts a legally placed function in which there is a pathway from an input reservoir to position $\mathcal{I}_0$ for the droplet on the perimeter of $\mathcal{R}_f$. Figure 5.7b shows an illegal placement, under which the droplet cannot route from the I/O reservoir to position $\mathcal{I}_0$ without entering $\mathcal{R}_f$. Figure 5.7c depicts a legal placement solution in which $\mathcal{R}_f$ still abuts the input reservoir, but is rotated such that the only pathway from the input to $\mathcal{R}_f$ is through position $\mathcal{I}_0$; this rotation approach works in this specific case, but may not generalize when $\mathcal{R}_f$ becomes larger and if the placement abuts multiple I/O reservoirs.

5.3.8 Calling Context and Multiple Function Versions

Function f may be called from multiple sites in a program. At each call site, number of droplets stored in the on-chip call stack will be equal to the number of droplets live across all function calls in the chain leading to the current call site. The size of the on-chip call stack determines the maximum available on-chip area to place $\mathcal{R}_f$. Allocating more space to a function will benefit its performance because: (1) more space means that more fluidic operations can be scheduled concurrently (i.e., operation-level parallelism);

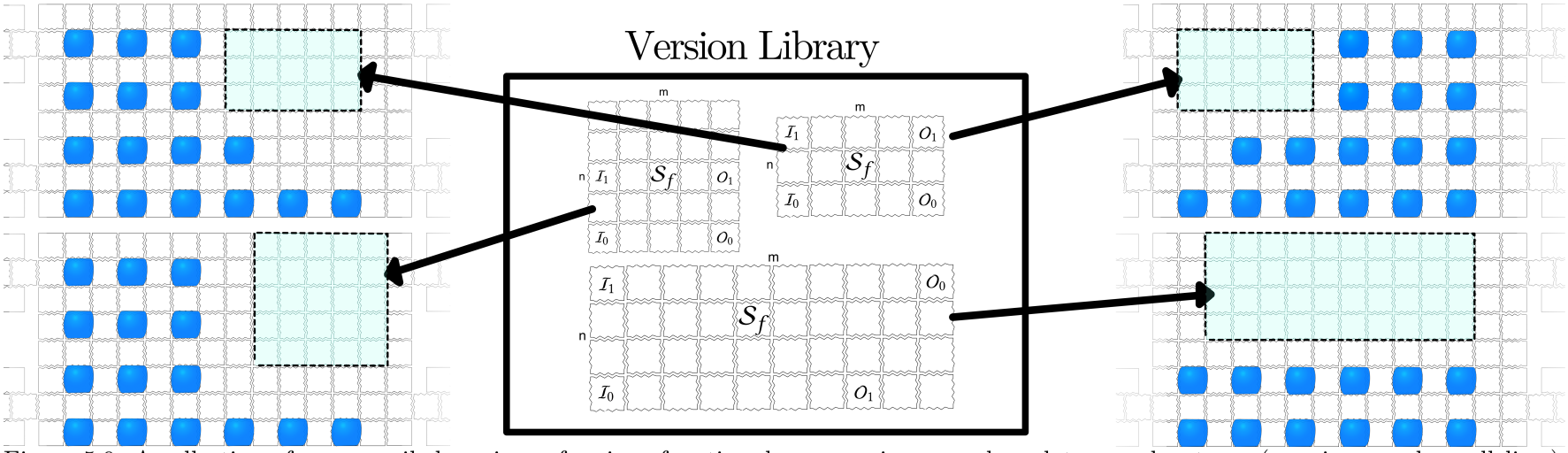


Figure 5.9: A collection of pre-compiled versions of a given function; larger versions may have latency advantages (e.g., increased parallelism), while smaller versions are able to be legally placed when a calling context limits feasible placement.

(2) prior work has shown that allocating more on-chip space to mixing operations can reduce mixing latency [180, 240, 241, 133].

If a single version of a function is provided, then it should be small enough to satisfy the placement constraint at all call sites in the program; otherwise, compilation will fail; however, better performance can be achieved at each call site (assuming that f exhibits ample parallelism) by compiling a unique version of f that optimally uses the space available.

If f is provided as a pre-compiled binary, then the calling context is not known when f is compiled. While compiling f to be as small as possible will maximize the likelihood that programs that call f can compile successfully; however,f will perform suboptimally at call sites where more than the minimum amount of space is available.

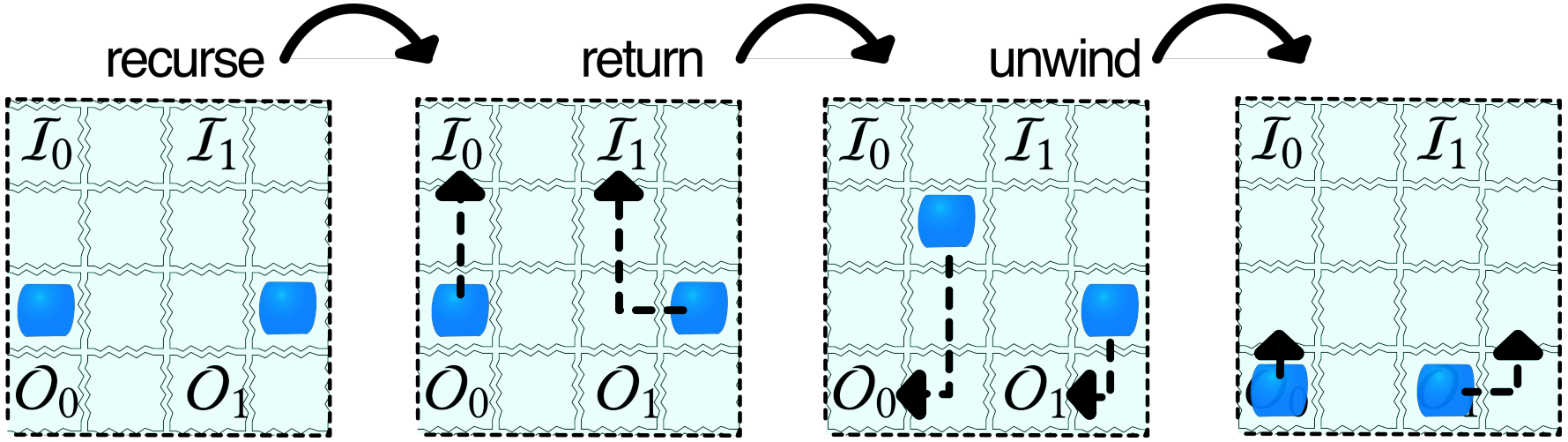
One solution is to pre-compile multiple versions of f with different area constraints, i.e., with different values of m and n (Fig. 5.9); while enumerating every admissible combination of $m \leq M$ and $n \leq N$ may be prohibitive, providing a few different versions can create relatively low-cost opportunities for the compiler to optimize performance by invoking a call to the best-performing implementation of f that satisfies placement constraints at each call site.

5.3.9 Recursion

Recursive function calls create additional challenges. We start with the simple case: tail recursion.

```
1  function foo(a, b) {
2      ab = mix a with b
3      c = split ab into 2
4      drain c[0]
5      f = detect fluorescence
           on c[1]
6      if ( f >= ... ) {
7         return c[0]
8      }
9      d = dispense ...
10     return foo(c[0], d)
11 }
```

(a)



(b)

Figure 5.10: (a) A (tail)recursive function that does not increase the size of the stack; such a function does not pose any threat to resource waste. Set-up and unwinding of recursive calls can be handled by the function without invention from the caller (b).

Tail Recursion: We first consider tail recursive calls that do not increase the size of the stack (e.g., Fig. 5.10). In this case, any legal placement of $\mathcal{R}_f$ will suffice, as all calls to f will occupy the same region. As shown in Fig. 5.10b, prior to each recursive call, all parameter droplets $\mathcal{D}_{in}$ must be routed to their corresponding virtual coordinates $\mathcal{C}_{in}$ on the perimeter of $\mathcal{R}_f$; and at each point where f may terminate, the compiler must route the set of output droplets $\mathcal{D}_{out}$ to their corresponding virtual coordinates $\mathcal{C}_{out}$ on the perimeter. Based on these assumptions, unwinding such a recursive call chain can be handled without any intervention from the caller.

General Recursion: In the general case, recursive functions can be written that create fluids that are live across each recursive call site, thereby increasing the size of the stack. This means that each successive call will have less available area, and that the DMFB will eventually run out of space after a statically-computable maximum call depth. Any recursive call sequence that exceeds the maximum call depth will fail.

If source code or the intermediate representation is available, general recursion can be implemented using an interpreter that performs *Just-In-Time (JIT)* compilation (Fig. 5.11). At each recursive call site, the interpreter can JIT-compile the function using fast-running SPR algorithms. The program terminates prior to completion if SPR fails due to lack of space. If static compilation is preferred, the recursive calls can be inlined up to the statically-computable maximum call depth, including termination criteria if the maximum call depth is exceeded during execution.

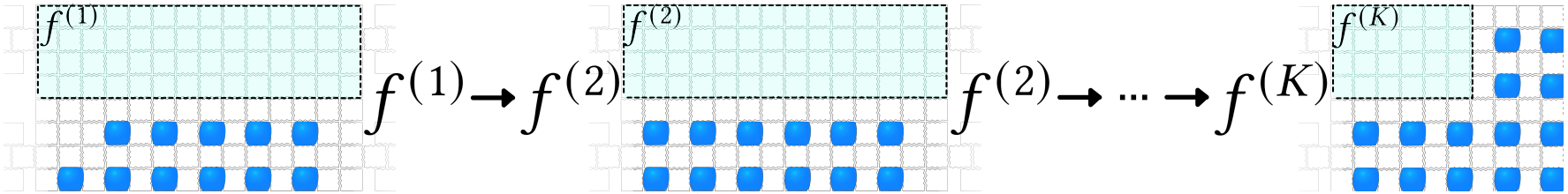
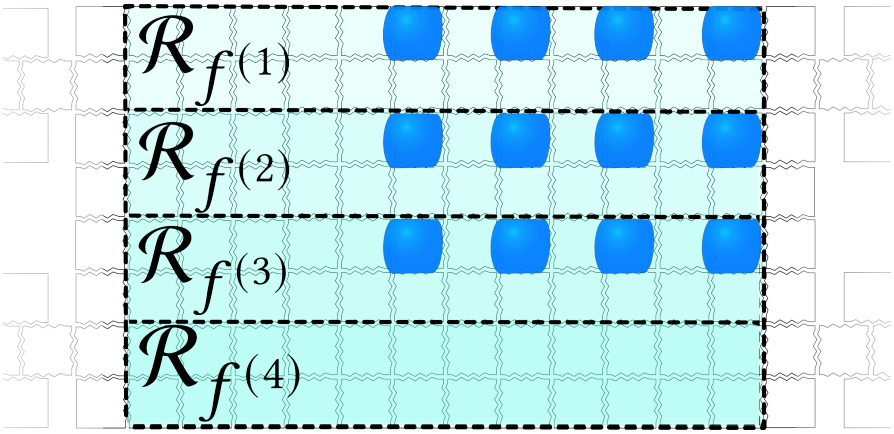


Figure 5.11: When source is available, a *Just-In-Time* (*JIT*) approach can maintain the droplet stack using the entire DMFB.

```
1  function f(I0) {
2      d1 = dispense ...
3      d2 = dispense ...
4      d3 = dispense ...
5      d4 = dispense ...
6      if (...) {
7          // I0, d2, d3, d4 all alive
8          ... = f(d1)
9      }
10     ...
11 }
```

(a)



(b)

Figure 5.12: (a) Pseudocode for a recursive function in which 4 droplets live across recursive calls; (b) when using pre-compiled recursive functions, we compile to a maximum call depth K ($K = 4$, here), where droplets live across a recursive call to $f^{(i+1)}$ from $f^{(i)}$ are stored within $\mathcal{R}_{f^{(i)}}$.

A pre-compiled binary implementation of a recursive function is somewhat more subtle. Let K denote the maximum call depth. The compiler can generate a sequence of function calls $f^{(1)}, f^{(2)}, \ldots, f^{(K)}$, in which $f^{(i)}$ recursively calls $f^{(i+1)}$ with an appropriate decrease in available space due to stack growth, i.e., $\mathcal{R}_{f^{(1)}} > \mathcal{R}_{f^{(2)}} > \ldots > \mathcal{R}_{f^{(K)}}$ (Fig. 5.12). The compiler will need to augment $f^{(K)}$ with termination criteria to prevent recursive calls beyond the maximum call depth. The precompiled binary would consist solely of $f^{(1)}$ and its associated physical function prototype $S_{f^{(1)}}$; the subsequent recursive calls to $f^{(2)}, \ldots, f^{(K)}$ would be implementation choices that would not be exposed to the caller. If desired, the compiler could generate multiple versions of $f^{(1)}$ with varying dimensions $m \times n$, and varying maximum call depths K, in accordance with the discussion in the preceding section.

5.4 Evaluation

5.4.1 Implementation

We implemented the strategies discussed in the previous section within `BioScript`'s compiler [178, 133, 134] and added syntactical constructs to `BioScript`'s grammar to support loading pre-compiled libraries. We continue to inline (non-recursive) functions where source code is available. We provided the compiler with an architecture specification matching `OpenDrop`'s standard 8×14 cartridge with its four I/O reservoirs and 3 heating regions as shown in Fig. 2.5b. As `BioScript`'s compiler does not support physical execution (it only targets a simulator), we implemented a simple translator and execution engine[4] to serially communicate with `OpenDrop`. While `OpenDrop` provides real-time capacitance measures that can be used for droplet tracking, volume estimation, and error

[4]Available at `https://www.github.com/tlove004/MFSimToOpenDrop`.

```
1  function thermocycle3(sample, n_iter, temp1, time1, temp2,
       time2, temp3, time3) {
2      repeat n_iter times {
3          heat sample at temp1 for time1
4          heat sample at temp2 for time2
5          heat sample at temp3 for time3
6      }
7      return sample
8  }
```

Figure 5.13: An example of a simple thermocycling function that accepts 3 time and temperature pairs, in addition to a number of iterations to cycle. A droplet-replenishing version (i.e., due to evaporation [104]) can be achieved by including a volume-sensing operation that directs distilled water to be added to the sample as necessary.

detection/correction, we have not yet implemented these features as they are not needed to support function calls.

We also ran each benchmark on `BioScript`'s cycle-accurate simulator using an abstract DMFB with the same dimensions as `OpenDrop`. Some benchmarks require more I/O than `OpenDrop` provides; these were evaluated using the simulator exclusively.

5.4.2 Benchmarks

To the best of our knowledge, prior benchmark programs for DMFBs were limited in scope and did not utilize functions (e.g., see Refs [41, 178, 133, 134, 239]). We rewrote some of these benchmarks to utilize functions, and also wrote several synthetic benchmarks that were aimed specifically to elucidate the technical concepts presented in the preceding sections. All of our benchmarks are specified using `BioScript`, and are available in Appendix B.3.

When translating pre-existing benchmarks, we extracted sub-steps as functions that, in our opinion, could reasonably generalize to other domains, and would represent candidates for distribution as pre-compiled binaries. For example, we define a function for *thermocycling* (Fig. 5.13, used in PCR benchmarks), a common sub-step in assays requiring

DNA replication, where a DNA sample's temperature is cycled between cooler/warmer temperatures for a user-specified number of iterations.

5.4.3 Setup

When possible, we target the `OpenDrop` device depicted in Fig. 2.5 using alternating current of 240 V at 1000 Hz[5]. When targeting `OpenDrop`, our primary concern is the fidelity of the specification (e.g., droplet transport, mixing, etc., occurring as expected) as opposed to implementation and evaluation of chemical or biological reactions. Experiments were performed using inert input fluids[6] rather than chemical reagents.

5.4.4 Discussion

We assess our approach by comparing the loading and executing of *pre-compiled* functions against equivalent programs where the same functions are inlined, and, for recursive functions, by measuring the maximum call depth K we can statically compile. Tables 5.1 and 5.2 list the results for results obtained using a cycle-accurate DMFB simulator.

Non-recursive For each non-recursive benchmark in Table 5.1, we report a direct comparison of inlined functions against loading equivalent pre-compiled libraries listing the total routing times (in ms) execution times (in s). We observe an average ∼2% increase in total execution times[7] over inlining when source is available, but note that these results have no comparison when only a pre-compiled library is at hand. As pre-compiled functions are pre-determined, the time spent routing (outside of functions) is

[5]As suggested by OpenDrop.
[6]The clear, red, and blue fluids optimized for `OpenDrop`, available here.
[7]Using the geometric mean over ratios.

Table 5.1: Results comparing execution of inlined vs. pre-compiled non-recursive functions on a cycle-accurate DMFB simulator

		Total Time	
Name	**Execution−Type**	**Routing (ms)**	**Execution (s)**
Synth1†	Inlined	1150	18.1
	Pre-compiled	710	19.2
PCR [6]	Inlined	410	590.5
	Pre-compiled	100	590.5
DRPCR [104]	Inlined	410	592.5
	Pre-compiled	100	593.5
	avg-diff:	***66.76%***	***-2.04%***

Table 5.2: Results for compiling recursive functions, where recursive calls may generate additional droplets; a droplet-generation factor (**DF**) and maximum depth before failure **K** is reported for compiling to an 8×14 electrode grid.

Name	**DF**	**K**
SynthTail†	0	∞
SynthHead†	1	3
ProteinSplit† [221]	2	1

reduced by ∼66% on average, although the time spent routing is comparatively minimal with respect to total execution times.

Recursive For the recursive benchmarks in Table 5.2, we report a droplet generating factor **DF** (i.e., how many droplets are added to the droplet stack at a recursive call) and maximal call depth **K** before failure. While benchmark SynthTail is able to execute without fail to arbitrary call depth, the other two benchmarks unsurprisingly reach maximum call depth quickly, as the 8×14 electrode grid quickly has its free regions consumed. Execution and routing times are dependent upon the call depth; hence, we omit their inclusion. Notably, we ran these benchmarks again after inlining $K+1$ recursive calls, and were also unable to compile at this depth; the restricted grid size was untenable to find legal SPR results.

`OpenDrop` Benchmarks capable of executing on `OpenDrop`'s architecture are marked with a †. For recursive functions, we statically compiled to the discovered maximum call depth K. We were able to successfully compile electrode activations for all benchmarks targeting `OpenDrop`, and used the translator/execution engine described in Section 5.4.1 to serially control the device.[8] For control-decisions requiring sensory feedback, we simulate feedback data (using pseudo-random number generation within reasonable bounds) as proof-of-concept modulo online monitoring equipment. All marked benchmarks suc-

[8] An example of execution is linked in Appendix B.3.

cessfully transported droplets as expected; as far as we are aware, this is the first time a high-level language has been compiled for direct execution on a commercially-available DMFB.

5.5 Conclusion

In this chapter, we presented important advancements for the programmability of DMFBs, enabling scientists to disseminate experimental chemical protocols in the form of parameterized functions. Despite unique challenges in dealing with physical fluids when setting up and executing a function call, we show that static compilation is tenable, preserving strong guarantees of success that are important when expensive reagents are in use. Proof-of-concept results show that our methods are applicable to real-world architectures available for use today. While there is room to investigate further application of our methods (e.g., to inherently dynamic programming models like `Puddle`'s), we believe the contributions presented herein enable scientists to accelerate the sharing of repeatable chemical experimentation on DMFBs, and can assist in overcoming the ongoing repeatability crisis.

Chapter 6

Conclusion

Programmable labs-on-a-chip (pLoCs) have promising benefits that could disrupt the reproducibility crisis in the life sciences, while reducing costs, increasing safety, and accelerating analytical results of biochemical assays. Despite the many promises that pLoC devices offer, their unwieldy operation prevented them from being more widely adopted. While languages and compilers for specifying and synthesizing assays have reached a relative level of maturity, gaps between front and back-ends revealed that a ready-to-adopt end-to-end workflow was out of reach. This part presented necessary additions to the workflow to close these gaps, and demonstrated their utility by executing assays on a readily available open source and open hardware pLoC. With identified gaps closed, scientists have a more manageable path toward adopting these devices into their workflows.

Part II

MediSyn: A Modular Pharmaceutical Discovery and Synthesis Framework

Chapter 7

Introduction

With few exceptions, the process of discovering and developing a new drug takes anywhere from 10-15 years and costs upwards of $2.6 billion US [169]. Despite this overhead, modern medicine has increased the duration and improved the quality of life around the globe, and has given rise to a $500 billion US pharmaceutical enterprise. In spite of this, nine of the ten leading causes of death in the US are directly related to disease [30]. Moreover, new disease-causing pathogens continue to emerge and wreak havoc on humanity, triggering years-long searches for new drugs.

In this part, we adopt aspects of *inductive program synthesis* to the domain of drug discovery, where input/output examples form a specification of drugs operating on target pathogens, and programs are string representations of a drug's primary structure.

Consider the problem of *inductive program synthesis*, where the goal is to find a program conforming to a particular semantic and/or syntactic form that satisfies a specification

given as input/output examples[107, 82, 188].[1] Solutions typically rely on domain-specific languages to constrain possible program spaces and employ various search strategies to find a program satisfying a set of specifications.

Rather than considering candidates as *satisfying* the specification, we frame the goal as a *superoptimization* task [200, 4, 196], where candidate programs are ranked, and the result is a distribution of drugs of interest that can be further explored in a wet-lab environment.

The rest of this part presents a new framework for the safe discovery and synthesis of pharmaceutical drugs we call *MediSyn*. After reviewing preliminary background on drug discovery and the various methods from superoptimization, program synthesis, and language modeling we adopt in Chapter 8, Chapter 9 presents *MediSyn*'s modular architecture. *MediSyn* provides a general-purpose superoptimizing search strategy implemented using a Markov Chain Monte Carlo approach over a probabilistic context-free grammar, leaving grammar specification, chemical synthesis, and evaluation routines for users to define. As a proof-of-concept, Chapter 10 provides a prototype implementation of *MediSyn* called *PepSyn* for the discovery of novel pharmaceutical peptides. *PepSyn* introduces a domain-specific language called *PepSketch* for concise domain specification, and a secondary process in the *programming by example* (PBE) paradigm we call *PepGen* for generating a candidate domain from user-provided examples. An *in silico* technique for estimating antimicrobial peptide activity is provided in order to drive the superoptimizing search. Chapter 11 discusses results using the *PepSyn* implementation of *MediSyn*, with focus on the expressiveness of the candidate spaces generated using both *PepSketch*

[1] This is in contrast to the corresponding machine learning task, where semantic/syntactic forms are irrelevant to the solution.

and *PepGen* approaches, and the potential for significant cost savings achievable by utilizing statistical inference during optimization. Finally, Chapter 12 discusses some loosely related work in grammar induction for bioinformantics tasks as well as machine learning approaches to drug discovery before concluding with directions for future work in Chapter 13.

Chapter 8

Preliminaries

8.1 Drug Discovery and Development

The majority of the normative 10-15 year process of developing a drug before its sale occurs at the beginning of the process, during discovery and development [169]. Figure 8.1 depicts an overview of the process: in the early stages, candidates are designed and tested in controlled settings to determine mechanisms of action, dosage, potential toxicity, etc. During this stage, pharmaceutical researchers typically employ high-throughput screening to conduct millions of chemical experiments in parallel with the goal of finding candidate compounds with interesting activity in relation to a target [29]. When interesting results (hits) are identified, a process of optimization refines specific compounds (leads) for further study. This process is both inefficient and costly, causing extended waits for the creation of lifesaving new medicines, and significant chemical waste. The approach presented in Chapters 9 and 10 posits that program synthesis techniques can efficiently

guide these processes, significantly reducing the cost and time associated with early drug development stages.

8.1.1 Antimicrobial Peptides

PepSyn (Chapter 10) synthesizes candidate programs as synthetic peptides.[1] A *peptide* is a short linear chain of amino acids – small organic compounds formed primarily from nitrogen, carbon, hydrogren, and oxygen. As opposed to larger chemical structures (e.g., proteins, small molecules, or macromolecules), peptides are relatively simple, featuring up to around 50 amino acids, and lacking a stable 3D structure [48].[2] A peptide's *primary sequence* is a string consisting of single-character representations of its constituent amino acids (see Table 8.1). *Antimicrobial peptides* (AMPs) are a class of naturally occurring peptides whose mechanism of action targets a cell's membrane, leading to cell death [74], and have been shown to evade pathogenic bacteria's ability to develop resistance [90]. Moreover, while AMPs have been observed for their antimicrobial affect (hence the name), recent research has shown they have broad-spectrum application (i.e., against viruses, cancers, etc.) [166], and searches for synthetic AMPs (i.e., not naturally occurring) are ongoing [164]. AMP's chemical simplicity, robustness against drug-resistance, and potential for broad application make them compelling candidates for new drugs.

[1] An expanded discussion of peptides and their *physical* synthesis is available in Appendix D.

[2] There is no consensus on the exact number of amino acids that differentiates a peptide from a protein [225]; the peptide classification (even for chains reaching > 50 amino acids) is used herein for consistency.

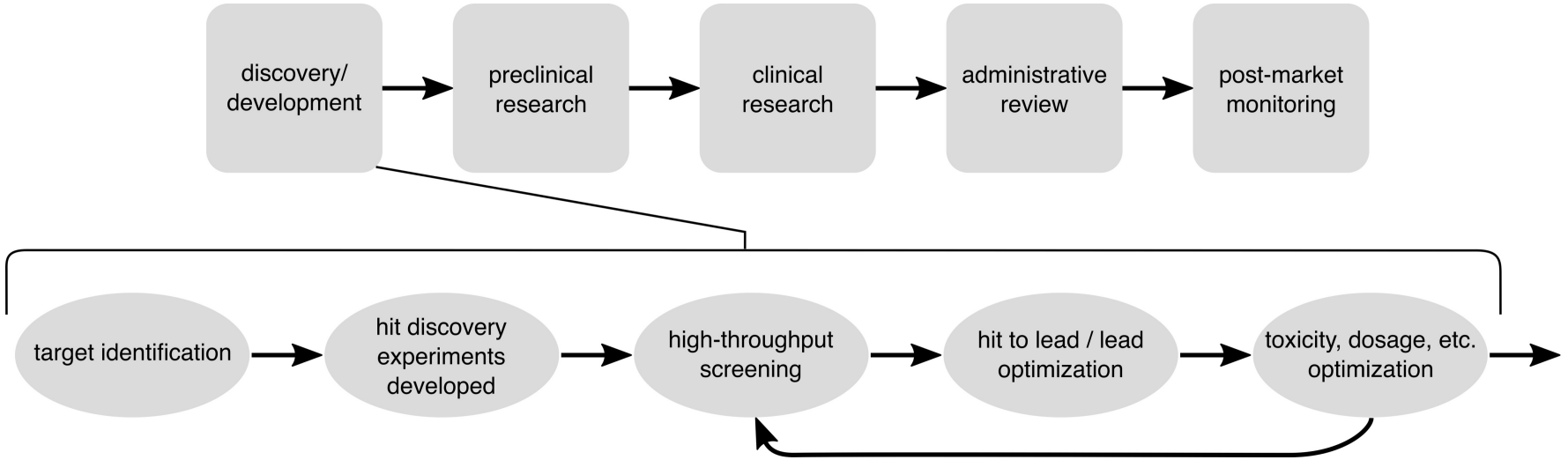


Figure 8.1: Top: A high-level overview of the drug development process; bottom: an exploded view of the discovery phase.

Table 8.1: The 21 proteinogenic amino acids found in the genetic code for eukaryotic organisms (including humans), and their single-letter abbreviations, organized by classification.

Hydrophilic (Σ_ζ)			
[+]Arginine	R	[+]Histidine	H
[+]Lysine	K		
[-]Aspartic Acid	D	[-]Glutamic Acid	E
[0]Serine	S	[0]Threonine	T
[0]Asparagine	N	[0]Glutamine	Q
Hydrophobic (Σ_Φ)			
Alanine	A	Valine	V
Isoleucine	I	Leucine	L
Methionine	M	Phenylalanine	F
Tyrosine	Y	Tryptophan	W
Other (Σ_O)			
Cysteine	C	Selenocysteine	U
Glycine	G	Proline	P
[+]: Positive	[-]: Negative	[0]: Uncharged	

8.1.2 Drug Efficacy Evaluation

The process of discovering, optimizing, and determining dosage of hits relies on various target-specific evaluation experiments. For example, in determining hits, a target (i.e., virus, bacterium, cancer cells, etc.) is typically exposed to a drug candidate to observe its effect on the target.

A primary metric utilized when determining hits and the dosage of a candidate drug against bacteria of interest is minimum inhibitory concentration (MIC) [228]. A drug's MIC against a target bacterium is the minimum concentration (typically expressed in μg/mL) required for a drug to visibly inhibit the growth of the target. The procedure of determining MIC *in vitro* involves subjecting a cultured target bacteria to varied dilutions of the candidate drug [237]. Chapter 10 discusses the use of user-provided input/output examples of peptides and their MICs in relation to target bacteria in order to generate a cost function to optimize over, in addition to constraining the program space in the *PepGen* approach presented therein.

8.1.3 Cost Considerations

Synthesizing and evaluating candidate drugs can be costly; for peptides, estimates range from around $75-$600 per gram for small-scale operations [87, 27], with costs decreasing as the bulk of product goes up [119]. Even with economies of scale, large pharmaceutical companies, seeing flat revenues due to consistently high costs, are abandoning research and development on pharmaceutical peptides [189]. We discuss cost estimates for fully *ex silico* drug discovery using our methods in Chapter 11.

8.2 Superoptimization and Program Synthesis

Superoptimization tasks a code generator with automatically finding (near) optimal and correct instructions for provided loop-free code. While the original formulation relied on complete and exhaustive program enumeration ([148]), limiting the feasibility of the technique's application, subsequent work has expanded the scope (up to complete program synthesis) through e.g., stochastic techniques [201, 4, 196] or aggregating hypotheses [110, 224]. We take inspiration from Refs. [201, 4, 196] when implementing the optimizing search process for *PepSyn*. The optimization process described in [201] aims to find an optimized form of a provided loop-free sequence of instructions using a Markov Chain Monte Carlo (MCMC) optimizing search. While superoptimization typically optimizes over the length of a program, Chapter 10 provides a function that optimizes over a peptide's MIC against target pathogens. The MCMC implementation presented in Chapter 9 closely resembles [196]'s approach, which, in a manner similar to [4], describes a Metropolis-Hastings MCMC ([88]) technique on a prior distribution represented by a probabilistic context-free-grammar (PCFG), and a jumping distribution on the parse tree used to generate an expression from the PCFG. Ref. [71] provides further inspiration; it

reveals an annealing method for MCMC that provides faster mixing (convergence to a steady-state) times than a stock MCMC approach, by dynamically adjusting exploration and exploitation via a tempering variable attached to the acceptance ratio.[3]

Automated code synthesis — in the context of alleviating the difficulty in programming a particular task for an end-user — has evolved from the relatively simple loop-free code optimizers (with goals similar to the superoptimizers discussed above), to the burgeoning field of program synthesis, where a user provides a specification of what a program is supposed to do, a search method to explore program candidates, and some form of evaluation for a candidate's efficacy. Recent algorithmic breakthroughs has allowed larger program spaces to be explored, with space-constraining techniques such as the SKETCH paradigm [212], and aggregating data structures such as version space algebras [162, 117, 209], e-graphs or finite tree automatons [110, 235, 172, 238], or the recent equality-constrained tree graphs [115] provide efficient means of representing and searching/validating exponential equivalent specifications. Methods of searching the program space has seen equal attention, with powerful deductive top-down approaches [82, 188], type-directed methods [66, 186, 182, 85, 161, 115], and stochastic techniques [4, 122, 21, 61, 152] providing state-of-the-art results.

PepSyn borrows ideas from *sketch*- [212] and *template*-based [219] program synthesis directly in the design of the *PepSketch* approach for *PepSyn* (Section 10.1.1). Refs. [212, 219] rely on *syntactic biasing* in their candidate specifications, where the program synthesis job is to find correct programs by filling in *holes* – program points that are intentionally left unspecified. With *PepSketch*, we provide programmers a straightforward

[3]While simulated annealing is itself an adaptation of the Metropolis-Hastings MCMC algorithm, the way *MediSyn* utilizes the tempering principle is not with the acceptance of samples, but as an adjustment to the jumping distribution's domain.

method to specify syntactic and semantic forms for correct peptide expressions, while allowing the superoptimization process to fill in holes.

The *PepGen* approach, which generates a space of peptide candidates using input/output examples (see Section 10.1.2) is inspired by the *programming by example* (PBE) paradigm [82, 128, 60, 209, 160], where user intent is specified by the provided examples.

8.2.1 Probabilistic Context-Free Grammars

Enumeration of programs is a crucial element of the program synthesis paradigm; statistical techniques relying on probabilistic models for suggesting programs with high likelihood of satisfying specifications are gaining in popularity, as they have enabled exponentially larger program spaces to be explored efficiently [122, 21, 61, 152, 15, 128]. Complete enumeration of the space of peptides is intractable (not to mention costly, should physical synthesis be part of the loop); consequently, the specification a user provides to *PepSyn* (either a *PepSketch* expression or through the *PepGen* UI) is transformed into a probabilistic context-free grammar (PCFG) for generating peptide candidates Sections 10.1.1 and 10.1.2. For the *PepGen* UI, the transformation process relies on input/output examples for learning *bias* in the resulting PCFG, an approach providing efficient state-of-the-art results [107, 122, 152, 182, 60, 85, 145, 161].

While the complexity of biological sequences include dependencies that would require context-sensitive or even unrestricted grammars to fully capture, the most common dependencies observed in peptides are limited nesting and/or branched dependencies, with crossing (e.g., repetition, copying) dependencies only rarely occurring [203]. Although crossing dependencies require the expressive power of at least context-sensitive grammars, the expressive power of CFGs can model both the branched and nested dependencies

commonly occurring in peptide sequences, so their modelling power is sufficient for our candidate space.

A CFG precisely describes how to derive syntactically legal strings in its language; a PCFG extends this capability with probabilities assigned to production rules. The objective of *PepSyn* is to derive drug candidates with a higher likelihood of pharmaceutical activity. We therefore adopt a PCFG as a baseline representation of a candidate search space, where the initial distribution of candidates the grammar derives corresponds to their pharmaceutical activity. For well-understood domains, the probability distributions may be known; however, in the general case, it is necessary to determine the probabilities empirically from data.

Formally, a *PCFG* is a tuple $G = (\mathcal{N}, \Sigma, S, \mathcal{R}, \mathcal{P})$, where $\mathcal{N} = \{N^1, \ldots, N^n\}$ is a set of n nonterminal symbols, Σ is an alphabet (i.e., a set of terminals), $S \in \mathcal{N}$ is the start symbol, $\mathcal{R}$ is a set of production rules mapping each nonterminal $\mathcal{N}^i \in \mathcal{N}$ to $j_i \geq 1$ sequences of terminals (and/or nonterminals) of the form $\mathcal{N}^i \rightarrow \rho_j^i$, where ρ_j^i is the j^{th} right-hand-side production rule associated with $\mathcal{N}^i$, and $\mathcal{P} : \mathcal{R} \rightarrow (0, 1]$ is a function mapping production rules to continuous probabilities such that:

$$\sum_{\rho=\rho_j^i} \mathcal{P}(\mathcal{N}^i \rightarrow \rho) = 1, \qquad \forall \mathcal{N}^i \in \mathcal{N}$$

Given a CFG, the probabilities associated with production rules can be estimated from unlabeled data using the inside-outside algorithm [13]:

$$\mathcal{P} : (\mathcal{N}^i \rightarrow \rho_j^i) \rightarrow \frac{Count(\mathcal{N}^i \rightarrow \rho_j^i)}{Count(\mathcal{N}^i)}, \qquad \forall \mathcal{N}^i \rightarrow \rho_j^i \in \mathcal{R}$$

Alternatively, if a grammar is not provided, a *treebank*, i.e., a corpus of parse trees (typically annotated with part-of-speech tags) [147], can be used to obtain a PCFG

by first extracting the underlying grammar (Fig. 8.2) before computing the maximum likelihood of each rule. We generate a treebank in Section 10.1.2 from user-supplied input/output examples for this purpose.

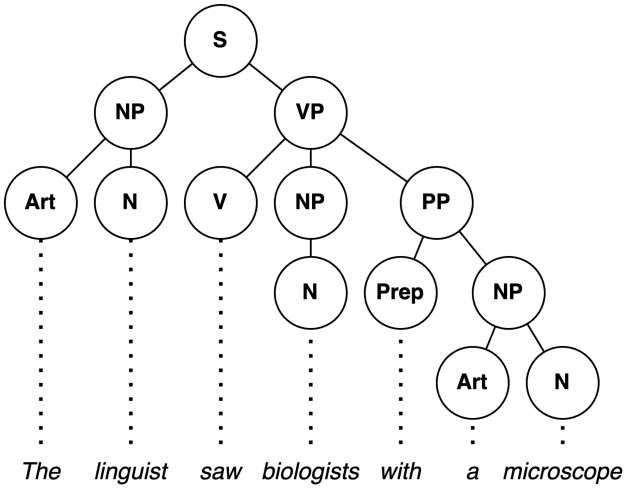
8.3 Word Embeddings and Semantic Clusters

In Section 10.1.2, we present a PBE-inspired approach to inducing a PCFG from example peptides provided by a user. To do so, we rely on techniques that provide lexical and relational semantic interpretation to generate a treebank for PCFG induction. Specifically, we learn distributed representations of *3-mers* — biological "words" consisting of 3 amino acids — that embed pairwise similarity by proximity in the distributed space. These representations are used to structurally tag unlabeled peptide sequences to form a treebank for PCFG induction. We present a brief introduction here of the techniques we adopt.

For the tasks of inferring lexical or relational semantics, researchers have by and large abandoned statistical models in favor of neural networks, where previously complex or manual laborious tasks (morphological segmentation, part-of-speech tagging, etc.) can be inferred directly using unsupervised methods. A popular technique used for these tasks is to construct word(sentence) embeddings from a neural network (word2vec, doc2vec, fasttext) [154, 155, 156, 120, 26], which encode the semantics of similar words(sentences) by a distributed representation of continuous vectors. The architecture of word2vec's skip-gram model is illustrated in Fig. 8.3a; given an input word, the network is optimized for the classification of words within a windowed context. Figure 8.3b illustrates a classic example of the semantic relationships that can be modeled using this approach for

```
(S (NP (Art The)
       (N linguist))
   (VP (V saw)
       (NP (N biologists))
       (PP (Prep with)
           (NP (Art a)
               (N microscope))))
)
```

(a)

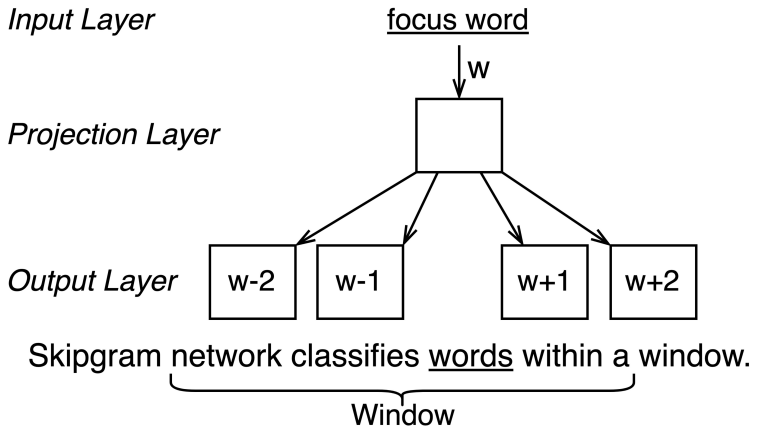


(b)

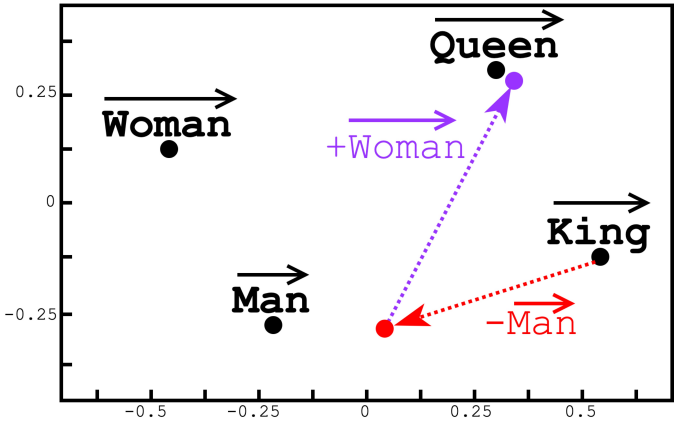
S	::=	NP VP
NP	::=	N
	|	Art N
VP	::=	V NP PP
PP	::=	Prep NP
N	::=	linguist
	|	biologists
	|	microscope
V	::=	saw
Prep	::=	with
Art	::=	The
	|	a

(c)

Figure 8.2: A treebank consists of a large collection of annotated sentences. Entries can be marked with e.g., lexical, morphological, and/or semantic tags. Left, example of a typical treebank entry representation for an ambiguous sentence. Its phrase structure parse tree (center)—which reveals that the biologists in question are laughably tiny—corresponds to an underlying grammar (right).



(a)



(b)

Original Sequence

$^{(1)}\overrightarrow{A}\ ^{(2)}\overrightarrow{B}\ ^{(3)}\overrightarrow{C}DEFGHIJK..$

Splittings

$$\begin{cases} 1)\ ABC, DEF, GHI, .. \\ 2)\ BCD, EFG, HIJ, .. \\ 3)\ CDE, FGH, IJK, .. \end{cases}$$

(c)

Figure 8.3: Distributed representations of words (word vectors) provide straightforward semantic similarity analysis. (a): A skip-gram word2vec model trains a small, fully connected network to learn words in the context of an input focus word. (b): A classic result of the word2vec approach is that linear regularities between words are maintained in the vector space [156]. (c): The ProtVec approach splits each discrete primary sequence into 3 sequences from overlapping 3-grams prior to training the word2vec-inspired model [11].

natural languages: linear regularity of semantics capturing both royalty and gender of the underlying representations [156]. Word vectors exist as points in the n-dimensional space determined by the trained weights in the hidden layer of the network; in general, semantically similar words appear near each other in the n-dimensional space, leading to a natural utilization of partitioning methods to discover e.g., semantic clusters [94, 234] and protein family classification [11, 171].

Refs. [11, 113, 173] take inspiration from the word2vec and related algorithms for biological sequences, showing that the underlying physicochemical semantics can also be characterized for biological sequence data. Ref. [11] defines a method to split peptide sequences into smaller biological "sentences" by splitting each sequence into 3 sequences of overlapping biological "words" of length three (3-mers) prior to training a skip-gram network (Fig. 8.3c). We combine Ref. [11]'s approach to training word vectors for peptide sequences with an unsupervised clustering method to generate a treebank of interest to induce a PCFG for the *PepGen* approach in *PepSyn* (Section 10.1.2). Additional related work is discussed in Chapter 12.

Chapter 9

Overview

MediSyn is a software framework for the safe discovery and synthesis of *de novo* pharmaceuticals. As shown in Figure 9.1, *MediSyn* provides a collection of interacting modules for general-purpose use. *PepSyn*, a peptide-specific implementation within the *MediSyn* framework, is presented in Chapter 10.

9.1 Modules

Front-end As the entry-point to the system, *MediSyn*'s `front-end` module determines a user interface that creates a candidate domain. A front-end module incorporates domain-level knowledge regarding e.g., the specific class of drug a user wants to discover, its structure, etc. While we provide a PCFG representation of a candidate domain (see Section 9.2), different models can be specified.

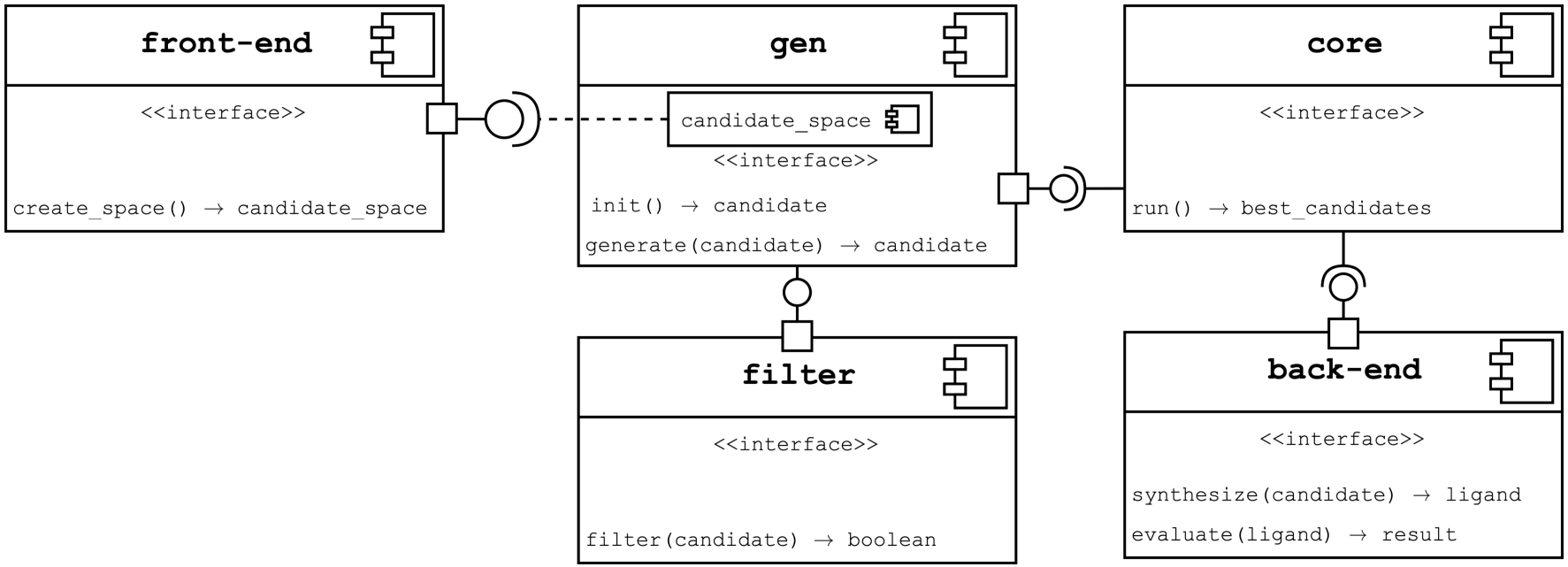


Figure 9.1: Simplified diagram of component structures and basic interfaces of *MediSyn*.

Gen Although exact search methods may vary, a common sub-task in optimizing search problems is *generating candidates*, and various methods are applicable to multiple search techniques. The *gen* module's `init` and *generate* methods accomplish this by clearly defining the process for generating candidates from the candidate space, and works together with optional filter module(s) to provide preferred candidates to the *core*. The gen module has a strict dependency on the front-end module – namely, the candidate space that the front-end generates is the underlying domain the gen module uses for candidate generation. The `init` method defines how to sample directly from the underlying candidate space, while the `generate` method defines how to jump from one candidate to another in the candidate space. The search method defined by the *core* (discussed next) may utilize one or both of these methods; e.g., a Frequentist-inspired search may randomly sample independent and identically distributed candidates utilizing the `init` method alone, whereas a Bayesian approach will define a jumping distribution using the `generate` method.

Core Domain knowledge, intuition, or a stochastic choice may lead a user to utilize a particular optimization strategy, be it domain-tailored or a black-box approach. The *core* module provides a user the flexibility to determine the search method employed while being agnostic to the underlying candidate domain. The interface for a core module is intentionally sparse: a user must define the `run` method, and has access to candidate generation from the gen module and synthesis/evaluation routines from the back-end module. By separating the generation process from the search method, users can incrementally modify the search process to experiment with different, related techniques. For example, the provided Metropolis-Hastings MCMC approach discussed in Section 9.2 utilizes the jumping distribution defined by the gen module's `generate` method, and

it is well known that the choice of jumping distribution can significantly impact an MCMC algorithm's convergence [73, 192]. Knowing this, users can utilize different gen modules to implement different MCMC techniques; for example, including the concept of momentum to the sampling technique, the jumping distribution can be modified to resemble Hybrid Monte Carlo [56]. Alternatively, new search methods can utilize existing candidate generation techniques where appropriate; e.g., in implementing or utilizing a black-box evolutionary algorithm, users can reuse the candidate representation and generation process to define mutation and/or crossover operations.

Filter The *filter* module allows the search to optionally reject syntactically legal candidates from being synthesized and evaluated based on criteria specified by the user. For example, the user may want to avoid synthesizing candidates with known or presumed cytotoxic or hemolytic properties (e.g., see Section 10.2 and Fig. 10.1); likewise, the user may want to filter candidates having a high likelihood of failure during synthesis to reduce the likelihood of wasting costly chemical reagents [159]. The only requirement is that the criteria for filtering be computational and decidable. *MediSyn*'s filter module interface cooperates with a gen module in order to screen these undesirable candidates prior to synthesis and evaluation.

Back-end Irrespective of the search method employed by the `core`, different problems may require different evaluation methods. For example, *chemically* synthesizing and evaluating candidates would require a human-in-the-loop or cyber-physical interface in which computation (candidate proposals, optimization strategy) interacts directly with a human-driven or automated-procedure to synthesize and characterize each candidate. Moreover, different classes of molecular structure or pathogenic targets are likely to

require different synthesis and evaluation approaches. On the other hand, users desiring *in silico* approximation of their objectives (as in Chapter 10) can forego chemical synthesis, while benefiting from the separation of the search method from the objective function and candidate evaluation methods. To abstract these relationships between searching and evaluation, *MediSyn*'s `back-end` module implements the `synthesize` and `evaluate` methods. The `synthesize` method returns an identifier to a synthesized candidate upon successful synthesis,[1] while `evaluate` returns a continuous value in $\mathcal{R}^+$. Together with the `core` module, a `back-end` completes a feedback loop that allows ground truth evaluations of candidates to inform the search process.

9.2 Generalized Core, Gen, and Back-end Modules

MediSyn provides a default candidate space representation of a PCFG (Section 8.3), allowing for generalized reuse in varied domains, as well as `core` and `gen` modules implementing a Markov Chain Monte Carlo (MCMC) superoptimization technique taking inspiration from [196, 71]. To ensure a relatively good starting state, we assume that the PCFG represents a prior probability distribution of drugs of interest where strings in the given language having higher likelihood occur in local minima in the posterior distribution; by weighting initial samples from the PCFG, we obviate the need for significant burn-in. To sample from the PCFG, we perform a leftmost derivation, where for each nonterminal $\mathcal{N}^i$ encountered, we randomly choose the rule to apply relative to its probability mapping in $\mathcal{P}$, s.t. $\mathcal{N}^i$ yields ρ^j with probability $\mathcal{P}(\mathcal{N}^i \rightarrow \rho^j)$. We denote such a yield from $\mathcal{N}^i$ to ρ^j as $\mathcal{N}^i \implies \rho^j$. A complete derivation of sample s from S is denoted $S \overset{*}{\implies} s$.

[1] A reserved error code is returned upon failure; which directs the `core` to ignore the candidate.

To initialize, `gen.init` performs weighted random sampling without replacement for a given number n of initial samples $\overrightarrow{X_t}$ by deriving n strings from the PCFG, where $\overrightarrow{X_t} = \{s \in \Sigma* : S \overset{*}{\implies} s\}$; note that, $\forall s \in \overrightarrow{X_t}$, $P(s) > 0$, as we are sampling directly from the PCFG.

`gen.generate` selects proposal candidates $\overrightarrow{X}$ from $Q(\overrightarrow{X}|\overrightarrow{Y})$, where Q is a *symmetric* jumping distribution that suggests $\overrightarrow{X}$ given the previous sample $\overrightarrow{Y}$. To generate a new candidate $x \in X$, we can randomly select a node $\hat{P} = P \implies y_p$ from the derivation tree $S \overset{*}{\implies} y \in \overrightarrow{Y}$, and recompute a leftmost derivation from this node, inserting the newly-parsed sub-tree in place of $\hat{P}$.

As mentioned in Section 8.2, we take inspiration from [71]'s simulated annealing adaptation to MCMC to encourage exploration of areas of interest, but rather than applying the annealing principle to the acceptance of candidates, our adaptation dynamically adjusts the way we select node $\hat{P}$ from the derivation tree. The chosen height of $\hat{P}$ is directly correlated with how much $\overrightarrow{X}$ differs from $\overrightarrow{Y}$. As $\hat{P}$ is selected randomly, we can perform weighted sampling that favors nodes closer to the leaves of the expression in proportion to the tempering variable.

Given a proposal candidate $x \in X$—where x has never been seen—the core calls the `synthesize` and `evaluate` methods from the back-end to retrieve its efficacy; in the case where the proposed candidate has been seen before, a stored result is returned; for simplicity, we denote this procedure as $f(x)$.

The technique in Algorithm 2 implements *MediSyn*'s provided MCMC `core` module. After initializing *score* and $\overrightarrow{B}$, variables used to keep track of the optimal set of candidates,

Algorithm 2 Population MCMC

1: **procedure** CORE.RUN(*thresh*, n, G, *early*)
2: $t \leftarrow 0$
3: $score \leftarrow 0$
4: $\overrightarrow{B} \leftarrow \emptyset$
5: $\overrightarrow{X_t} \leftarrow$ `gen.init()`
6: **while** True **do**
7: $\overrightarrow{X'} \leftarrow$ `gen.generate(`$\overrightarrow{X_t}$`)`
8: $\overrightarrow{A} \leftarrow \{\alpha_i : \alpha_i = \texttt{f}(x'_i)\,/\,\texttt{f}(x'_{t,i})\} \; \forall x'_i \in \overrightarrow{X'}, \forall x_{t,i} \in \overrightarrow{X_t}$
9: $(score, \overrightarrow{B}) \leftarrow$ `update_best()`
10: $u \leftarrow$ `Uniform(`$0, 1$`)`
11: $\overrightarrow{X_{t+1}} \leftarrow \begin{cases} x'_i, & u \leq \alpha_i \\ x_t, & u > \alpha_i \end{cases} \; \forall x'_i \in \overrightarrow{X'}, \forall x_{t,i} \in \overrightarrow{X_t}, \forall \alpha_i \in \overrightarrow{A}$
12: $t \leftarrow t + 1$
13: **if** $((score \leq thresh$ or $t > G) \wedge early) \vee (score \leq thresh \wedge t > G)$ **then**
14: *break*
return $(score, \overrightarrow{B})$

along with sampling a prior ($\overrightarrow{X_t}$) as the starting point for the search, the `core` interfaces with the `gen` and `back-end` modules for generating, synthesizing, and evaluating proposal candidates. The `update_best` routine keeps track of the best score and optimal set of candidates. We continue searching for candidates until we have found candidate(s) with a minimum desired fitness (*thresh*) and/or have run a minimum number of iterations (G), where either condition will terminate the search if the supplied *early* exit variable evaluates true.

Chapter 10

PepSyn

Here, we present *PepSyn* as a set of modules that collectively form a proof-of-concept implementation of *MediSyn* using approaches from program synthesis for discovering peptide pharmaceuticals. Figure 10.1 depicts a high-level overview of *PepSyn*'s operation: the structure of prototypical drugs allows for generation of syntactically legal candidates (Sections 10.1.1 and 10.1.2); a filtering step prunes undesirable or otherwise impractical candidates prior to evaluation (Section 10.2); and, once obtained, the evaluation metric is fed back into the generation step to drive the stochastic generation of the next candidates (Section 10.3).

As a demonstration of *MediSyn*'s modularity, *PepSyn* implements two `front-end` modules that generate a candidate space as a PCFG; these modules aim to be expressive and palatable to practitioners who specialize in the chemical or biological sciences, while varying in both complexity and scope. The first (Section 10.1.1) implements a simple domain-specific language we call *PepSketch* for specifying the structure of peptide

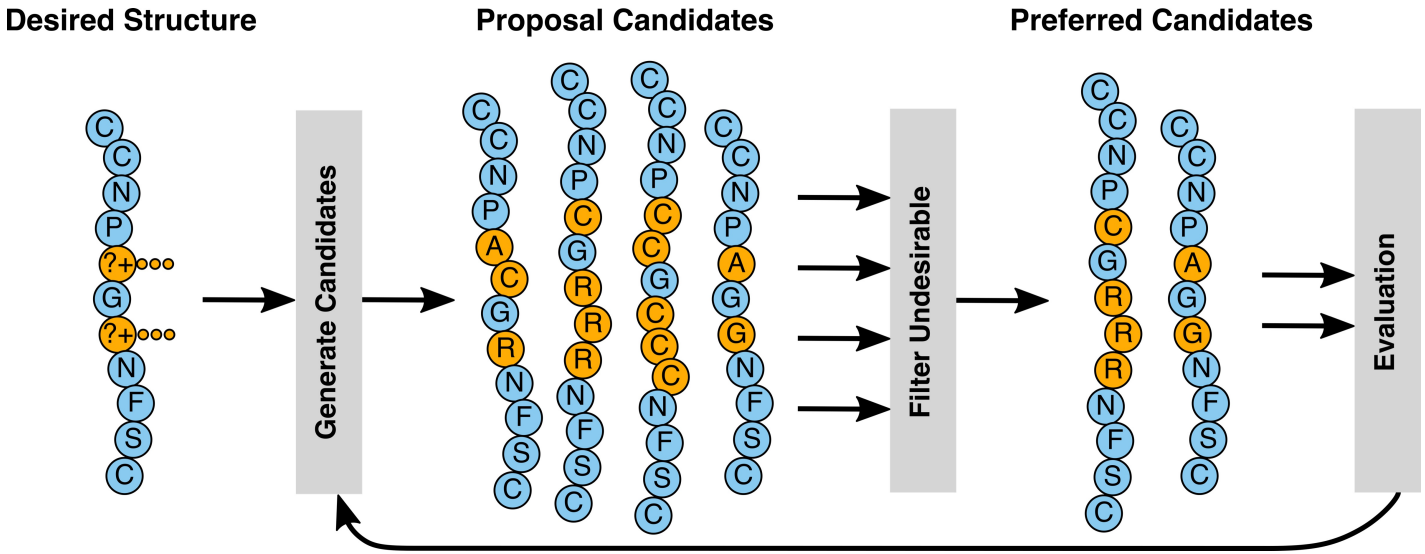


Figure 10.1: *PepSyn* implements a stochastic drug discovery process for peptide drugs utilizing the *MediSyn* framework. A desired peptide drug's structure is modeled as a PCFG using an appropriate front-end; candidate structures are generated using *MediSyn*'s provided gen module to sample from the PCFG; a filter module removes proposed candidates that are inferred as toxic or unlikely to be possible to (chemically) synthesize, and MIC evaluation of the unfiltered (preferred) candidates is computed to progress *MediSyn*'s provided implementation of an optimizing MCMC search.

PE ::=			*PepSketch* Expression:
	LE		Ligand Expression
|	***LE*** `where` ***CB***		Category Binding(s)
LE ::=			Ligand Expression:
	σ	$\sigma \in \Sigma$	Amino Acid Residue
|	***LE*** ***LE***		Concatenation
|	***WS***		Wildcard Symbol
|	`(`***OE***`)?`		Optional Expression
|	`(`***LE*** `|` ***LE***`)`		Alternation
|	`[`***RE***`]`		Repeat Expression
OE ::=			Optional Expression:
	LE		Ligand Expression
RE ::=			Repeat Expression:
	LE `*`		Kleene-star
|	***LE*** `+`		Kleene-plus
|	***LE*** i	$i \in \mathbb{Z}$	Repeat Number
CB ::=			Category Binding(s):
	CB ***CB***		Concatenation
|	***WS*** `in` σ	$\sigma \subseteq \Sigma$	Wildcard Binding
WS ::=	$any \notin \Sigma \cup \Sigma_{res}$		Wildcard Symbol

Figure 10.2: *PepSketch*'s syntax.

sequences for any particular functional objective (e.g., antimicrobial, antiviral, anticancer, etc.); the second (*PepGen*, Section 10.1.2) demonstrates an approach in the programming by example (PBE) paradigm, whereby a selection of example antimicrobial peptides are directly used to synthesize a candidate specification that the system optimizes over.

The subsequent subsections discuss their implementation and how they each converge to a common representation of a PCFG to specify the space of candidates we wish to optimize. We utilize the MCMC implementation in *MediSyn*'s provided `core` and `gen` modules (Section 9.2), while the `filter` (Section 10.2) and `back-end` (Section 10.3) modules are peptide-specific.

```
1  RGG(G|R)LCYCR##%C%CV(GR)? where
2      % in hydrophobic amino acids
```

Figure 10.3: A *PepSketch* which specifies the protegrin family of antimicrobial peptides

10.1 *PepSyn* Front-ends

10.1.1 *PepSketch* Front-end

PepSketch is a regex-like domain-specific-language that allows a user to succinctly express a candidate space for peptide ligands as a template, using a sequence of known and unknown amino acid residues and optional bindings of wildcards to particular sub-alphabets in its language (Fig. 10.2). As shown in Table 8.1, the 21 amino acids used to construct peptides are classified into groups based on various physicochemical properties; we define *PepSketch*'s alphabet Σ to be these 21 unique amino acid characters along with an empty string λ. Sub-alphabets are used to specify common subsets of amino acids:

$$\Sigma = \Sigma_\zeta \cup \Sigma_\Phi \cup \Sigma_O \cup \lambda, \tag{10.1}$$

where $\Sigma_\zeta, \Sigma_\Phi$, and Σ_O are the sets of hydrophilic, hydrophobic, and all "other" amino acids, respectively; further refinements (e.g., $\Sigma_{\zeta[+]}$ for positively charged hydrophilic amino acids) are defined for fine-tuned binding expressions. A set of reserved symbols $\Sigma_{res} = \{ \texttt{(}, \texttt{)}, \texttt{|}, \texttt{[}, \texttt{]}, \star, +, ?, i \in \mathbb{Z}\}$ are restricted from wildcard specification.

A user of *PepSketch* is assumed to possess an intimate understanding of a target's physicochemical properties, with a high level of confidence that a viable candidate will have specific amino acids, groups, or particular properties of residues in the sequence at specific locations, but is unsure of which amino acid(s) should occur at *every* location in the peptide. The *PepSketch* depicted in Fig. 10.3, which specifies a subset of the protein family of antimicrobial peptides [163], illustrates this idea: concrete residues occur inline

with wildcards representing a single amino acid substitution (i.e., non-repeating), where the % wildcard is bound to the hydrophobic amino acid sub-alphabet (Σ_Φ), and the # wildcard is unbound (i.e., it can resolve to *any* $\sigma \in \Sigma$).

The module implementing the *PepSketch* language transforms a *PepSketch* expression into a PCFG with probabilities initialized uniformly across productions. For example, the protegrin *PepSketch* in Fig. 10.3 induces the PCFG $G = (\{PE, G\_R, \#, \%, ?\}, \Sigma, PE, \mathcal{R}_\rho, \mathcal{P}_\rho)$, where $\mathcal{R}_\rho$ is:

$$\texttt{PE} \rightarrow \text{RGG}\texttt{G\_R}\text{LCYCR}\texttt{\#\#\%}\text{C}\texttt{\%}\text{CV}\texttt{?}$$

$$\texttt{G\_R} \rightarrow \text{G}$$

$$\texttt{G\_R} \rightarrow \text{R}$$

$$\texttt{\#} \rightarrow \sigma \in \Sigma$$

$$\texttt{\%} \rightarrow \sigma_\phi \in \Sigma_\Phi$$

$$\texttt{?} \rightarrow \text{GR}$$

$$\texttt{?} \rightarrow \lambda$$

and $\mathcal{P}_\rho$ uniformly maps probabilities for each production based on the number of productions associated with each nonterminal; e.g., $\mathcal{P}_\rho(\texttt{G\_R} \rightarrow \text{G}) = 0.5$ (two production rules associated with `G_R`), and $\mathcal{P}_\rho(\texttt{\%} \rightarrow x) = 0.125\ \forall x \in \Sigma_\Phi$ (eight production rules associated with `%`).

Fig. 10.4 presents a few of the formal transformation rules. The transformation judgment for a ***ligand*** is ***ligand*** $\triangleright \mathcal{N}, \mathcal{R}, \mathcal{P}$ where $\mathcal{N}$ is the nonterminal resulting from the transformation of ***ligand***, $\mathcal{R}$ is the set of generated production rules, and $\mathcal{P}$ is the probability distribution for the rules. The rule CONCAT for concatenation transforms the

$$\textsc{Concat}\quad \frac{\boldsymbol{ligand}_1 \triangleright \mathcal{N}_1, \mathcal{R}_1, \mathcal{P}_1 \qquad \boldsymbol{ligand}_2 \triangleright \mathcal{N}_2, \mathcal{R}_2, \mathcal{P}_2 \qquad \mathcal{N} \text{ fresh in } \mathcal{R}_1, \mathcal{R}_2}{\boldsymbol{ligand}_1\ \boldsymbol{ligand}_2 \triangleright \mathcal{N}, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{\mathcal{N} \to \mathcal{N}_1 \cdot \mathcal{N}_2\}, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{(\mathcal{N} \to \mathcal{N}_1 \cdot \mathcal{N}_2) \mapsto 1\}}$$

$$\textsc{Alt}\quad \frac{\boldsymbol{ligand}_1 \triangleright \mathcal{N}_1, \mathcal{R}_1, \mathcal{P}_1 \qquad \boldsymbol{ligand}_2 \triangleright \mathcal{N}_2, \mathcal{R}_2, \mathcal{P}_2 \qquad \mathcal{N} \text{ fresh in } \mathcal{R}_1, \mathcal{R}_2}{\boldsymbol{ligand}_1 \mid \boldsymbol{ligand}_2 \triangleright \mathcal{N}, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{\mathcal{N} \to \mathcal{N}_1 \mid \mathcal{N} \to \mathcal{N}_2\}, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{(\mathcal{N} \to \mathcal{N}_1) \mapsto 0.5 \mid (\mathcal{N} \to \mathcal{N}_2) \mapsto 0.5\}}$$

$$\textsc{Wild}\quad \boldsymbol{wildcard} \triangleright \mathcal{N}_{wildcard}, \emptyset, \emptyset$$

$$\textsc{WildBind}\quad \boldsymbol{wildcard}\ \texttt{in}\ \sigma \triangleright \{\overline{\mathcal{N}_{wildcard} \to \sigma_i}\}, \{\overline{(\mathcal{N}_{wildcard} \to \sigma_i) \mapsto 1/|\sigma|}\}$$

$$\textsc{Bind}\quad \frac{\boldsymbol{ligand} \triangleright \mathcal{N}, \mathcal{R}_1, \mathcal{P}_1 \qquad \boldsymbol{binding} \triangleright \mathcal{R}_2, \mathcal{P}_2}{\boldsymbol{ligand}\ \texttt{where}\ \boldsymbol{binding} \triangleright \mathcal{N}, \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{P}_1 \cup \mathcal{P}_2}$$

Figure 10.4: *PepSketch* Transformation Rules

two ***ligand***s separately to obtain a nonterminal, set of productions and a probability distribution for each. It then generates a fresh nonterminal and a production rule that maps it to concatenation of the two nonterminals, and maps this single rule to the probability one. The rule ALT, for alternation, is similar except that it generates two rules, one for each of the two alternatives, and maps each to probability 0.5. Each ***wildcard*** is unique; we use the nonterminal $\mathcal{N}_{\textbf{\textit{wildcard}}}$ to represent a wildcard, as the rule WILD shows.

Therefore, the transformation judgment for a binding is $\textbf{\textit{binding}} \triangleright \mathcal{R}, \mathcal{P}$ where the resulting nonterminal is implicitly $\mathcal{N}_{\textbf{\textit{wildcard}}}$. For a binding ***wildcard*** `in` σ, the rule WILDBIND generates a rule for each member of σ, and distributes the probability equally between them. The rule BIND transforms the binding and the ligand, and then aggregates the resulting rules and distributions.

10.1.2 *PepGen* Front-end

Whereas *PepSketch*'s intended user base consists of scientists who have intimate insight leading to a concise bounding on the candidate specification, others with less domain expertise may wish to discover a new drug based on an aggregate knowledge of relevant properties of existing drugs. To meet this need *PepGen* (Fig. 10.5) implements a text-based user interface in the programming by example paradigm that infers a PCFG from a corpus of peptides $\mathcal{D}$, which the user selects from relations that we define on a public-facing repository containing experimentally validated data on antimicrobial activity in peptides.[1] Each example peptide drug $d_i \in \mathcal{D}$ is a pair $(\mathbf{s_i}, x_i)$, where $\mathbf{s_i}$ is d_i's amino acid string and $x_i \in \mathbb{R}$ is its MIC value. *PepGen* constructs a PCFG that

[1] Data gathered from the Database of Antimicrobial Activity and Structure of Peptides (DBAASP) [183].

generalizes the structural properties and relative antimicrobial activity of sequences present in $\mathcal{D}$. To do so, *PepGen* generates a treebank $\mathcal{T}_\mathcal{D}$ from $\mathcal{D}$. Recall that a treebank is a set of parse trees (see Section 8.3 and Fig. 8.2a) whose unique derivations form all the productions of its corresponding CFG; derivations that occur in multiple parse trees within the treebank do not generate redundant productions, but instead correlate to the likelihood that a given parse will occur; our objective is to increase the likelihood of parses that lead to superior antimicrobial activity relative to other parses. For each drug $d_i \in \mathcal{D}$, *PepGen* generates a set of parse trees, τ_{d_i}; $\mathcal{T}_\mathcal{D}$ is defined as their union:

$$\mathcal{T}_\mathcal{D} = \bigcup_{d_i \in \mathcal{D}} \tau_{d_i} \tag{10.2}$$

The following discussion describes how to generate τ_{d_i} for a single peptide $d_i \in \mathcal{D}$

10.1.2.1 Augmentation

We first *augment* $\mathbf{s_i}$ into a set of semantically similar sequences, such that the number of augmented sequences is proportional to d_i's antimicrobial activity relative to all the peptides in $\mathcal{D}$. These augmented sequences encode learned physicochemical properties of $\mathbf{s_i}$ using a word embedding space Δ, which is trained offline and encompasses our entire database of antimicrobial peptides. Our approach is inspired by `ProtVec` [11], which decomposes amino acid sequences into 3-mers (3-character subsequences; see Section 8.3) and learns their semantic similarity as the proximity of points in a 100-dimensional space. We train our model exclusively using peptides with experimentally validated antimicrobial effects, with the goal of discovering semantic relationships between 3-mers with antimicrobial function. Our training dataset consists of 14,271 primary sequences of antimicrobial peptides ranging from 3 to 120 amino acids in length. As shown earlier in Fig. 8.3c, we expand this dataset by splitting each peptide into 3 sets of 3-mers

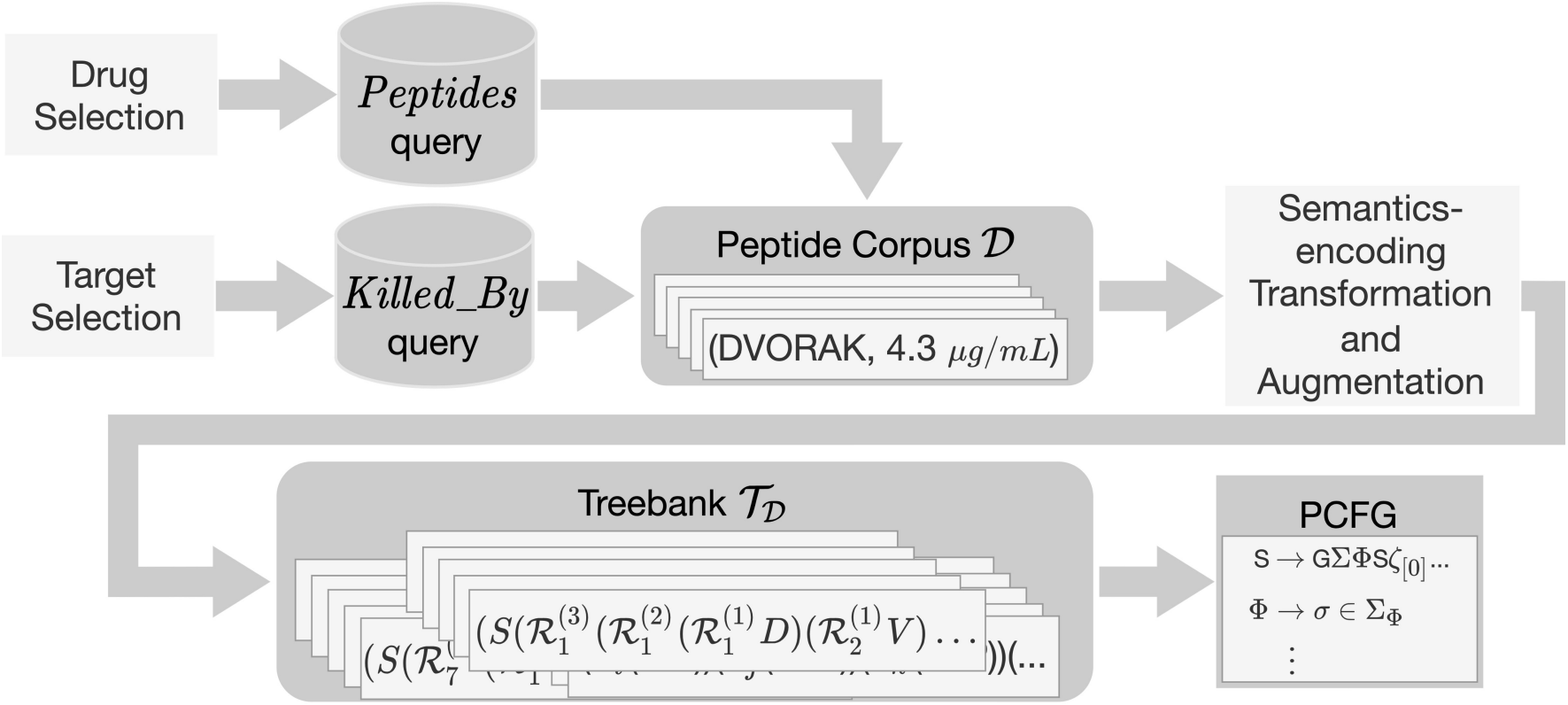


Figure 10.5: *PepGen*: A selection of antimicrobial peptides or a group of targets with experimentally validated susceptibility to a set of antimicrobial peptides are used to form an appropriate corpus to induce a candidate space represented as a PCFG.

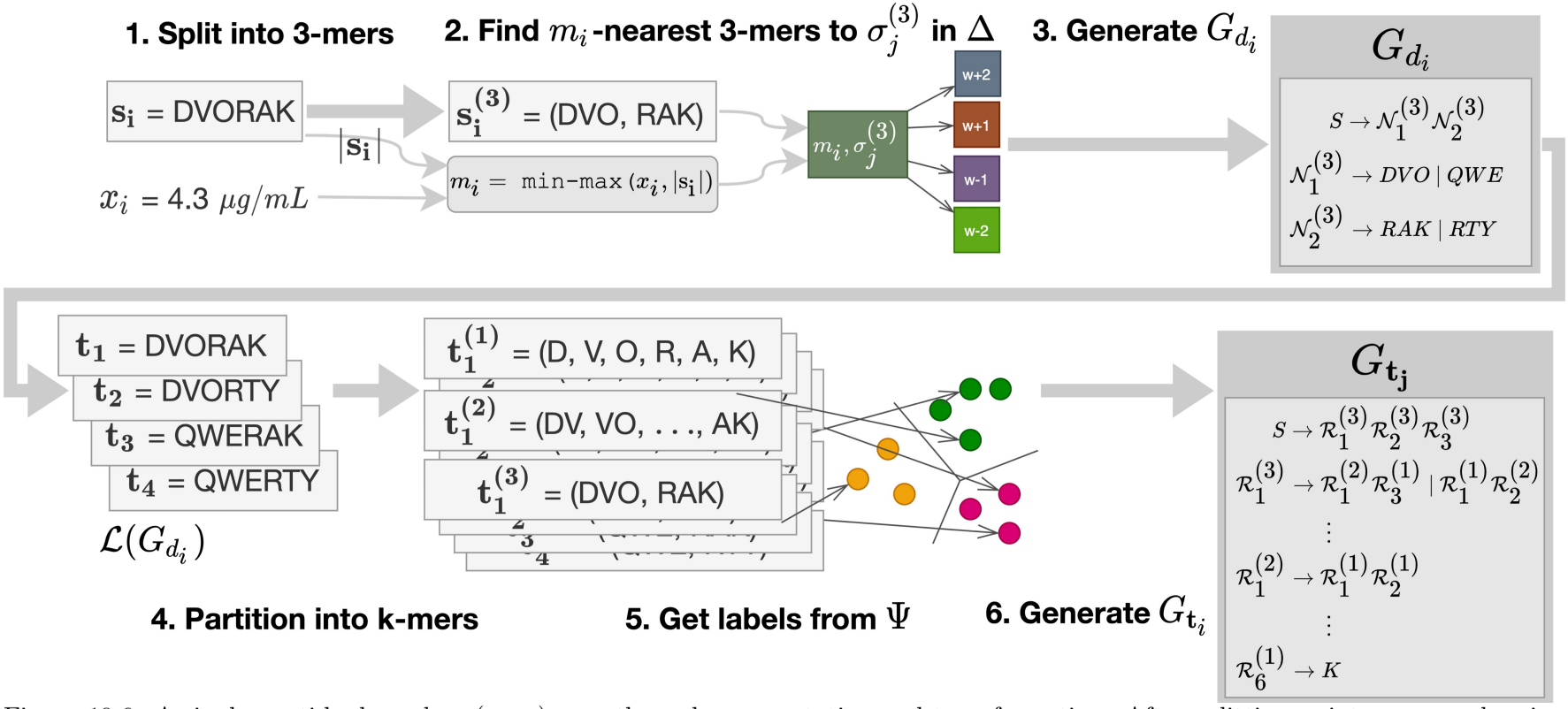


Figure 10.6: A single peptide drug $d_i = (\mathbf{s_i}, x_i)$ goes through augmentation and transformation. After splitting $\mathbf{s_i}$ into non-overlapping 3-mers, we generate a grammar G_{d_i} by associating 3-mers from $\mathbf{s_i}$ with their m_i-nearest 3-mers in a word embedding model. The bolstered corpus $\mathbf{t_j} = \mathcal{L}(G_{d_i})$ is then transformed into a separate grammar $G_{\mathbf{t_j}}$ by associating k-mers from each $\mathbf{t_j}$ with labels found in a clustering of semantically related k-mers. Parse trees for each $\mathbf{t_j}$ are generated using $G_{\mathbf{t_j}}$, providing a treebank for PCFG inference [13, 147].

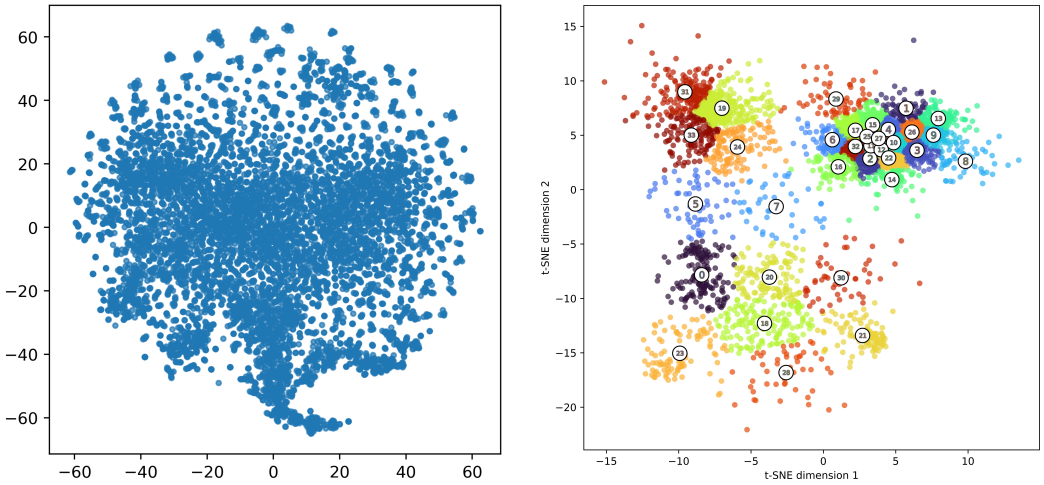


Figure 10.7: Left: 100-dimensional points from a word embedding model are projected onto a 2D space using parametric t-SNE, which embeds a high-dimensional dataset in a latent space of lower dimension in a way that maintains local pairwise distances between data points [232]. Right: Another word embedding space is similarly projected to 2D-dimensions; a partitioning method is used to determine semantic clusters for generating a grammar from a set of unlabeled peptides.

prior to training, yielding 8,526 unique 3-mers appearing in 42,813 distinct sequences. The resulting 100-dimensional embedding space, projected on 2-dimensions in Fig. 10.7, captures the semantic similarity of 3-mers that occur in close proximity in our model.

Let $\mathbf{s_i} = \sigma_1\sigma_2\ldots\sigma_{n_i}$. If the length n_i of $\mathbf{s_i}$ is not divisible by 3, we append one or two 'X' characters, which represent amino acids that do not alter its function [22]; this ensures that $\mathbf{x_i}$ can be fully decomposed into a sequence of non-overlapping 3-mers $\mathbf{s_i^{(3)}} = (\sigma_{\mathbf{1}}^{(\mathbf{3})}, \sigma_{\mathbf{2}}^{(\mathbf{3})}, \ldots, \sigma_{\mathbf{n_i/3}}^{(\mathbf{3})})$, where $\sigma_{\mathbf{1}}^{(\mathbf{3})} = \sigma_1\sigma_2\sigma_3$, $\sigma_{\mathbf{2}}^{(\mathbf{3})} = \sigma_4\sigma_5\sigma_6$, ..., $\sigma_{\mathbf{n_i/3}}^{(\mathbf{3})} = \sigma_{n_i-2}\sigma_{n_i-1}\sigma_{n_i}$.

For a positive integer m_i, which is uniquely computed for d_i (details below), let $\delta(m_i, \sigma_{\mathbf{j}}^{(\mathbf{3})})$ be the set of m_i-nearest 3-mers to $\sigma_{\mathbf{j}}^{(\mathbf{3})}$ in Δ; by convention, we assume that $\sigma_{\mathbf{j}}^{(\mathbf{3})} \in \delta(m_i, \sigma_{\mathbf{j}}^{(\mathbf{3})})$, but there is no guarantee that any of other 3-mers in $\delta(m_i, \sigma_{\mathbf{j}}^{(\mathbf{3})})$ will also be constituent 3-mers of $\mathbf{s_i^{(3)}}$. We denote an arbitrarily selected 3-mer from $\delta(m_i, \sigma_{\mathbf{j}}^{(\mathbf{3})})$ as $\sigma_{\mathbf{k}}^{(\mathbf{3})} = \sigma_{k_1}\sigma_{k_2}\sigma_{k_3}$, which has no ordinal relation to the 3-mers in $\mathbf{s_i^{(3)}}$ or amino acids in $\mathbf{s_i}$.

An MCMC search could perturb $\mathbf{s_i}$ by replacing a constituent 3-mer $\sigma_{\mathbf{j}}^{(\mathbf{3})} \in \mathbf{s_i^{(3)}}$ with a randomly selected 3-mer $\sigma_{\mathbf{k}}^{(\mathbf{3})} \in \delta(m_i, \sigma_{\mathbf{j}}^{(\mathbf{3})})$. This perturbation would minimally impact the biological function of the resulting peptide due to the close proximity of $\sigma_{\mathbf{j}}^{(\mathbf{3})}$ to $\sigma_{\mathbf{k}}^{(\mathbf{3})}$ in the embedding space Δ; at the same time, the perturbation could improve antimicrobial activity, i.e., if the resulting peptide had a lower MIC.

We define the augmentation of d_i as a context free grammar $G_{d_i}(\mathcal{N}_{d_i}, \Sigma, S, \mathcal{R}_{d_i})$. We associate a non-terminal $\mathcal{N}_j^{(3)}$ with each 3-mer $\sigma_{\mathbf{j}}^{(\mathbf{3})} \in \mathbf{s_i^{(3)}}$. Then the set of non-terminals is

$$\mathcal{N}_{d_i} = S \cup \bigcup_{j=1}^{n_i/3} \mathcal{N}_j^{(3)}. \tag{10.3}$$

We define a top-level production $\mathcal{R}_0^{(3)} = S \rightarrow \mathcal{N}_1^{(3)}\mathcal{N}_2^{(3)} \ldots \mathcal{N}_{n_i/3}^{(3)}$, and a set of productions $\mathcal{R}_j^{(3)}$ with each 3-mer $\sigma_{\mathbf{j}}^{(\mathbf{3})} \in \mathbf{x_i^{(3)}}$:

$$\mathcal{R}_j^{(3)} = \bigcup_{\sigma_{\mathbf{k}}^{(\mathbf{3})} \in \delta(m_i, \sigma_{\mathbf{j}}^{(\mathbf{3})})} \mathcal{N}_j^{(3)} \rightarrow \sigma_{k_1}\sigma_{k_2}\sigma_{k_3} \tag{10.4}$$

Then the set of productions is:

$$\mathcal{R}_{d_i} = \bigcup_{j=0}^{n_i/3} \mathcal{R}_j^{(3)} \tag{10.5}$$

We enumerate all the strings in G_{d_i}'s language $\mathcal{L}(G_{d_i})$, yielding the desired set of augmented sequences.

Our description of *augmentation* necessitates one final detail: how to empirically choose an appropriate value of m_i; we make this decision using a min-max scaling function [102], adjusted for the length of each $d_i \in \mathcal{D}$. Let $A = [a_{min}, a_{max}]$ be a bound range of scaled MIC activity determined empirically from $\mathcal{D}$ as follows:

$$a_{min} = \min_{(\mathbf{s_i}, x_i) \in \mathcal{D}} \left\{ \frac{1}{x_i} \right\} \tag{10.6}$$

$$a_{max} = \max_{(\mathbf{s_i}, x_i) \in \mathcal{D}} \left\{ \frac{1}{x_i} \right\} \tag{10.7}$$

We also define a target integer range $B = [b_{min}, b_{max}]$ to bound the number of augmented sequences each $d_i \in \mathcal{D}$ will produce. Our procedure for determining m_i guarantees that $b_{min} \leq (m_i)^{n_i/3} \leq b_{max}$. The lower bound, $b_{min} \geq 2$, is a constant value provided by the user.[2] Let $\mu_{\mathcal{D}}$ and $\sigma_{\mathcal{D}}$ be the arithmetic mean and standard deviation[3] of the lengths of the amino acid sequences in $\mathcal{D}$. Then

$$b_{max} = 10^{\left\lceil \frac{1}{3}(\mu_{\mathcal{D}} + 1.5\sigma_{\mathcal{D}}) \log(b_{min}) \right\rceil}, \tag{10.8}$$

For a given drug $d_i = (\mathbf{s_i}, x_i) \in \mathcal{D}$, with $n_i = |\mathbf{s_i}|$ the value of m is then computed by scaling function:

$$m_i = \left\lfloor \sqrt[n_i/3]{b_{min} + \left(\frac{1}{x_i} - a_{min}\right) \frac{b_{max} - b_{min}}{a_{max} - a_{min}}} \right\rfloor. \tag{10.9}$$

10.1.2.2 Semantics-encoding Transformation

Augmentation, described in the previous section, transforms a single amino acid sequence $\mathbf{s_i}$ of length $n_i = |\mathbf{s_i}|$ into a language $\mathcal{L}(G_{d_i})$ containing $(m_i)^{n_i/3}$ length-n_i amino acid sequences. We next transform each sequence $\mathbf{t_j} \in \mathcal{L}(G_{d_i})$ into a set of tagged parse trees $\tau_{\mathbf{t_j}}$, and finally form τ_{d_i} as their union:

$$\tau_{d_i} = \bigcup_{\mathbf{t_j} \in \mathcal{L}(G_{d_i})} \tau_{\mathbf{t_j}} \tag{10.10}$$

To do so, we semantically tag subsequences of each amino acid sequence $\mathbf{t_i} \in \mathcal{L}(G_{d_i})$ in a clustering model Ψ that encodes semantic relationships among arbitrary-length subsequences,[4] in contrast to `ProtVec` [11], which exclusively encodes a space of 3-mers (Fig. 8.3c). To this end, we train a secondary model, denoted $\texttt{ProtVec}^{(3)}$, in which we

[2] We use $b_{min} = 2$ for our experiments.

[3] As a minor abuse of notation, we note that we have used σ previously to represent terminal characters in amino acid strings; in this one instance we re-use σ as the standard deviation, which is standard in statistical literature.

[4] Our `ProtVec`-inspired model, $\texttt{ProtVec}^{(k)}$, is detailed in the Appendix E.

split the original peptide string $\mathbf{s_i}$ into 1-, 2-, and 3-mers, yielding an input vocabulary of 8,986 entries appearing in 85,626 unique sequences. After fitting $\texttt{ProtVec}^{(3)}$, we project the 100-dimensional embedded vectors onto 2 dimensions, and partition the space into semantically-related clusters [227] using an efficient k-means algorithm [53]. These cluster labels are used when transforming an augmented corpus of peptide drugs into a treebank of interest for PCFG induction (Section 8.3).

For each amino acid sequence $\mathbf{t_j} = \sigma_1\sigma_2\ldots\sigma_{n_i} \in \mathcal{L}(G_{d_i})$, let $\mathbf{t_j^{(1)}}$, $\mathbf{t_j^{(2)}}$, and $\mathbf{t_j^{(3)}}$ denote the decomposition of $\mathbf{t_j}$ into respective sequences of 1-mers, 2-mers, and 3-mers as follows:

$$\mathbf{t_j^{(1)}} = \left(\sigma_{\mathbf{1}}^{(\mathbf{1})}, \sigma_{\mathbf{2}}^{(\mathbf{1})}, \ldots, \sigma_{\mathbf{n_i}}^{(\mathbf{1})}\right), \sigma_{\mathbf{k}}^{(\mathbf{1})} = \sigma_k, 1 \leq k \leq n_i; \tag{10.11}$$

$$\mathbf{t_j^{(2)}} = \left(\sigma_{\mathbf{1}}^{(\mathbf{2})}, \sigma_{\mathbf{2}}^{(\mathbf{2})}, \ldots, \sigma_{\mathbf{n_i-1}}^{(\mathbf{2})}\right), \sigma_{\mathbf{k}}^{(\mathbf{2})} = \sigma_k\sigma_{k+1}, 1 \leq k \leq n_i - 1; \tag{10.12}$$

and $\mathbf{t_j^{(3)}}$ is defined analogously to $\mathbf{s_i^{(3)}}$ in the preceding subsection. We note that we allow over-lapping of 2-mers in Eq. (10.12) as a prudent introduction of ambiguity in the resulting grammar,[5] and to negate the need for additional padding of 'X's when $|t_j|$ is not divisible by 6.

Let $\psi\left(\sigma_{\mathbf{k}}^{(\ell)}\right)$ be the label of the cluster that Ψ assigns to $\sigma_{\mathbf{k}}^{(\ell)}$ in the $\texttt{ProtVec}^{(3)}$ space. We define the set of all parse trees using a context free grammar $G_{\mathbf{t_j}}(\mathcal{N}_{\mathbf{t_j}}, \Sigma, S, \mathcal{R}_{\mathbf{t_j}})$. We associate a non-terminal with each unique label $\psi\left(\sigma_{\mathbf{k}}^{(\ell)}\right)$. Then the set of non-terminals is:

$$\mathcal{N}_{\mathbf{t_j}} = S \cup \bigcup_{k=1}^{n_i} \psi\left(\sigma_{\mathbf{k}}^{(\mathbf{1})}\right) \cup \bigcup_{k=1}^{n_i-1} \psi\left(\sigma_{\mathbf{k}}^{(\mathbf{2})}\right) \cup \bigcup_{k=1}^{n_i/3} \psi\left(\sigma_{\mathbf{k}}^{(\mathbf{3})}\right) \tag{10.13}$$

[5] Ambiguity works in our favor, as it reflects the possibility to perturb alternative structures in a resulting peptide sequence for greater exploration [54].

We define a top-level production $\mathcal{R}_S = S \rightarrow \mathcal{R}_1^{(3)}\mathcal{R}_2^{(3)}\ldots\mathcal{R}_{n/3}^{(3)}$; we then define productions $\mathcal{R}_k^{(3)}$, $\mathcal{R}_k^{(2)}$, and $\mathcal{R}_k^{(1)}$ as follows:

$$\mathcal{R}_k^{(3)} = \psi\left(\sigma_{\mathbf{k}}^{(\mathbf{3})}\right) \rightarrow \psi\left(\sigma_{\mathbf{3k-2}}^{(\mathbf{2})}\right)\psi\left(\sigma_{\mathbf{3k}}^{(\mathbf{1})}\right) \mid \psi\left(\sigma_{\mathbf{3k-2}}^{(\mathbf{1})}\right)\psi\left(\sigma_{\mathbf{3k-1}}^{(\mathbf{2})}\right) \tag{10.14}$$

$$\mathcal{R}_k^{(2)} = \psi\left(\sigma_{\mathbf{k}}^{(\mathbf{2})}\right) \rightarrow \psi\left(\sigma_{\mathbf{k}}^{(\mathbf{1})}\right)\psi\left(\sigma_{\mathbf{k+1}}^{(\mathbf{1})}\right) \tag{10.15}$$

$$\mathcal{R}_k^{(1)} = \psi\left(\sigma_{\mathbf{k}}^{(\mathbf{1})}\right) \rightarrow \sigma_k \tag{10.16}$$

The complete set of productions is therefore:

$$\mathcal{R}_{\mathbf{t_j}} = \mathcal{R}_S \cup \bigcup_{k=1}^{n_i} \mathcal{R}_k^{(1)} \cup \bigcup_{k=1}^{n_i-1} \mathcal{R}_k^{(2)} \cup \bigcup_{k=1}^{n_i/3} \mathcal{R}_k^{(3)} \tag{10.17}$$

Intuitively, this grammar allows for the perturbation of any peptide $\mathbf{t_j} \in \mathcal{L}(G_{d_i})$ by randomly selecting a 1-, 2-, or 3-mer and replacing it with another 1-, 2-, or 3-mer that is similar in terms of the classification labels in embedding model Ψ; due to the similar relation, the biological function of the peptide is unlikely to radically change, while the possibility exists that the peptide resulting from the perturbation has a lower MIC.

We generate parse trees τ_{t_j} using an Early parser [58]; Eqs. (10.2) and (10.10) then complete the generation of $\mathcal{T}_\mathcal{D}$. Finally, we use the inside-outside algorithm on $\mathcal{T}_\mathcal{D}$'s underlying grammar to generate the PCFG of interest [13, 147].

While the *augmentation* step bolsters the corpus so that semantically similar sequences to superior drugs appear more often, leading to an increased likelihood that a desired semantic form occurs in the corpus, the *transformation* into parse trees provides rules that specify an underlying grammar for drugs of interest from the provided examples.

10.2 Peptide Filter

The `gen` module provided by *MediSyn* interfaces with an optional `filter` module which can be utilized to prevent synthesis and evaluation of undesirable candidates. For *PepSyn*, we implemented a simple inference model that either accepts or rejects each candidate based on inferred cytotoxicity [84].

Certain peptide sequences may have properties that cause them to be prohibitively difficult to (chemically) synthesize or may have toxic effects; we filter out peptide candidates that fall below thresholds listed in [159]. For example, Fig. 10.1 depicts a syntactic specification that could result in proposals being generated that are of the family of neurotoxic "conotoxins," *one of the most toxic substances known to man* that is subject to the same government regulations as Anthrax and Ebola Virus [31]. Furthermore, the candidate structure's two unbound holes provide opportunities to propose candidates that would be difficult to synthesize. Of the proposal candidates depicted in Fig. 10.1, the candidate `CCNPACGRNFSC` conforms to a conotoxin, while `CCNPCCGCCCNFSC` contains a high ratio of cysteine amino acids, increasing its likelihood of instability [159]. We filter these candidates out before the `gen` module provides its list of the remaining preferred candidates to the `core` module.

10.3 Back-end

The `back-end` module we implement for *PepSyn* implements *in silico* evaluation. While *MediSyn*'s `back-end` module is capable of completing a cyber-physical feedback loop to control a physical apparatus that synthesizes and evaluates proposed candidates (Chapter 9), our evaluation here is limited to a purely computational approach. *PepSyn*'s

`back-end` module implements the `synthesize` method by returning a unique identifier (to simulate successful synthesis), and estimates each candidate's MIC *in silico* using the `evaluate` method, discussed next.

10.3.1 MIC Estimation

With peptides, form follows function, and sequence alignment provides a reasonable approximation of biologically related forms; hence, one can expect peptides having significant sequence similarity to also have similar antimicrobial function [10, 9]. The `evaluate` method implemented by *PepSyn*'s `back-end` module infers a point estimate of a candidate peptide's minimum inhibitory concentration (MIC) assuming a statistical model where peptides that have significant sequence similarity also have similar antimicrobial function. Let V be the user-provided specification of peptides with experimentally validated MICs, where each $v \in V = (v_{seq}, v_{mic})$[6] We estimate the MIC $\mathbf{c}_{mic}$ of a candidate $\mathbf{c}$ utilizing sequence alignments of $\mathbf{c}$ against each $v \in V$. Shown in Eq. (10.18), $\mathbf{c}_{mic}$ is computed as the geometric mean of the ratio of v_{mic} and the similarity of v_{seq} with $\mathbf{c}$ for all $v \in V$:

$$\mathbf{c}_{mic} = \left(\prod_{v \in V} \frac{v_{mic}}{norm\_align_{v_{seq}}(\mathbf{c})} \right)^{\frac{1}{|V|}}, \tag{10.18}$$

where $norm\_align_{v_{seq}}$ is a pairwise sequence alignment function associated with v_{seq} that computes a *normalized* sequence similarity score against the candidate c. We normalize raw alignments due to interpretation difficulties in the presence of e.g., varied lengths or scoring parameters. The alignment in Fig. 10.8 illustrates the issue: a 100% match with one alignment scores the same as an alignment where 1/3 of the amino residues are different or missing from the candidate. To compute a raw alignment score, we use

[6]When using *PepGen*, $V = \mathcal{D}$; when using *PepSketch*, the user must load V directly. We denote the pair as (v_{seq}, v_{mic}) in lieu of $(\mathbf{s}_i, x_i)$ for clarity that V is not necessarily equal to $\mathcal{D}$.

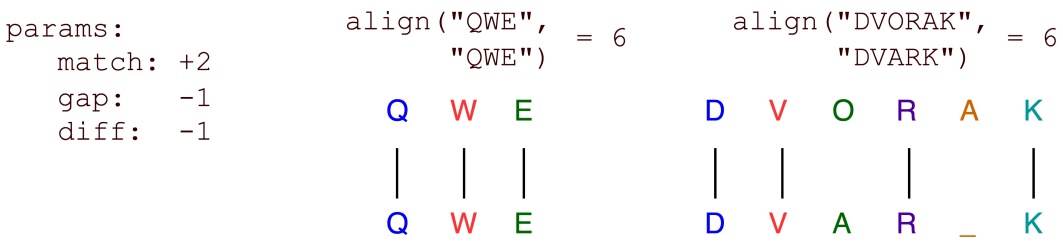


Figure 10.8: Difficulty of interpreting sequence alignment similarity scores: due to varied lengths and scoring parameters, sequence alignment scores must be normalized to provide consistent interpretation.

the Smith-Waterman local alignment algorithm [211] with a point accepted mutation (PAM) substitution matrix — a scoring matrix that directly corresponds to evolutionary mutations in peptide sequences, relating the likelihood of replacing a single amino acid at each point with any other amino acid [45]. Let $\texttt{raw}(v_{seq}, \mathbf{c})$ return a raw alignment score. In order to normalize similarity scores, each $v_{seq} \in V$ has its own *min-max* $norm\_align_{v_{seq}}$ function (Eq. (10.19)) where the max — corresponding to an exact match — is the result of aligning v_{seq} with v_{seq}, and the min — corresponding to any *least biologically-related* peptide of equal length — is computed as $w * |v_{seq}|$, where w is the penalty in the PAM matrix for substituting a wildcard (i.e., an unspecified amino acid at the given position),[7] i.e.,

$$norm\_align_{v_{seq}} = \lambda v_{seq}.\lambda \mathbf{c}.\ \frac{\texttt{raw}(v_{seq}, \mathbf{c}) - w * |v_{seq}|}{\texttt{raw}(v_{seq}, v_{seq}) - w * |v_{seq}|} \tag{10.19}$$

After evaluating c_{mic} using our normalized alignments, the result is fed back to *MediSyn*'s provides `core` module to inform MCMC acceptance.

[7] In practice, we specify $w = -8$ in the PAM matrix.

Chapter 11

Evaluation

Our stated goals for *MediSyn* is to

1. *automate* the discovery process

2. reduce that normative costs necessary for discovering viable drug candidates

We assess goal 1 by (a) characterizing the candidate spaces (i.e., PCFGs) that *PepSyn*'s *PepSketch* and *PepGen* front-ends generate and (b) observing acceptance rates and mixing times for *MediSyn*'s MCMC implementation. If a bespoke *PepSketch* and a corresponding corpus selection in *PepGen* are able to generalize search spaces corresponding to drugs with known activity, then *PepSyn* is able to automate the generation of good priors distributions. If *MediSyn*'s MCMC implementation converges to a stable distribution, then we can conclude that *MediSyn* is able to automate the optimizing search over the provided candidate space. We assess goal 2 by way of extrapolation; i.e., given the

observations from (a) and (b), we estimate ranges of costs associated with each search if the back-end were to rely on physically synthesizing and evaluating each candidate.

The remainder of this section details our choice of benchmarks (Section 11.1) and evaluation methodologies (Section 11.2) prior to reporting and discussing results in Sections 11.3 and 11.4.

11.1 Benchmarks

Table 11.1 characterizes a set of benchmark pairs used to evaluate goals 1 and 2. For each benchmark, we provide a numbered identifier of the form $\#_{FrontEnd}$, where $FrontEnd$ is either PS (for *PepSketch*) or PG (for *PepGen*), and a short description of what the benchmark entails.

Table 11.1: Benchmarks

ID	Description
1_{PS} 1_{PG}	These aim to emulate the *protegrin* family of peptides, a family of short peptides (typically 16-18 amino acids) with antimicrobial and antiviral applications [163].
2_{PS} 2_{PG}	We target the ATCC 25922 strain of *e. coli*, a standard used for testing AMPs, with these benchmarks [175].
3_{PS} 3_{PG}	Coronaviruses, a family of crown-shaped microbes that have received a lot of attention as of late, are targeted by these benchmarks [124].
4_{PS} 4_{PG}	Distinctin, a family of AMPs found in tree frogs [43, 17].
5_{PS} 5_{PG}	This pair of benchmarks combines emulation of protegrin (as in the first pair of benchmarks) while focusing on targeting the ATCC 25922 strain of *e. coli*.

11.2 Methods

We utilize each pair of benchmarks to characterize each front-end's ability to generate a PCFG. For each benchmark, we manually create *PepSketch* expressions that generalize the antimicrobial peptides associated with the goal; e.g., the *PepSketch* from Fig. 10.3

aims to emulate *protegrin* for 1_{PS}. For *PepGen* versions of each benchmark, we query each drug/target name in the descriptions as described in Section 10.1.2. Evaluation of each benchmark pair uses an identical set of peptides with known antimicrobial activity, leading to meaningful direct comparisons. We assess each of goals 1 and 2 by answering the following questions:

Q1. Is *PepSketch*/*PepGen* able to generalize a candidate search space? How expressive are the languages induced by *PepSketch*/*PepGen*?

Q2. Does the MCMC approach described in Chapter 9 work on the induced grammars from *PepSyn*? Can it achieve good mixing? If so, how many candidates must be evaluated prior to converging to a stable distribution?

Q3. What approximate costs are necessary to chemically synthesize candidates before the posterior converges?

11.3 Results

Results for assessing goal 1 for each benchmark are listed in Tables 11.2 and 11.3. Table 11.2 reports details of the probabilistic grammar G_P and its language $\mathcal{L}(G_P)$ that are induced through each method; we report:

- The number of terminal symbols $|\Sigma|$, non-terminal symbols $|\mathcal{N}|$, and rules $|\mathcal{R}|$ in G_P.
- The *size* of the search space $|\mathcal{L}(G_P)|$, i.e., the number of peptides $\mathcal{L}(G_P)$ specifies.
- The average length $\mathbf{avg}\,|\mathbf{c}|$ sampled candidates in $\mathcal{L}(G_P)$.

- The number of holes/terms used in describing the benchmark. For *PepSketch*, we use the term *holes* to refer to a position in the sequence specified as an *optional, repeat,* or *wildcard* expression. For *PepGen*, we use *terms* to refer to the number of search terms used to select $\mathcal{D}$.

Table 11.3 reports details regarding the MCMC optimization search provided by *MediSyn*. We report *acceptance rate* α, *mixing time* Mix, the average length of a sampled candidate $\mathbf{avg}\,|\mathbf{c}|$, and approximations of cost requirements if evaluations were to occur *ex silico*.

11.4 Discussion

Table 11.2 helps us answer **Q1**; by taking the geometric mean of each feature (for each front-end), we can find the average difference—denoted **avg-diff**—between the approaches. We find that a *PepSketch* expression has, on average, 585.27 fewer terminals, 21.02 fewer non-terminals, and 14,179.49 fewer productions rules than *PepGen*, leading to grammars that are significantly more concise, and spaces that on average have 1.07E+40 fewer candidates. Even so, the average size of candidate lengths are similar, where a *PepGen* candidate is only ∼4 more amino acids than one from *PepSketch*. While *PepGen* grammars are objectively more expressive, they come from significantly less user-input: we note that there are on average 13.55 more holes in *PepSketch* expressions than the number of terms we needed to select to perform the same benchmarks. *PepSketch* does not generate a treebank; on average, it's transformation into $\mathcal{L}(G_P)$ takes ∼43 fewer minutes than *PepGen*'s.

Table 11.2: Characteristics of the PCFGs that *PepSyn*'s two front-ends create.

ID	$\lvert\Sigma\rvert$	$\lvert\mathcal{N}\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{L}(G_P)\rvert$	**avg |c|**	**Holes / Terms**	**Time To Generate (mm:ss.ms)**		
							$\mathcal{L}^{\dagger}_{G_{d_i}}$	$\mathcal{T}^{\dagger}_{\mathcal{D}}$	$\mathcal{L}(G_P)$
1_{PS}	23	5	35	1.24E+05	17	5			00:00.063
1_{PG}	265	38	12347	3.77E+18	17	1	00:00.425	01:17.637	00:18.592
2_{PS}	23	12	43	4.96E+05	18	12			00:00.088
2_{PG}	4427	39	224378	1.00E+85	36	1	00:32.353	04:05.974	04:55.667
3_{PS}	23	22	71	1.53E+12	49	22			00:00.196
3_{PG}	715	38	915	4.84E+52	51	1	00:00.030	00:37.768	00:14.864
4_{PS}	23	20	73	2.46E+16	28	20			00:00.076
4_{PG}	166	30	166397	3.27E+19	30	1	00:10.173	01:45.645	02:08.212
5_{PS}	23	26	85	4.92E+16	22	26			00:00.067
5_{PG}	598	39	1387	2.35E+24	20	2	00:00.509	00:37.851	00:15.180
avg-diff‡	**585.27**	**21.92**	**14,179.49**	**1.07E+40**	**3.75**	**-13.55**			**00:43.592**

† - results specific to *PepGen* ‡ - difference of geometric means of each feature for each front-end

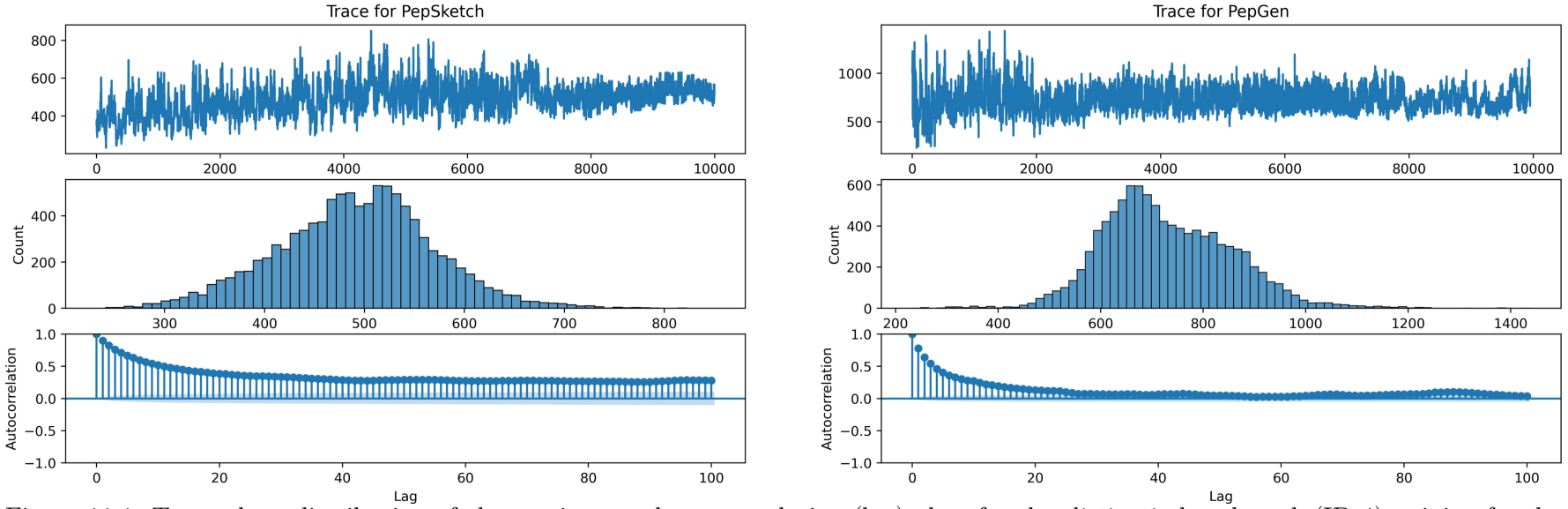


Figure 11.1: Trace plots, distribution of observations, and autocorrelation (lag) plots for the *distinctin* benchmark (ID 4); mixing for the *PepGen* (right) technique suggests superior stability of the distribution in this benchmark.

Table 11.3: The acceptance rate (α), time to converge to a posterior distribution (Mix), average length of a peptide candidate **avg** $|\mathbf{c}|$, and approximate *costs* required to perform experimentation *ex silico*.

ID	$\alpha^{\dagger}$	$Mix^{\dagger,\ddagger}$	**avg** $\lvert\mathbf{c}\rvert$	$\sim Cost$	
				Lower	**Upper**
1_{PS}	92.95%	0	17	\$3,188	\$42,500
1_{PG}	92.19%	250	17	\$31,875	\$425,000
2_{PS}	96.34%	0	18	\$3,375	\$45,000
2_{PG}	97.66%	200	36	\$54,000	\$720,000
3_{PS}	97.76%	0	49	\$9,188	\$122,500
3_{PG}	95.09%	250	51	\$95,625	\$1,275,000
4_{PS}	97.58%	0	28	\$5,250	\$70,000
4_{PG}	96.07%	0	30	\$5,625	\$75,000
5_{PS}	97.22%	50	22	\$8,250	\$110,000
5_{PG}	94.89%	250	20	\$37,500	\$500,000

† - avg. of 10 independent chains, ‡ - nearest 50

Q2 is resolved with Table 11.3; *MediSyn*'s provided MCMC technique shows high acceptance rates of proposal candidates; this either indicates that the jumping distribution does not provide for significant exploration, or that the acceptance ratio often evaluates close to 1 even when jumps are distant. As a jump can be selected as high up as the start symbol in the parse tree, any candidate in the grammar is a possible proposal; this may indicate that the grammars we generate distribute candidates with high likelihood of correlation w.r.t. their antimicrobial activity. Observed mix times (using trace plots, e.g., Fig. 11.1) are insightful: the *PepSketch* approach began its search within the posterior for all but one benchmark. The *PepGen* approach required 200 iterations prior to convergence on average.

Table 11.3 provides low and high estimates of costs to answer **Q3**: we associate a cost-per-amino acid based on typical reporting (Section 8.1.3). For benchmarks requiring zero burn-in, we assume a search of at least 25 candidates. Using a low bound of \$7.50 and high bound of \$100, *PepSketch* averages (by the geometric mean) between $\sim$\$5,000-\$70,000, while *PepGen* averages between $\sim$\$32,000-\$430,000. While *PepSketch*'s propensity

to begin in its target distribution provides potential cost savings over *PepGen*, the boundaries for *PepGen*'s approximate costs are troublesome in the light of the estimated $2.6 billion that is typical for drug discovery [169].

There are several threats to the validity of our results. Namely, high acceptance rates during MCMC could indicate lack of exploration. Moreover, our evaluation metric (Section 10.3) uses *in silico* point approximations that may not correlate to physical evaluations. Finally, the cost approximations we provide, while given a range of upper and lower bounds to err on the side of caution, do not account for e.g., personnel, equipment/machinery, etc., that may be typical for pharmaceutical corporations.

Chapter 12

Related

In addition to the relevant material in program synthesis and superoptimization this work takes inspiration from presented in Chapter 8, there is a breadth of literature in the context of bioinformantics (for PCFG induction) and machine learning (for drug discovery).

Grammar Induction for Protein Modeling: Several efforts in inducing PCFGs for modelling protein sequences for various purposes have been proposed [202, 57, 229] [202] utilizes an n-gram Bayesian classifier to annotate non-terminals with classification results while inducing a (non-probabilistic) context-free grammar from a corpus of frog antimicrobial peptides. Their CFG rules are modified with probabilities correlating to the cardinality of discovered clusters. [57] estimates the probabilities for rules on an underlying (non-probabilistic) CFG in a way similar to [181], in which parentheses are used to denote semantically related structures in otherwise unlabeled corpora to estimate a PCFG using the inside-outside algorithm [13]. [57] utilizes computationally derived

rules for protein contact map constraints as a proxy to [181]'s parenthetical annotations, which model sequences in a given protein that are likely to have close spatial proximity in their folding. Our *PepGen* approach accomplishes the similar goal of finding semantically similar motifs, but our weighting of rules is accomplished through augmentation using perturbations of nearest-neighbor 3-mers in the trained word embedding vector space.

Machine Learning Approaches to Drug Discovery: There has been a significant effort in applying machine learning (ML) to the task of drug discovery over the past several decades, well beyond the scope of what is reasonable to cover here, ranging from simple models (e.g., SVM's [132], decision trees [144], and linear regressors [89]) to more complex deep-learning approaches [36],[1] and several pharmaceutical companies are approaching drug discovery directly with ML [193, 165, 116, 44, 247, 252] A recent review ([59]) discusses the latest advancements and their shortcomings, including a lack of transparency (i.e., the ability to interpret resulting models) and the requirement for large (typically labeled) datasets. Emerging techniques that might overcome these limitations are highlighted, including applying recommendation systems (for semi-interpretable results) [214] and transfer learning (for learning from small datasets) [244, 83, 204]. In contrast, our approach mitigates these concerns by design — formal grammars lead to direct interpretation, and user-supplied specifications to optimize over can be few. The most direct relation to our approach is in the design of *PepGen*, where we use data augmentation to overcome issues with smaller datasets.

[1] Excellent reviews are regularly published, cataloging efforts in this domain [231, 36, 59, 118, 132, 248].

Chapter 13

Conclusion

This part presented *MediSyn*, a framework that aims to reduce the high costs and time associated with drug discovery and development by automating the early phases using techniques originally developed for superoptimization of program specifications in program synthesis. The modular architecture of *MediSyn* provides for straightforward extension and specialization for *de novo* discovery of viable drugs of differing types. *MediSyn*'s utility was demonstrated with *PepSyn*, a proof-of-concept implementation for pharmaceutical peptides, featuring a domain-specific-language front-end that allows researchers to succinctly specify a candidate space we call *PepSketch* and a secondary front-end that induces a search space based on a selection of known pharmaceutical peptides. Evaluation shows that *MediSyn*'s Markov Chain Monte Carlo technique achieves good mixing on the distributions of peptide drugs provided by *PepSyn*, typically with little need for burn-in. The contributions provide a hopeful path forward for life scientists to adopt emerging technologies in their workflows, and have the potential to disrupt the $500 billion US pharmaceutical industry. Future work for *MediSyn* should

investigate novel back-end techniques that complete a cyber-physical feedback loop, where physical candidate evaluation replaces (or augments) *in silico* inference, implement specifications for new domains (e.g., small molecules, D/RNA-based drugs, etc.), and investigate different synthesis and/or enumeration techniques.

Conclusion

Life scientists are in need of disruptive tools to improve the efficiency, costs, and reproducibility of their important work.

In Part I, this dissertation addressed ways in which these problems could be solved for biological and chemical scientists who utilize or otherwise rely on analytical biochemical *assays* that can be automated, miniaturized, and accelerated using programmable laboratories-on-a-chip (pLoCs), but are currently performed at the benchtop. It presented practical solutions to gaps in the end-to-end workflow of programming, compiling, and executing assays on commercially-available pLoCs.

Part II presented the modular *MediSyn* framework as a solution to reduce the high costs and time associated with drug discovery and development, and demonstrated its nascent utility with *PepSyn*, a proof-of-concept peptide-specific implementation of *MediSyn*. These contributions provide a practical path forward for life scientists to adopt emerging technologies in their workflows, and have the potential to disrupt the $500 billion US pharmaceutical industry with faster and cheaper solutions for discovering and developing life-saving drugs.

Appendix A

Mix Module Resizing Example

We provide a simple mixing tree as an example of our resizing heuristic (Fig. A.1). The assay uses the time t specified for a 2×2 mixer. The assay, converted into its fluidic dependency graph (shown as a directed acyclic graph (DAG)) (Fig. A.2a), has a width of 4. The following discussion details how we find the width.

```
1  /* dispense a, b, c, d, e, f, g, h */
2  ab = mix a with b for 10s
3  cd = mix c with d for 10s
4  ef = mix e with f for 10s
5  gh = mix g with h for 10s
6  abcd = mix ab with cd for 10s
7  efgh = mix ef with gh for 10s
8  abcdefgh = mix abcd with efgh for 10s
9  drain abcdefg
```

Figure A.1: A Synthetic Assay: the specification given as a BioScript protocol. Default mixing times are given by the scientist, typically based on a 2×2 module's latency (Table 2.2)

From visual inspection, the DAG in Fig. A.2a can clearly parallelize four instructions (nodes 1, 2, 3, and 4). While this is clear in this case through visual inspection, for any arbitrary assay it is not so. By finding a maximum antichain of the DAG of width w, we reveal an upper limit on instruction level parallelism for a given assay. Figure A.2c

depicts a maximum matching on the bipartite graph derived from the DAG in Fig. A.2a. From this matching, we partition the DAG into the chains shown in Fig. A.2b from which we find $w = 4$.

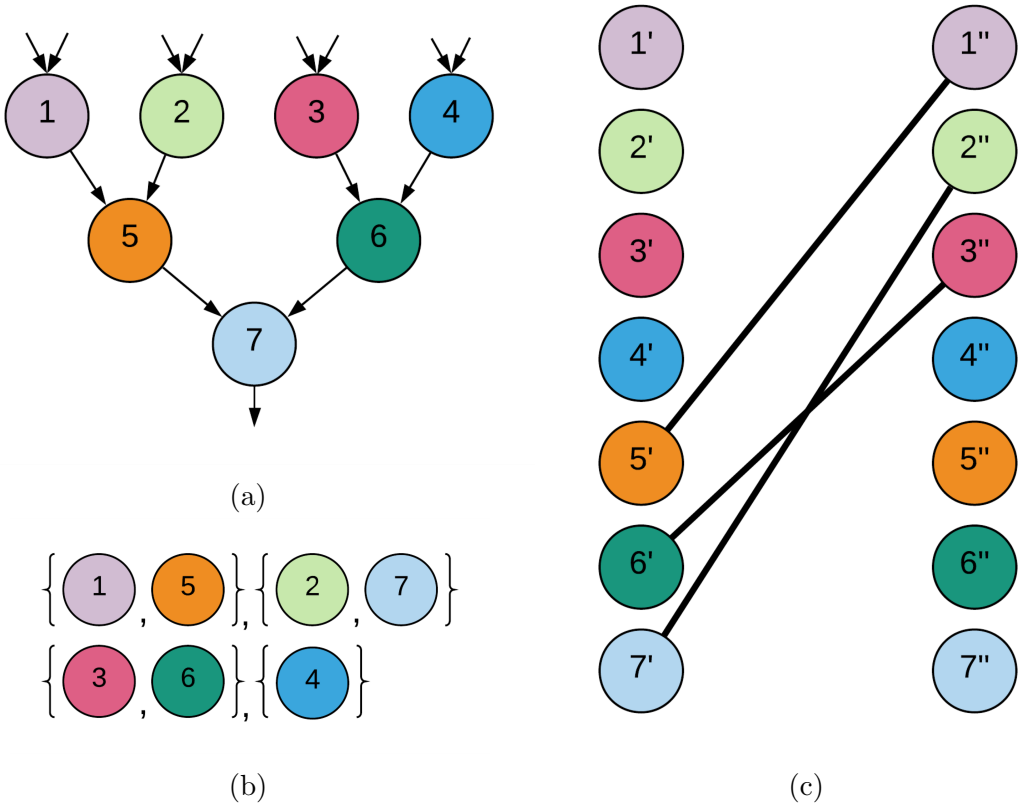


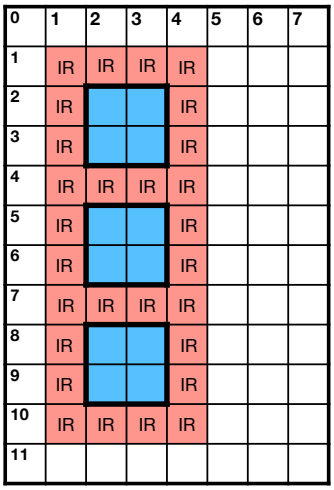
Figure A.2: **The dependency graph and derived bipartite graph for the Assay in Figure A.1:** The dependency graph (a) has a width $w = 4$, corresponding to a maximum number of 4 parallel operations. To find the maximum number of parallel operations, we find a maximum matching on the dependency graph's derived bipartite graph (c), and use this to partition the DAG into a set of w chains (b)

Figure A.3 shows the number of work modules the scheduler allocates for different module sizes given an architecture of 8×12. We can fit three of the various $2 \times y$ modules (Figs. A.3a to A.3c), or four 1×4 modules (Fig. A.3d). Given this architecture, Algorithm 1 returns the 1×4 module; maximizing operation-level parallelism within the block.

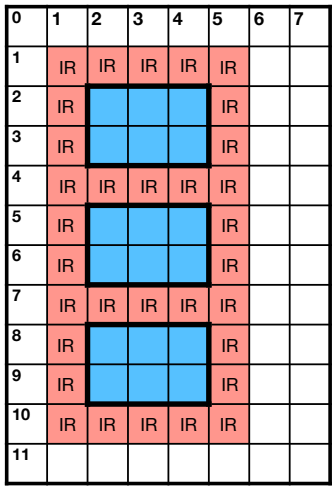
Prior to resizing, the scheduled time to execute this assay is 40 seconds, noting only three operations can be performed in parallel with the given size and any I/O latencies are amortized away. After resizing, each mix operation's latency is reduced to $10 \times 4.6/9.95 \approx 4.62$ seconds. The total rescheduled time is then ≈ 13.87 seconds, ignoring I/O latencies[1].

This example highlights the difficulty of finding an optimal size for a mix module. In this case, an optimal choice would be to use three 2×4 modules rather than four 1×4 modules, as the 2×4 modules would reduce the schedule to ≈ 11.66 seconds, ignoring I/O latencies. While true in this particular case, it does not generalize. Specifically, assays do not always have the binary tree structure as given in Fig. A.2a, and typically have varying amounts of operation-level parallelism at different depths. As the parallelism in Fig. A.2a monotonically decreases with its depth, the benefits of exploiting the maximum parallelism is restricted, whereas an arbitrary assay may have significant gains through increased parallelism throughout.

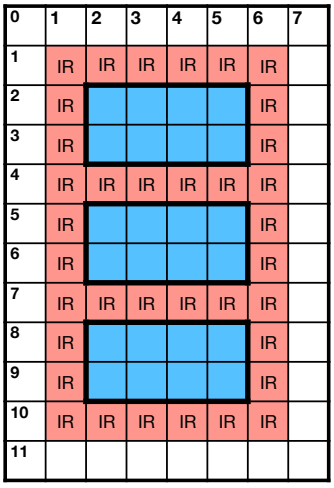
[1]This is a simplification. In reality, I/O operations take time, which we allocate for during scheduling. For example, this particular assay is scheduled for 17 seconds to account for I/O latencies



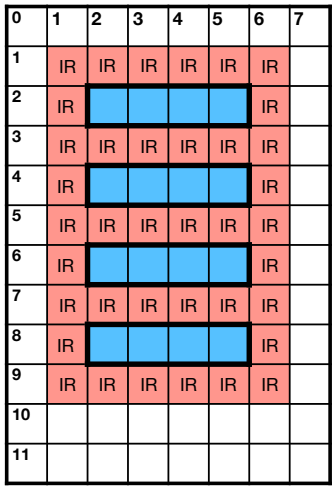
(a) 2 × 2 - 3 modules



(b) 2 × 3 - 3 modules



(c) 2 × 4 - 3 modules



(d) 1 × 4 - 4 modules

Figure A.3: Given an 8 × 12 chip, the scheduler will allocate as many work modules as will fit for a given size, with room left for droplet routing. Cells marked "`IR`" form an interference region around work modules, in which droplets cannot be routed. The scheduler ensures that work modules provide enough room for vertical routing *streets* (in the case where we have more than a single column of work modules), as well as an `IR`-free perimeter for I/O routing.

Appendix B

Benchmarks

B.1 Benchmarks for Chapter 4

Listings for each of the benchmarks used in Chapter 3 follow. While many of these are available in [178, 41]'s supplemental materials, they have all been updated to match syntax updates in BioScript.

B.1.1 Benches from [178]

Listing B.1: BroadSpectrumOpiate ([12, 108, 146]).

```
1  module fluorescence
2
3  manifest Anti_Morphine
4  manifest Anti_Oxy
5  manifest Anti_Fentanyl
6  manifest Anti_Ciprofloxcin
7  manifest Anti_Heroin
8  manifest UrineSample
9
10 instructions:
11
12 us1 = dispense 10 units of UrineSample
13 us2 = dispense 10 units of UrineSample
14 us3 = dispense 10 units of UrineSample
15 us4 = dispense 10 units of UrineSample
16 us5 = dispense 10 units of UrineSample
17
```

```
aa = dispense 10 units of Anti_Morphine
a = mix us1 with aa
bb = dispense 10 units of Anti_Oxy
b = mix us2 with bb
cc = dispense 10 units of Anti_Fentanyl
c = mix us3  with cc
dd = dispense 10 units of Anti_Ciprofloxcin
d = mix us4 with dd
ee = dispense 10 units of Anti_Heroin
e = mix us5 with ee

MorphineReading = detect fluorescence on a for 5s
OxyReading = detect fluorescence on b for 5s
FentanylReading = detect fluorescence on c for 5s
CiproReading = detect fluorescence on d for 5s
HeroinReading = detect fluorescence on e for 5s

dispose a
dispose b
dispose c
dispose d
dispose e
```

Listing B.2: CancerDetection ([210]).

Listing B.3: Ciprofloxacin ([108]).

```
module fluorescence

manifest ciprofloxacin_enzyme
manifest distilled_water
manifest ciprofloxacin_conjugate
manifest tmb_substrate
manifest urinesample
manifest stop_reagent

instructions:

us = dispense 20 units of urinesample
cfc = dispense ciprofloxacin_conjugate
cfe = dispense ciprofloxacin_enzyme

a = mix us with cfe
b = mix cfc with a for 60s
heat b at 23c for 60m
dispose b

repeat 5 times {
    water = dispense 250 units of distilled_water
    cfe = dispense ciprofloxacin_enzyme
    temp = mix water with cfe for 45s
    dispose temp
}
tmb = dispense 50 units of tmb_substrate
cfe = dispense ciprofloxacin_enzyme

d = mix tmb with cfe
```

```
heat d at 25c for 30m

cfe = dispense ciprofloxacin_enzyme
stop = dispense 100 units of stop_reagent
e = mix cfe with stop for 60s

urine_reading = detect fluorescence on e for 5m
dispose d
dispose e
```

Listing B.4: Diazepam ([91]).

```
module fluorescence

manifest diazepam_enzyme
manifest urinesample
manifest diazepam_antibody
manifest distilled_water
manifest stop_reagent
manifest hrp_conjugate
manifest tmb_substrate

instructions:

urine = dispense 50 units of urinesample
dpe = dispense diazepam_enzyme
a = mix urine with dpe for 60s
anti = dispense 100 units diazepam_antibody
b = mix a with anti for 60s
heat b at 23c for 30m
dispose b

repeat 3 times {
    water = dispense 250 units of distilled_water
    dpe = dispense diazepam_enzyme
    a = mix water with dpe for 45s
    dispose a
}

hrpc = dispense 150 units of hrp_conjugate
dpe = dispense diazepam_enzyme
cc = mix hrpc with dpe
heat cc at 23c for 15m
dispose cc

repeat 3 times {
    water = dispense 250 units of distilled_water
    dpe = dispense diazepam_enzyme
    a = mix water with dpe for 45s
    dispose a
}

tmb = dispense 100 units of tmb_substrate
dpe = dispense diazepam_enzyme
d = mix tmb with dpe
heat d at 23c for 15m

stop = dispense 100 units of stop_reagent

```

```
reagent = mix d with stop for 60s
Negative_Reading = detect fluorescence on reagent for 30m
dispose reagent
```

Listing B.5: Dilution ([91]).

```
manifest substance_a
manifest substance_b
manifest substance_c
manifest dilutant1
manifest dilutant2
manifest dilutant3

instructions:

sa = dispense 10 units of substance_a
d1 = dispense 1 units of dilutant1

first_dilute = mix sa with d1
x = split first_dilute into 2

dispose x[0]

d2 = dispense 1 units of dilutant2

second_dilute = mix x[1] with d2
y = split second_dilute into 2
dispose y[0]

d3 = dispense 1 units of dilutant3

third_dilute = mix y[1] with d3
z = split third_dilute into 2
dispose z[0]

sb = dispense 10 units of substance_b
sc = dispense 10 units of substance_c

fourth_dilute = mix sb with sc
a = split fourth_dilute into 2
dispose a[0]

final_dilute = mix z[1] with a[1]
b = split final_dilute into 2
dispose b[0]
dispose b[1]
```

Listing B.6: Fentanyl et al. ([146]); several enzyme-linked-immunosorbent assay (ELISA) variants exist, all following the same structure.

```
module fluorescence

manifest antigen
manifest urine_sample
manifest fentanyl_conjugate
manifest tmb_substrate
manifest distilled_water
manifest stop_reagent
```

```
9
10 instructions:
11
12 aa = dispense 20 units of urine_sample
13 aaa = dispense antigen
14 a = mix aa with aaa
15 bb = dispense 100 units of fentanyl_conjugate
16 b = mix bb with a for 60s
17 heat b at 23c for 60m
18 dispose b
19
20 repeat 6 times {
21     zz = dispense 350 units of distilled_water
22     z = mix zz with a for 45s
23     dispose z
24 }
25
26 aaaa = dispense 100 units of tmb_substrate
27 a = mix aaaa with a
28 heat a at 23c for 30m
29
30 aaaaa = dispense 100 units of stop_reagent
31 a = mix a with aaaaa for 60s
32 negative_reading = detect fluorescence on a for 30m
33 dispose a
```

Listing B.7: FullMorphine ([91]).

```
1  module fluorescence
2
3  manifest Antigen1
4  manifest Antigen2
5  manifest Antigen3
6  manifest morphine_conjugate
7  manifest negative_standard
8  manifest diluted_sample
9  manifest positive_standard
10 manifest distilled_water
11 manifest tmb_substrate
12 manifest stop_reagent
13
14 instructions:
15 d = dispense 20 units of negative_standard
16 e = dispense Antigen1
17 a = mix d with e
18 f = dispense 20 units of positive_standard
19 g = dispense Antigen2
20 b = mix f with g
21 h = dispense 20 units of diluted_sample
22 i = dispense Antigen3
23 cc = mix h with i
24
25 dd = dispense 100 units of morphine_conjugate
26 a = mix dd with a for 60s
27 ff = dispense 100 units of morphine_conjugate
28 b = mix ff with b for 60s
29 hh = dispense 100 units of morphine_conjugate
30 cc = mix hh with cc for 60s
31
```

```
heat a at 23c for 60m
heat b at 23c for 60m
heat cc at 23c for 60m

dispose a
dispose b
dispose cc

repeat 6 times {
    j = dispense 350 units of distilled_water
    k = dispense Antigen1
    aa = mix j with k for 45s
    l = dispense 350 units of distilled_water
    m = dispense Antigen2
    bb = mix l with m for 45s
    n = dispense 350 units of distilled_water
    o = dispense Antigen3
    cc = mix n with o for 45s

    dispose aa
    dispose bb
    dispose cc
}

jj = dispense 100 units of tmb_substrate
kk = dispense Antigen1
aa = mix jj with kk
ll = dispense 100 units of tmb_substrate
mm = dispense Antigen2
bb = mix ll with mm
nn = dispense 100 units of tmb_substrate
oo = dispense Antigen3
cc = mix nn with oo

heat aa at 23c for 30m
heat bb at 23c for 30m
heat cc at 23c for 30m

p = dispense stop_reagent
aa = mix p with 100 units of aa for 60s
q = dispense stop_reagent
bb = mix q with 100 units of bb for 60s
r = dispense stop_reagent
cc = mix r with 100 units of cc for 60s

negative_reading = detect fluorescence on aa for 30m
positive_reading = detect fluorescence on bb for 30m
sample_reading = detect fluorescence on cc for 30m

dispose aa
dispose bb
dispose cc
```

Listing B.8: GlucoseDetection ([6]).

```
module fluorescence

manifest reagent
manifest glucose
```

```
manifest distilled_water
manifest Sample

instructions:

aa = dispense 10 units of glucose
bb = dispense 10 units of reagent
result1 = mix aa with bb for 10s
reading1 = detect fluorescence on result1 for 30s
aaa = dispense 10 units of distilled_water
bbb = dispense 10 units of reagent
a = mix aaa with bbb for 30s
dispose a

cc = dispense 10 units of glucose
dd = dispense 20 units of reagent
result2 = mix cc with dd for 10s
reading2 = detect fluorescence on result2 for 30s
ccc = dispense 10 units of distilled_water
ddd = dispense 10 units of reagent
a = mix ccc with ddd for 30s
dispose a

ee = dispense 10 units of glucose
ff = dispense 40 units of reagent
result3 = mix ee with ff for 10s
reading3 = detect fluorescence on result3 for 30s
eee = dispense 10 units of distilled_water
fff = dispense 10 units of reagent
a = mix eee with fff for 30s
dispose a

gg = dispense 10 units of glucose
hh = dispense 80 units of reagent
result4 = mix gg with hh for 10s
reading4 = detect fluorescence on result4 for 30s
ggg = dispense 10 units of distilled_water
hhh = dispense 10 units of reagent
a = mix ggg with hhh for 30s
dispose a

ii = dispense 10 units of glucose
jj = dispense 10 units of reagent
result5 = mix ii with jj for 10s
reading5 = detect fluorescence on result5 for 30s
iii = dispense 10 units of distilled_water
jjj = dispense 10 units of reagent
a = mix iii with jjj for 30s

dispose a
dispose result1
dispose result2
dispose result3
dispose result4
dispose result5
```

Listing B.9: ImageProbeSynth ([6]).

```
manifest ion_exchange_beads
```

```
manifest fluoride_ions_f
manifest mecn_solution
manifest hydrochloric_acid

instructions:

ieb = dispense 10 units of ion_exchange_beads
fif = dispense 10 units of fluoride_ions_f

aa = mix ieb with fif for 30s

heat aa at 100c for 30s
heat aa at 120c for 30s
heat aa at 135c for 3m

ms = dispense 10 units of mecn_solution
bb = mix aa with ms for 30s

heat bb at 100c for 30s
heat bb at 120c for 50s

hcl = dispense 10 units of hydrochloric_acid

cc = mix bb with hcl for 60s
heat cc at 60c for 60s

dispose cc
```

B.1.2 Benches from [41]

Listing B.10: OpiateDetection, adapted from ([12, 146, 108, 158]); the various versions listed refer to control-decisions made at (simulated) runtime.

```
module fluorescence

manifest Anti_Morphine
manifest Anti_Oxy
manifest Anti_Fentanyl
manifest Anti_Ciprofloxcin
manifest Anti_Heroin
manifest UrineSample
manifest DistilledWater
manifest TMBSubstrate
manifest StopReagent

stationary HeroinEnzyme
manifest HeroinConjugate
stationary CiproEnzyme
manifest CiproConjugate
stationary OxyEnzyme
manifest OxyConjugate
stationary FentanylEnzyme
manifest FentanylConjugate

instructions:

// BroadSpectrumOpiate panel
```

```
us1 = dispense 10 units of UrineSample
us2 = dispense 10 units of UrineSample
us3 = dispense 10 units of UrineSample
us4 = dispense 10 units of UrineSample
us5 = dispense 10 units of UrineSample

aa = dispense 10 units of Anti_Morphine
a = mix us1 with aa
bb = dispense 10 units of Anti_Oxy
b = mix us2 with bb
cc = dispense 10 units of Anti_Fentanyl
c = mix us3  with cc
dd = dispense 10 units of Anti_Ciprofloxcin
d = mix us4 with dd
ee = dispense 10 units of Anti_Heroin
e = mix us5 with ee

MorphineReading = detect fluorescence on a for 5s
OxyReading = detect fluorescence on b for 5s
FentanylReading = detect fluorescence on c for 5s
CiproReading = detect fluorescence on d for 5s
HeroinReading = detect fluorescence on e for 5s

dispose a
dispose b
dispose c
dispose d
dispose e

// true branch
if (MorphineReading >= 0.75 or OxyReading >= 0.75 or FentanylReading >= 0.75 or CiproReading >= 0.75 or HeroinReading >= 0.75) {
    hs = mix 20 units of UrineSample with 100 units of HeroinConjugate at HeroinEnzyme for 1m
    cs = mix 20 units of UrineSample with 100 units of CiproConjugate at CiproEnzyme for 1m

    heat hs at 23c for 60m
    heat ms at 23c for 60m

    dispose hs
    dispose ms

    // wash enzymes
    repeat 6 times {
        wash[2] = dispense 350 units of DistilledWater
        send wash[0] to HeroinEnzyme for 45s
        send wash[1] to CiproEnzyme for 45s
        drain wash
    }

    tmb1 = dispense 100 units of TMBSubstrate
    tmb2 = dispense 50 units of TMBSubstrate

    send tmb1 to HeroinEnzyme
    send tmb2 to CiproEnzyme

    heat tmb1 at 23c for 30m
    heat tmb2 at 23c for 30m

```

```
    stop1 = mix 100 units of StopReagent with tmb1 for 60s
    stop2 = mix 100 units of StopReagent with tmb2 for 60s

    FinalHeroinReading = detect fluorescence on stop1 for 30m
    FinalCiproReading = detect fluorescence on stop2 for 5m

    drain stop1
    drain stop2

    // if both false run oxy
    if (FinalHeroinReading <= 0.75 and FinalCiproReading <= 0.75) {
        os = mix 20 units of UrineSample with 100 units of OxyConjugate
            at OxyEnzyme for 1m

        heat os at 23c for 60m

        drain os

        // wash
        repeat 6 times {
            wash = dispense 350 units of DistilledWater
            send wash to OxyEnzyme for 45s
            drain wash
        }

        tmb = dispense 100 units of TMBSubstrate

        send tmb to OxyEnzyme

        heat tmb at 23c for 30m

        stop = mix tmb with 100 units of StopReagent for 1m

        FinalOxyReading = detect fluorescence on stop for 30m

        drain stop
    }
} // else
else {
    fs = mix 20 units of UrineSample with 100 units of FentanylConjugate
        at FentanylEnzyme for 1m
    os = mix 20 units of UrineSample with 100 units of OxyConjugate at
        OxyEnzyme for 1m

    heat fs at 23c for 60m
    heat os at 23c for 60m

    drain fs
    drain os

    // wash enzymes
    repeat 6 times {
        wash[2] = dispense 350 units of DistilledWater
        send wash[0] to FentanylEnzyme for 45s
        send wash[1] to OxyEnzyme for 45s
        drain wash
    }

    tmb1 = dispense 100 units of TMBSubstrate
    tmb2 = dispense 50 units of TMBSubstrate
```

```
139
140     send tmb1 to FentanylEnzyme
141     send tmb2 to OxyEnzyme
142
143     heat tmb1 at 23c for 30m
144     heat tmb2 at 23c for 30m
145
146     stop1 = mix 100 units of StopReagent with tmb1 for 60s
147     stop2 = mix 100 units of StopReagent with tmb2 for 60s
148
149     FinalFentanylReading = detect fluorescence on stop1 for 30m
150     FinalOxyReading = detect fluorescence on stop2 for 5m
151
152     drain stop1
153     drain stop2
154 }
```

Listing B.11: PCRDropletReplacement ([104]).

```
1  module weight
2  manifest PCRMasterMix
3  manifest Template
4
5  instructions:
6
7  a = dispense 50 units of PCRMasterMix
8  b = dispense 50 units of Template
9  PCRMix = mix a with b for 1s
10
11 repeat 5 times {
12     heat PCRMix at 95c for 20s
13     volumeWeight = detect weight on PCRMix
14
15     if (volumeWeight <= 50) {
16         c = dispense 25 units of PCRMasterMix
17         d = dispense 25 units of Template
18         replacement = mix c with d for 5s
19         heat replacement at 95c for 45s
20         PCRMix = mix PCRMix with replacement for 5s
21     }
22
23     heat PCRMix at 68c for 30s
24     heat PCRMix at 95c for 45s
25 }
26
27 heat PCRMix at 68c for 5m
28 dispose PCRMix
```

Listing B.12: ProbabilisticPCR ([137]).

```
1  module fluorescence
2  manifest Buffer
3  manifest PCRMix
4
5  instructions:
6  a = dispense 10 units of PCRMix
7  b = dispense 10 units of Buffer
8  PCR_Master_Mix = mix a with b
```

```
9
10 heat PCR_Master_Mix at 94c for 2m
11
12 repeat 5 times {
13     heat PCR_Master_Mix at 94c for 20s
14     heat PCR_Master_Mix at 50c for 40s
15 }
16
17 DNA_Sensor = detect fluorescence on PCR_Master_Mix for 30s
18
19 if (DNA_Sensor <= 80) {
20     dispose PCR_Master_Mix
21 }
22
23 repeat 4 times {
24 heat PCR_Master_Mix at 94c for 20s
25 heat PCR_Master_Mix at 50c for 40s
26 }
27
28 heat PCR_Master_Mix at 70c for 60s
29 dispose PCR_Master_Mix
30
31 d = dispense 10 units of PCRMix
32 e = dispense 10 units of Buffer
33 f = mix e with d
34 dispose f
```

Listing B.13: PCR ([6]).

```
1  module fluorescence
2  manifest pcr_mixture
3
4  instructions:
5
6  a = dispense pcr_mixture
7
8  heat a at 95c for 5s
9
10 repeat 20 times {
11     heat a at 53c for 15s
12     heat a at 72c for 10s
13 }
14
15 x = detect fluorescence on a for 3m
16
17 dispose a
```

B.2 Benchmarks for Chapter 3

Listings for each of the benchmarks used in Chapter 3 follow; DAG images and code representations are available here.

B.2.1 SLE-only

Recall that all SLE-only benchmarks set $\Delta = 0$ as an assumption that a lack of explicit pause-points in the original assay indicates that reagents may be volatile. In each of the following, `@use.in 0` is used to set the Δ appropriately.

Listing B.14: Fentanyl Enzyme-linked-immunosorbent assay (ELISA) with immediate use constraints, adapted from [146]. A similar routine is used for detecting various opiates.

```
module fluorescence

stationary antigen // antigen is baked onto the top dmfb plate
manifest urine_sample
manifest fentanyl_conjugate
manifest tmb_substrate
manifest distilled_water
manifest stop_reagent

instructions:
@use.in 0s
sample = mix 20 units of urine_sample on antigen for 15s
@use.in 0s
reagent = mix 100 units of fentanyl_conjugate on antigen for 35s
@use.in 0s
mixture = mix sample with reagent for 20s

@use.in 0s
heat mixture at 23c for 20m
dispose mixture

repeat 6 times {
    wash = mix 350 units of distilled_water on antigen for 10m
    dispose wash
}

@use.in 0s
substrate = mix 100 units of tmb_substrate on antigen for 30s
@use.in 0s
heat substrate at 23c for 25m

@use.in 0s
stop = mix 100 units of stop_reagent on antigen for 10s
stop = mix stop with substrate for 16m
dispose stop

negative_reading = detect fluorescence on antigen for 30m
```

Listing B.15: Basic thermocycling PCR assay for DNA replication; pcr master is sensitive to temperature – we assume no time for storing droplets; adapted from [6].

```
module fluorescence
manifest template
manifest pcr_master
```

```
manifest forward_primer
manifest reverse_primer

instructions:
@use.in 0s
pcr_mix = mix 10 units of pcr_master with 10 units of template for 5s

/* warm up pcr_mix */
@use.in 0s
heat pcr_mix at 95c for 5s

@use.in 0s
primer_mix = mix 10 units of forward_primer with 10 units of
    reverse_primer for 5s

@use.in 0s
sample = mix pcr_mix with primer_mix for 5s

// this repeat block is inlined
repeat 20 times {
    @use.in 0s
    heat sample at 53c for 15s
    @use.in 0s
    heat sample at 72c for 10s
}

x = detect fluorescence on sample for 3m

dispose sample
```

B.2.1.1 Multiplexed

Each of the following benchmarks stress the scheduler's ability to deal with multiple parallel operations with timing constraints.

Listing B.16: Multiplexed PCR with four targets within a single template.

```
module fluorescence
manifest template
manifest ftp1, ftp2, ftp3, ftp4   // forward target primer 1, 2, 3, 4
manifest rtp1, rtp2, rtp3, rtp4   // reverse target primer 1, 2, 3, 4
manifest pcr_master_mix

instructions:

// sample preparations
p1_mix = mix 1 units of ftp1 with 1 units of rtp1 for 5s
p2_mix = mix 1 units of ftp2 with 1 units of rtp2 for 5s
p3_mix = mix 1 units of ftp3 with 1 units of rtp3 for 5s
p4_mix = mix 1 units of ftp4 with 1 units of rtp4 for 5s

@use.in 0s
pcr_mix = mix 10 units of template with 4 units of pcr_master_mix for 10s

// without use.in constraints, these could sit dormant
```

```
samples = split pcr_mix into 4

@use.in 0s
target1 = mix samples[0] with p1_mix for 5s
@use.in 0s
target2 = mix samples[1] with p2_mix for 5s
@use.in 0s
target3 = mix samples[2] with p3_mix for 5s
@use.in 0s
target4 = mix samples[3] with p4_mix for 5s

//initialization (necessary for hot-start polymerases)
@use.in 0s
heat target1 at 95c for 45s
@use.in 0s
heat target2 at 95c for 45s
@use.in 0s
heat target3 at 95c for 45s
@use.in 0s
heat target4 at 95c for 45s

//thermocycling, typically 20-50 times
repeat 3 times  {
    // denature
    @use.in 0s
    heat target1 at 98c for 15s
    @use.in 0s
    heat target2 at 98c for 15s
    @use.in 0s
    heat target3 at 98c for 15s
    @use.in 0s
    heat target4 at 98c for 15s

    //anneal
    @use.in 0s
    heat target1 at 50c for 30s
    @use.in 0s
    heat target2 at 50c for 30s
    @use.in 0s
    heat target3 at 50c for 30s
    @use.in 0s
    heat target4 at 50c for 30s

    //elongation/extension
    @use.in 0s
    heat target1 at 74c for 3m
    @use.in 0s
    heat target2 at 74c for 3m
    @use.in 0s
    heat target3 at 74c for 3m
    @use.in 0s
    heat target4 at 74c for 3m
}

//final elongation
@use.in 0s
heat target1 at 70c for 5m
@use.in 0s
heat target2 at 70c for 5m
@use.in 0s
```

```
heat target3 at 70c for 5m
@use.in 0s
heat target4 at 70c for 5m

dispose target1
dispose target2
dispose target3
dispose target4
```

Multiplexed InVitro Colorimetric Detection Listings B.17 and B.18 show the general layout for this group of benchmarks adapted from [221], which combines various colorimetric detection assays (from [218]) into a multiplexed sequencing model for collections of samples and reagents. Listing B.19 shows a `Python` script we used to generate these multiplexed `BioScript` protocols.

Listing B.17: Multiplexed InVitro with 2 samples and 2 reagents, adapted from [221].

```
module sensor
manifest Plasma
manifest Serum
manifest Glucose
manifest Lactate

instructions:
@use.in 0s
mix1 = mix 10 units of Plasma with 10 units of Glucose for 5s
det1 = detect sensor on mix1 for 5s
dispose mix1

@use.in 0s
mix2 = mix 10 units of Plasma with 10 units of Lactate for 5s
det2 = detect sensor on mix2 for 5s
dispose mix2

@use.in 0s
mix3 = mix 10 units of Serum with 10 units of Glucose for 5s
det3 = detect sensor on mix3 for 5s
dispose mix3

@use.in 0s
mix4 = mix 10 units of Serum with 10 units of Lactate for 5s
det4 = detect sensor on mix4 for 5s
dispose mix4
```

Listing B.18: Multiplexed InVitro with 2 samples and 3 reagents, adapted from [221].

```
module sensor
manifest Plasma
manifest Serum
manifest Glucose
```

```
manifest Lactate
manifest Pyruvate

instructions:
@use.in 0s
mix1 = mix 10 units of Plasma with 10 units of Glucose for 5s
det1 = detect sensor on mix1 for 5s
dispose mix1

@use.in 0s
mix2 = mix 10 units of Plasma with 10 units of Lactate for 5s
det2 = detect sensor on mix2 for 5s
dispose mix2

@use.in 0s
mix3 = mix 10 units of Plasma with 10 units of Pyruvate for 5s
det3 = detect sensor on mix3 for 5s
dispose mix3

@use.in 0s
mix4 = mix 10 units of Serum with 10 units of Glucose for 5s
det4 = detect sensor on mix4 for 5s
dispose mix4

@use.in 0s
mix5 = mix 10 units of Serum with 10 units of Lactate for 5s
det5 = detect sensor on mix5 for 5s
dispose mix5

@use.in 0s
mix6 = mix 10 units of Serum with 10 units of Pyruvate for 5s
det6 = detect sensor on mix6 for 5s
dispose mix6
```

Listing B.19: Python script for generating multiplexed InVitro diagnostics.

```
samples = ["Plasma", "Serum", "Saliva", "Urine"]
reagents = ["Glucose", "Lactate", "Pyruvate", "Glutamate"]
# choose how many samples and reagents
s = 3
r = 4
usein = True
n = 0

with open("output/InVitro_{}s_{}r.bs".format(s, r), mode='w') as file:
    file.write("\nmodule sensor")

    for i in range(1, s+1):
        file.write("\nmanifest {}".format(samples[i-1]))

    for i in range(1, r+1):
        file.write("\nmanifest {}".format(reagents[i-1]))

    file.write("\n\ninstructions:")

    count = 1
    for i in range(1, s+1):
        for j in range(1, r+1):
            if usein:
```

```
                file.write("\n@use.in {}s".format(n))
            file.write("\nmix{} = mix 10 units of {} with 10 units of {}
                for 5s".format(count, samples[i-1], reagents[j-1]))
            file.write("\ndet{} = detect sensor on mix{} for 5s".format(
                count, count))
            file.write("\ndispose mix{}\n".format(count))
            count += 1
```

B.2.1.2 Split-Dilutes

The ProteinSplit variants are sequencing assays with high fan-out; as with the InVitro benchmarks, this group is derived from [221], and use a serial-dilution method ([62]) to construct diluting Bradford reactions ([218]). Listings B.20 and B.21 provide the first two benchmarks of this type, while Listing B.22 gives a `Python` script we use to generate these benchmarks.

Listing B.20: ProteinSplit1 dilution assay, adapted from [221].

```
module sensor
manifest DsS //sample
manifest DsB //buffer
manifest DsR //reagent

instructions:
@use.in 0s
mix1 = mix 10 units of DsS with 10 units of DsB for 3s
slt1 = split mix1 into 2

// path 1
@use.in 0s
mix2 = mix slt1[0] with 10 units of DsB for 3s
@use.in 0s
mix3 = mix mix2 with 10 units of DsB for 3s
@use.in 0s
mix4 = mix mix3 with 10 units of DsB for 3s
@use.in 0s
mix5 = mix mix4 with 10 units of DsB for 3s
@use.in 0s
mix6 = mix mix5 with 10 units of DsR for 3s
det1 = detect sensor on mix6 for 30s
dispose mix6

// path 2
@use.in 0s
mix7 = mix slt1[1] with 10 units of DsB for 3s
@use.in 0s
mix8 = mix mix7 with 10 units of DsB for 3s
@use.in 0s
mix9 = mix mix8 with 10 units of DsB for 3s
@use.in 0s
```

```
mix10 = mix mix9 with 10 units of DsB for 3s
@use.in 0s
mix11 = mix mix10 with 10 units of DsR for 3s
det2 = detect sensor on mix11 for 30s
dispose mix11
```

Listing B.21: ProteinSplit2 dilution assay, adapted from [221].

```
module sensor
manifest DsS //sample
manifest DsB //buffer
manifest DsR //reagent

instructions:
@use.in 0s
mix1 = mix 10 units of DsS with 10 units of DsB for 3s
slt1 = split mix1 into 2

@use.in 0s
mix2 = mix slt1[0] with 10 units of DsB for 3s
slt2 = split mix2 into 2

@use.in 0s
mix3 = mix slt1[1] with 10 units of DsB for 3s
slt3 = split mix3 into 2

// path 1
@use.in 0s
mix4 = mix slt2[0] with 10 units of DsB for 3s
@use.in 0s
mix5 = mix mix4 with 10 units of DsB for 3s
@use.in 0s
mix6 = mix mix5 with 10 units of DsB for 3s
@use.in 0s
mix7 = mix mix6 with 10 units of DsB for 3s
@use.in 0s
mix8 = mix mix7 with 10 units of DsR for 3s
det1 = detect sensor on mix8 for 30s
dispose mix8

// path 2
@use.in 0s
mix9 = mix slt2[1] with 10 units of DsB for 3s
@use.in 0s
mix10 = mix mix9 with 10 units of DsB for 3s
@use.in 0s
mix11 = mix mix10 with 10 units of DsB for 3s
@use.in 0s
mix12 = mix mix11 with 10 units of DsB for 3s
@use.in 0s
mix13 = mix mix12 with 10 units of DsR for 3s
det2 = detect sensor on mix13 for 30s
dispose mix13

// path 3
@use.in 0s
mix14 = mix slt3[0] with 10 units of DsB for 3s
@use.in 0s
mix15 = mix mix14 with 10 units of DsB for 3s
```

```
@use.in 0s
mix16 = mix mix15 with 10 units of DsB for 3s
@use.in 0s
mix17 = mix mix16 with 10 units of DsB for 3s
@use.in 0s
mix18 = mix mix17 with 10 units of DsR for 3s
det3 = detect sensor on mix18 for 30s
dispose mix18

// path 4
@use.in 0s
mix19 = mix slt3[1] with 10 units of DsB for 3s
@use.in 0s
mix20 = mix mix19 with 10 units of DsB for 3s
@use.in 0s
mix21 = mix mix20 with 10 units of DsB for 3s
@use.in 0s
mix22 = mix mix21 with 10 units of DsB for 3s
@use.in 0s
mix23 = mix mix22 with 10 units of DsR for 3s
det4 = detect sensor on mix23 for 30s
dispose mix23
```

Listing B.22: Python script for generating protein dilution benchmarks.

```
import math
# num = 2^exp (eg. proteinsplit 2 gets exp = 2 for 4 dilution samples)
# this gives us a concentration factor diluting the protein sample
exp = 6
num = 2 ** exp
usein = True
n = 0

with open("output/ProteinSplit_{}.bs".format(exp), mode='w') as file:
    file.write("\nmodule sensor")
    file.write("\nmanifest DsS //sample")
    file.write("\nmanifest DsB //buffer")
    file.write("\nmanifest DsR //reagent\n")

    file.write("\ninstructions:")
    if usein:
        file.write("\n@use.in {}s".format(n))
    file.write("\nmix1 = mix 10 units of DsS with 10 units of DsB for 3s
        ")
    file.write("\nslt1 = split mix1 into 2")

    for i in range(2, num):
        file.write("\n")
        if usein:
            file.write("\n@use.in {}s".format(n))
        file.write("\nmix{} = mix slt{}[{}] with 10 units of DsB for 3s"
            .format(i, math.floor(i / 2), 0 if i % 2 == 0 else 1))
        file.write("\nslt{} = split mix{} into 2".format(i, i))

    for i in range(0, num):
        j = num+i*5
        file.write("\n\n// path {}".format(i+1))
        if usein:
            file.write("\n@use.in {}s".format(n))
```

```
33          file.write("\nmix{} = mix slt{}[{}] with 10 units of DsB for 3s"
                .format(j, math.floor((num+i)/2), 0 if j % 2 == 0 else 1))
34          if usein:
35              file.write("\n@use.in {}s".format(n))
36
37          file.write("\nmix{} = mix mix{} with 10 units of DsB for 3s"
                .format(j+1, j))
38          if usein:
39              file.write("\n@use.in {}s".format(n))
40
41          file.write("\nmix{} = mix mix{} with 10 units of DsB for 3s"
                .format(j+2, j+1))
42          if usein:
43              file.write("\n@use.in {}s".format(n))
44
45          file.write("\nmix{} = mix mix{} with 10 units of DsB for 3s"
                .format(j+3, j+2))
46          if usein:
47              file.write("\n@use.in {}s".format(n))
48
49          file.write("\nmix{} = mix mix{} with 10 units of DsR for 3s"
                .format(j+4, j+3))
50          file.write("\ndet{} = detect sensor on mix{} for 30s".format(i+1,
                 j+4))
51          file.write("\ndispose mix{}".format(j+4))
52      file.write("\n")
```

B.2.2 Mixed

The benchmarks listed as "Mixed" in Table 3.5 combine the various timing constraints introduced in Chapter 3. These were derived for the express purpose of stressing the scheduler, and do not correspond to any meaningful biochemical reactions.

Listing B.23: all_six

```
1  module sensor
2  manifest a
3  manifest b
4  manifest c
5  manifest d
6  manifest e
7
8  instructions:
9
10 @use.in 30s
11 ab = mix 1 units of a with 1 units of b for 15s
12 @finish.at 37s
13 cd = mix 1 units of c with 1 units of d for 10s
14
15 t_e = dispense 2 units of e
16 temp_e = split t_e into 2
17
18 @finish.in 15s
```

```
heat temp_e[0] at 30c for 15s

@finish.after 10s
heat cd at 90c for 35s

@use.at 5s
cde = mix cd with temp_e[0] for 5s

@use.after 5s
abcde = mix ab with cde for 10s

abcde = mix abcde with temp_e[1] for 5s

result = detect sensor on abcde for 5s

dispose abcde
```

Listing B.24: all_six_2

```
module sensor

manifest a
manifest b
manifest c
manifest d
manifest e

instructions:
temp_a = dispense 10 units of a

// droplet given time to cool before mixing with b
@use.after 60s
heat temp_a at 80c for 30s  //edge to 22

// droplet must be measured exactly 25s after mixing
@use.at 25s
cd = mix 10 units of c with 10 units of d for 10s //edge to 24

//volatile reaction needs to be mixed with cd within 30s
@finish.in 30s
ab = mix temp_a with 10 units of b for 10s  // edge to 28

x = detect sensor on cd for 15s // edge to 28

// want to start measuring changes after adding ab within 10s
@use.in 10s
abcd = mix ab with cd for 10s // edge to 30

y = detect sensor on abcd for 15s  //edge to 34

// final mixture needs to sit after mixing before detection
@finish.after 60s
abcde = mix abcd with 10 units of e for 10s //edge to 36

z = detect sensor on abcde for 10s  //edge to 40

// after cooling mixture, must collect sample at precise time
@finish.at 10s
heat abcde at 10c for 30s
```

```
41
42 dispose abcde
```

Listing B.25: all_eq

```
1  manifest a
2  manifest b
3  manifest c
4  manifest d
5  manifest e
6  manifest f
7
8  instructions:
9
10 @finish.at 10s
11 tab = mix 2 units of a with 2 units of b
12 ab = split tab into 4
13
14 @use.at 0s
15 heat ab[0] at 10c for 10s
16
17 @use.at 2s
18 abc = mix ab[1] with 1 units of c for 10s
19
20 @finish.at 5s
21 abc = mix abc with ab[0] for 10s
22
23 @use.at 10s
24 abd = mix ab[2] with 1 units of d for 4s
25
26 @finish.at 14s
27 abe = mix ab[3] with 1 units of e for 10s
28
29 tf = dispense 1 units of f
30
31 @use.at 20s
32 heat tf at 10c for 10s
33
34 @finish.at 10s
35 abcd = mix abc with abd for 5s
36
37 @use.at 10s
38 abef = mix abe with tf for 10s
39
40 abcdef = mix abcd with abef for 5s
41
42 dispose abcdef
```

Listing B.26: all_finish

```
1 manifest a
2 manifest b
3 manifest c
4 manifest d
5
6 instructions:
7
8 @finish.in 30s
```

```
ab = mix 1 units of a with 1 units of b for 10s

tc = dispense 1 units of c
@finish.at 15s
heat tc at 10c for 5s

@finish.after 15s
cd = mix tc with 1 units of d for 13s

abcd = mix ab with cd for 13s
dispose abcd
```

Listing B.27: all_start

```
manifest a
manifest b
manifest c
manifest d

instructions:

@use.in 30s
ab = mix 1 units of a with 1 units of b for 10s

tc = dispense 1 units of c
@use.at 15s
heat tc at 10c for 5s

@use.after 15s
cd = mix tc with 1 units of d for 13s

abcd = mix ab with cd for 13s
dispose abcd
```

Listing B.28: infeasible

```
manifest a
manifest b

instructions:

@finish.in 5s
ab = mix 1 units of a with 1 units of b

heat ab at 10c for 6s

dispose ab
```

B.3 Benchmarks for Chapter 5

As we describe all benchmarks, clearly code is available for each of these; however, when compiling, we first compile all function specifications and save their library representation, then load the library when compiling the main function (in *BioScript*, these are operations following the `instructions:` tokens).

Listing B.29: Synth1

```
// imports
import bar from my_lib
/* bar is defined as:
 * function bar(C, D) {
 *     ab = mix C with D for 2s
 *     x = detect weight on ab for 1s
 *     if (x <= 0.5) {
 *         heat ab at 10c for 1s
 *     }
 *     return ab
 * }
 */

// substances
module weight
manifest A
manifest B

functions:
function foo(x) {
    b = dispense B
    heat x at 10c for 2s
    heat b at 10c for 2s
    Y = bar(x, b)
    heat Y at 10c for 1s
    Y = split Y into 2
    drain Y[0]
    return Y[1]
}

instructions:
a = dispense A
bb = dispense B
heat a at 10c for 1s
heat bb at 10c for 1s // live across call to foo
a = foo(a)
if (9 < 10) {
    b = dispense B
    a = mix a with b for 2s
}
heat a at 10c for 1s
heat bb at 10c for 1s
dispose a
```

```
dispose bb
```

Listing B.30: PCR

```
// Basic pcr assay with thermocycling
import * from pcr
/*
pcr contains:
function thermocycle(sample) {
    repeat 25 times {
        heat sample at 95c for 3m
        heat sample at 53c for 30s
        heat sample at 72c for 20s
    }
    return sample
}
and
function pcr(sample, primers) {
    master_mix = dispense pcr_master_mix
    sample = mix sample with master_mix
    heat sample at 95c for 1m
    sample = mix sample with primers

    sample = thermocycle(sample)

    heat sample at 53c for 20s

    return sample
}
*/

module fluor
module volume
manifest pcr_master_mix
manifest template
manifest primers

instructions:

sample = dispense template
prim = dispense primers
result = pcr(sample, prim)

x = detect fluor on result

drain result
```

Listing B.31: DRPCR

```
// Basic pcr assay with thermocycling
import * from pcr
/*
    pcr contains:
function thermocycle_drop_replace(sample) {
    repeat 25 times {
        heat sample at 95c for 3m
        x = detect volume on sample
        if (x <= 1) {
```

```
            repl = dispense pcr_master_mix // water could be used, but
                not enough reservoirs for opendrop
            sample = mix sample with repl
        }
        heat sample at 53c for 30s
        heat sample at 72c for 20s
    }
    return sample
}
    and
function pcr_drop_replace(sample, primers) {
    master_mix = dispense pcr_master_mix
    sample = mix sample with master_mix
    heat sample at 95c for 1m
    sample = mix sample with primers

    sample = thermocycle_drop_replace(sample)

    heat sample at 53c for 20s

    return sample
}
*/

module fluor
module volume
manifest pcr_master_mix
manifest template
manifest primers

instructions:

sample = dispense template
prim = dispense primers
result = pcr_drop_replace(sample, prim)

x = detect fluor on result

drain result
```

Listing B.32: SynthTail

```
// imports
import tail from my_lib
/* tail is defines as:
 * function tail(a) {
 *   heat a at 10c for 1s
 *   x = detect sensor on a
 *   if (x <= 0.5) {
 *       a = tail(a)
 *   }
 *   return a
 * }
 */

module sensor
manifest A

instructions:
```

```
a = dispense A
a = tail(a)
    drain a
```

Listing B.33: SynthHead

```
import head from my_lib
/* head is defined as
 * function head(a) {
 *    heat a at 10c
 *    x = detect sensor on a
 *    if (x <= 0.5) {
 *        a2 = dispense
 *        a2 = foo(a2)
 *        a = mix a with a2
 *    }
 *    return a
 * }
 */

module sensor
manifest A

instructions:
a = dispense A
a = head(a)
drain a
```

Listing B.34: ProteinSplit, adapted from [221].

```
/*
 * Function recursively builds split trees for diluting samples
 */
module sensor
manifest DsS
manifest DsB
manifest DsR

functions:
function dilute_and_detect(sample) {
    // as sample is array of 2 droplets, could use SIMD instructions
    repeat 4 times {
        buffer[2] = dispense 1 units of DsB
        sample[0] = mix sample[0] with buffer[0] for 3s
        sample[1] = mix sample[1] with buffer[1] for 3s
    }
    reagent[2] = dispense 1 units of DsR
    // example SIMD mix
    final = mix sample with reagent for 3s
    // SIMD detect
    result = detect sensor on final for 30s
    drain final
    return result
}

function split_recurse(exp, samples) {
    if (exp == 1) {
        return dilute_and_detect(samples)
```

```
29      }
30
31      // sample is array of 2 droplets, we access each in turn for
            recursive calls
32      buffer = dispense DsB
33      pre_dilute = mix samples[0] with buffer for 3s
34      pre_dilutes = split pre_dilute into 2
35      res1 = split_recurse(exp-1, pre_dilutes)
36
37      buffer = dispense DsB
38      pre_dilute = mix samples[1] with buffer for 3s
39      pre_dilutes = split pre_dilute into 2
40      res2 = split_recurse(exp-1, pre_dilutes)
41
42      return res1
43  }
44
45  function protein_split(exp) {
46      sample = dispense 2 units of DsS
47      buffer = dispense 2 units of DsB
48      mixture = mix sample with buffer for 3s
49      paths = split mixture into 2
50      return split_recurse(exp, paths)
51  }
52
53  instructions:
54
55  // example calls
56  dilution_factor_1 = protein_split(1)
57  dilution_factor_2 = protein_split(2)
```

B.4 OpenDrop Demos

Videos demoing execution of compiled BioScript programs on OpenDrop are available here.

Appendix C

Pseudocode for Relative Interval Scheduling

Relative Interval Scheduling (Section 3.4.1) is summarized in Algorithm 3. While the procedure is relatively straightforward, partial pseudocode is provided for parsing time constraints in phase 1 (Algorithm 5), for expanding storage windows in phase 2 (Algorithm 6), and for retrieving violations on the forest (Algorithm 4).

Algorithm 3 Relative Interval Scheduling

```
1: procedure SCHEDULE(Architecture A, DAG G)
2:     ParseTimeConstraints(G)                          ▷ aborts if infeasible
3:     forest ← CreateForest(G)
4:     violations ← GetViolations(forest, A)            ▷ fails if not enough modules
5:     while violations ≠ ∅ do
6:         ExpandStorageWindow(G, violations)           ▷ fails if no progress
7:         forest ← CreateForest(G)
8:         violations ← GetViolations(forest, A)
9:     return Ω(G)
```

Algorithm 4 Discovering Resource Violations in a Relative Interval Forest

```
1: function GetViolations(Set of Relative Interval Forests F, Architecture A)
2:     violations ← ∅
3:     maxHeats ← number of heats in A
4:     maxDetects ← number of sensors in A
5:     maxModules ← number of modules in A
6:     maxInputsOf ← number of inputs of each type in A
7:     max ← (heats : maxHeats), (detects : maxDetects), ...
8:     for time from lowest(F) to highest(F) do
9:         for type in [heats, detects, modules, inputs] do
10:            if F.overlaps(time, type) ≥ max[type] then
11:                if type is modules then
12:                    Fail(R)                          ▷ cannot overcome this
13:                duration ← duration of violation
14:                violations.insert([type, duration, set of overlapping ops])
15:    return violations
```

Algorithm 5 Parsing Time Constraints (Phase 1)

```
1: procedure ParseTimeConstraints(DAG G)
2:     for child ∈ reverse(G) do
3:         for parent ∈ parents(child) do
4:             if isConstrained(parent) then
5:                 clat ← child.latency
6:                 dur ← parent.constrDuration
7:                 newStore ← StorageNode
8:                 switch parent.constrType do
9:                     case SGE
10:                        newStore.window ← Window(+inf)
11:                    case SEQ
12:                        newStore.latency ← dur
13:                        G.insertStoreBetween(newStore, parent, child)
14:                        break
15:                    case SLE
16:                        parent.window ← Window(dur)
17:                        break
18:                    case FGE
19:                        newStore.window ← Window(+inf)
20:                    case FEQ
21:                        if dur - clat < 0 then
22:                            Abort(infeasible)
23:                        newStore.latency ← dur - clat
24:                        G.insertStoreBetween(newStore, parent, child)
25:                        break
26:                    case FLE
27:                        if dur - clat < 0 then
28:                            Abort(infeasible)
29:                        parent.window ← Window(dur - clat)
30:                        break
31:            else
32:                parent.window ← Window(+inf)
33:    EqualizePaths(G)                                 ▷ ommitted
```

Algorithm 6 Satisfy a Resource Violation

```
1: procedure ExpandStorageWindow(DAG G, Set of Violations V)
2:     for violation in V do
3:         dur ← violation.duration
4:         for pair of ops u, v in violation.overlappingOps do
5:             switch path relationship between path_u and path_v do
6:                 case path_u and path_v converge at w
7:                     if Storage windows exist between u and w with combined durations ≥ dur
   then
8:                         expanded ← 0
9:                         for window ω between u and w do
10:                            if expanded = dur then
11:                                break
12:                            expand ω by Δ = min{dur − expanded, ω.size}
13:                            expanded ← expanded + Δ
14:                    return
15:                case path_u and path_v diverge from w
16:                    ...
17:                case path_u and path_v diverge from y and converge at z
18:                    ...
19:                case path_u and path_v neither converge nor diverge
20:                    ...
21:    Fail(—V—)                                          ▷ did not make any progress
```

Appendix D

Overview of Peptide Synthesis

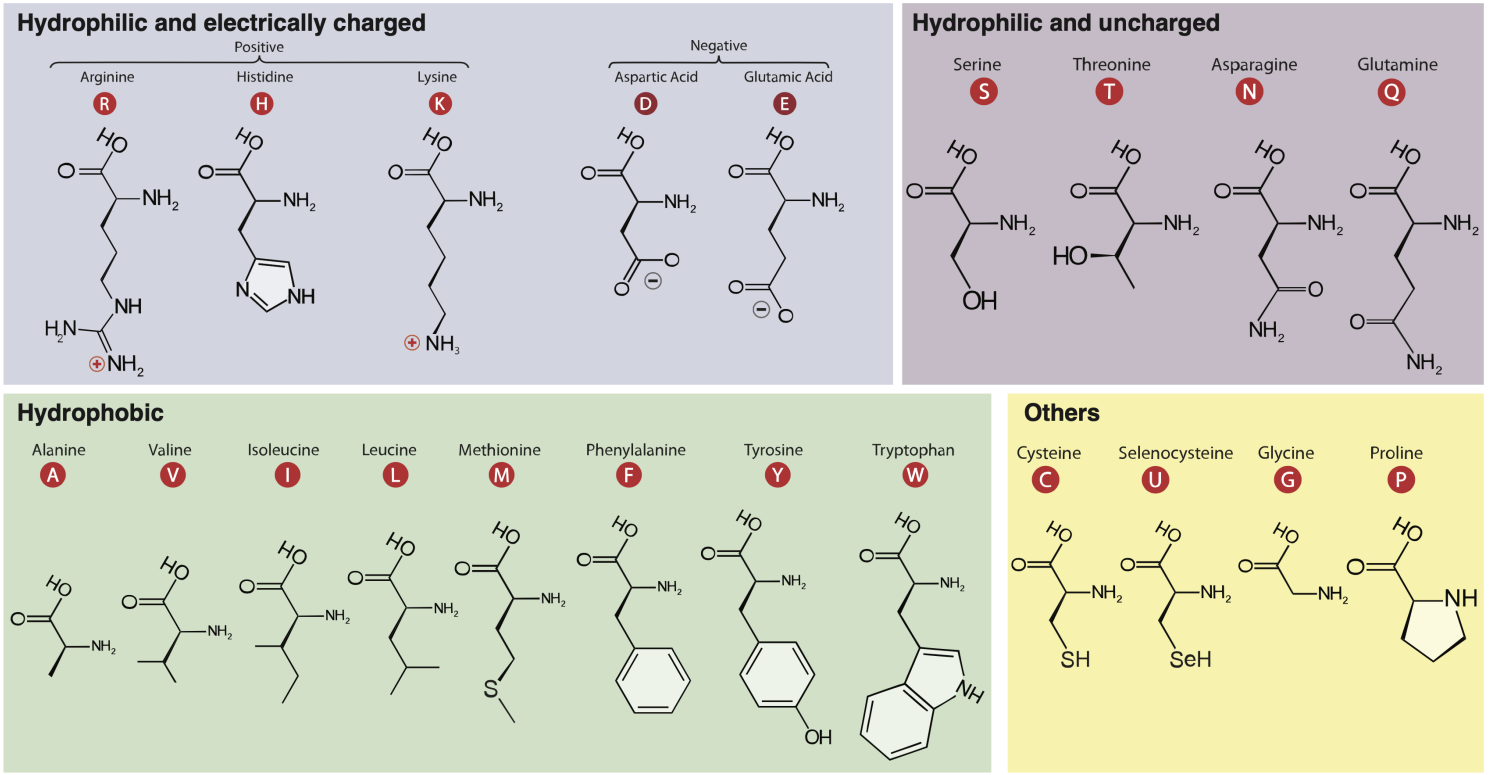
PepSyn, a proof-of-concept implementation of the *MediSyn* framework presented in Chapter 10, generates candidate antimicrobial peptides. Here, we provide additional detail to what peptides are and the process for chemically synthesizing them — a step that would be necessary to fully-automate the discovery process. A *peptide* is a short linear chain of bonded amino acids. As opposed to larger chemical structures (e.g., proteins, small molecules, or macromolecules), peptides are relatively simple, featuring up to around 50 amino acids, and lacking a stable 3D structure [48]. The synthesis of peptides involves chemically coupling amino acids sequentially through peptide bonds. As shown in Fig. D.1b, an amino acid is composed of a carboxyl group (-COOH), amino group (-NH_2), and a unique *side chain* (R group) between. A *protecting group* (PG) is typically added to the amino group of amino-acids to prevent unintended chemical reactions; as such, a protected amino acid must be *deprotected* during synthesis to allow coupling. Coupling of amino acids is achieved via peptide bonds, which occurs when the carboxyl group of one amino acid is coupled with the amino group of another through a

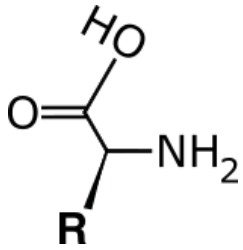
condensation reaction, leaving a byproduct of a water molecule. Figure D.1a shows the chemical structures of the 21 amino acids that naturally occur in humans, with expanded forms of the side chains.

Solid-phase peptide synthesis is the *de facto* standard for producing synthetic peptides [34, 19]. Figure D.2 depicts the process: a solid resin bead support forms a substrate onto which protected amino acids are sequentially deprotected and then coupled through peptide bonds; in order to facilitate coupling, protected amino acids are first deprotected to expose the amino groups; as each new amino acid is coupled to the nascent peptide, condensed water is rinsed away; when complete, a cleavage reagent separates the peptide from the resin bead.

Certain properties of the growing peptide (e.g., hydrophobicity and amino acid composition) can negatively affect successful synthesis, leading to truncation, deletion, or other anomalies in the end result [159]. For this reason, Section 10.2 implements a mechanism to enable *PepSyn* to filter candidates from the search that are likely to fail during synthesis.

[1] "Molecular structures of the 21 proteinogenic amino acids" by Dan Cojocari is licensed under CC BY-SA 3.0.





(a) (b)

Figure D.1: Left: Chemical structures and single-letter abbreviations for the 21 proteinogenic amino acids found in the genetic code for eukaryotic organisms (including humans).[1] Right: A prototypical amino-acid (top) and the same amino-acid with a *protection group* (PG) guarding its amino-group. These skeletal structures omit underlying carbon atoms (where line segments connect); the carboxyl group (-COOH) is toward the top of each depiction, while the side chain (R-group) is depicted with a wedged connection.

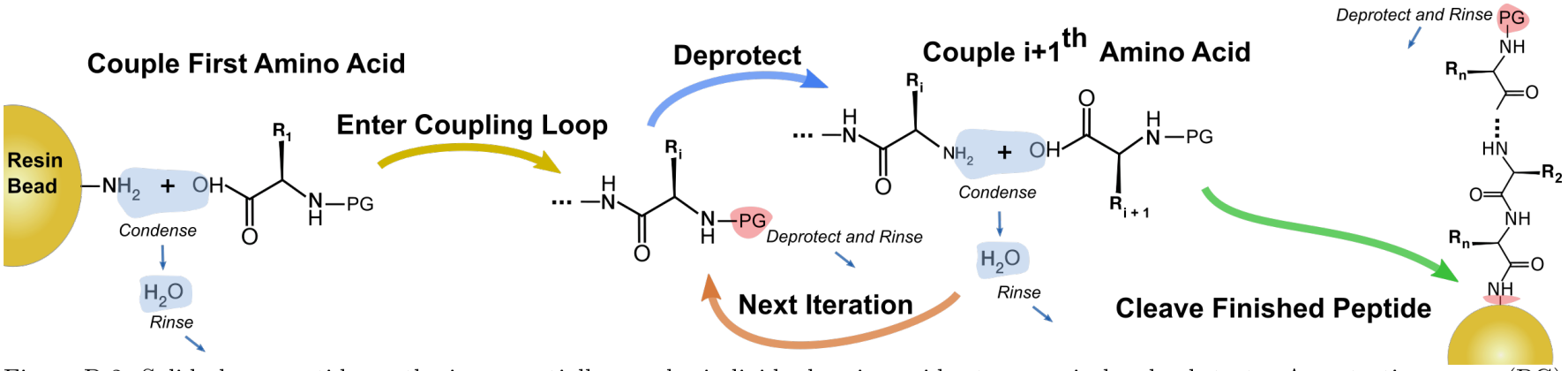


Figure D.2: Solid-phase peptide synthesis sequentially couples individual amino acids atop a resin bead substrate. A protecting group (PG) must be removed between each coupling, and a final cleaving reagent removes the finished peptide from the resin bead.

Appendix E

$BioVec^{(k)}$

An expanded continuous distributed representation for biological sequences

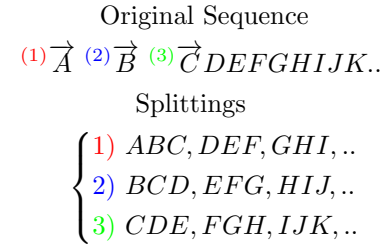


Figure E.1: ProtVec's approach to representing a single peptide is to decompose the peptide into 3 "splittings" made up of the original peptide's overlapping 3-mers [11].

The original $ProtVec$ model described by [11] is a specific instance of a so-called generalized $BioVec$ model for biological sequence data tailored for peptides. [11]'s $ProtVec$ technique decomposed each peptide in their training corpus as 3 "sentences" composed of overlapping 3-mers as illustrated in Fig. E.1. Figure E.2 depicts how our $ProtVec^{(3)}$ model used for training the clustering model Ψ expands the pre-processing step: after

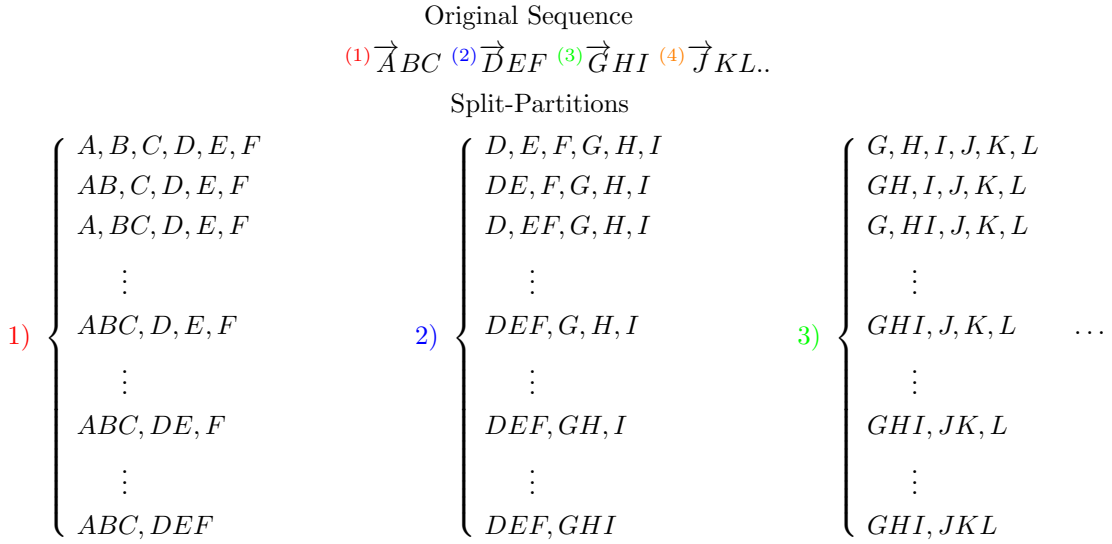


Figure E.2: A $ProtVec^{(3)}$ model, in addition to a generalization of "splittings" used in [11]'s $ProtVec$ model, represents each peptide by splitting it into segments of (up to) length $2 * 3 = 6$, where each segment begins at a σ_i with $i \pmod 3 = 0$, and decomposes each segment into all possible (in order) partitions of 1-, 2-, and 3-mers.

splitting peptides into segments of (up to) length $6 = 3 * 2$, each segment is decomposed into biological "sentences" using all partitions of 1-, 2-, and 3-mers from the segment.[1] For the $PepGen$ front-end module to $PepSyn$, the trained model provides proximity relations between the vocabulary of all constituent 1-, 2-, and 3-mers in the peptide corpus $\mathcal{D}$; whereas [11]'s model only provides proximity between 3-mers.

We describe a generalization for continuous distributed representations of biological sequence data consisting of biological words of up to length $k \in \mathbb{N}_1$ we call $BioVec^{(k)}$. In general, a $BioVec^{(k)}$ model pre-processes its input in a manner similar to the methods shown in Figs. E.1 and E.2; given an integer k, input instances are decomposed into: a single sentence of all 1-mers, 2 sentences of overlapping 2-mers, ..., and k sentences of overlapping k-mers. Additionally, we decompose each input into segments of (up to) length $2 * k$, where each segment begins at a σ_i such that $i \pmod k = 0$. Segmented

[1] Note: Fig. E.2 omits the depiction where we still include 3 sequences of overlapping 3-mers from the original sequence as in Fig. E.1, in addition to overlapping 1-, and 2-mers.

regions are partitioned into all (in order) combinations of 1-, 2, ..., and k-mers within the segment. Without loss of generality, the overlap between two ordered segments $\mathbf{s}_1$ and $\mathbf{s}_2$ ensures that the last k-mer in $\mathbf{s}_1$ is the first k-mer in $\mathbf{s}_2$, so that the generated training sentences of all partitions of 1-, 2-, ..., and k-mers from the original sequence are represented by windows of (up to) k words.

E.1 Number of sentences a $BioVec^{(k)}$ model processes from a single input instance

Given an input $\mathbf{s} = \sigma_1\sigma_2 \ldots \sigma_{|\mathbf{s}|}$ of length $|\mathbf{s}|$, the pre-processing step generates:

$$\sum_{i=1}^{k} i \tag{E.1}$$

sentences directly from the original input, using overlapping i-mers in a manner similar to [11] (see Fig. E.1 for 3-mers). We say a sentence is composed of at least 2 words, restricting $k \leq \frac{|\mathbf{s}|+1}{3}$ for a given input $\mathbf{s}$.

Additionally, $\mathbf{s}$ is split into n segments $(\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_n)$, where

$$n = \begin{cases} |\mathbf{s}|/k - 1 & \text{if } |\mathbf{s}| \pmod 3 = 0 \\ \lfloor |\mathbf{s}|/k \rfloor & \text{otherwise} \end{cases} \tag{E.2}$$

and — when $|\mathbf{s}| \pmod k = 0$ — the length of each segment is $2 * k$, or — when $|\mathbf{s}| \pmod k \neq 0$ — the length of the first $n-1$ segments is $2 * k$, and the length of the n^{th} segment is $k + |\mathbf{s}| \pmod k$.

For each segment $\mathbf{s}_i$ of length $j = |\mathbf{s}_i|$, we generate sentences using all in-order partitions of 1-, 2-, ..., and k-mers from $\mathbf{s}_i$. The number of sentences $count(\mathbf{s}_i)$ generated for $\mathbf{s}_i$ corresponds to $F_j^{(k)}$, the generalized form of the j^{th} Fibonacci term of *order* k (that is, $count(\mathbf{s}_i) \equiv F_j^{(k)}$), whereby the j^{th} term is the sum of the previous k terms [69]. [55]

provides a simplified formula for finding the j^{th} term of a Fibonacci sequence of order k:

$$F_j^{(k)} = \sum_{i=1}^{k} \frac{\alpha_i - 1}{2 + (k+1)(\alpha_i - 2)} \alpha_i^j \tag{E.3}$$

where α_i are the roots of $x^k - x^{k-1} - \cdots - 1 = 0$.[2] [216] gives the roots for $1 \leq k \leq 10$.

Then, the total number of sentences generated for $\mathbf{s}$ is

$$count(\mathbf{s}) = \sum_{i=1}^{k} i \quad + \sum_{j=|\mathbf{s}_i|, 1 \leq i \leq n} F_j^{(k)} \tag{E.4}$$

E.2 Number of sentences in the training corpus for *PepGen*'s $ProtVec^{(3)}$ model

Our $ProtVec^{(3)}$ model produces sentences from a given sequence that follows the so-called *tribonacci* terms, the Fibonacci terms of order 3; for the j^{th} term, using [55]'s notation and simplifying, we have:

$$F_j^{(3)} = \sum_{i=1}^{3} \frac{\alpha_i - 1}{4\alpha_i - 6} \alpha_i^j \tag{E.5}$$

where

$$\alpha_1 = \frac{1}{3}\left(\sqrt[3]{19 + 3\sqrt{33}} + \sqrt[3]{19 - 3\sqrt{33}} + 1\right)$$

$$\alpha_2 = \frac{1}{6}\left(2 - \sqrt[3]{19 + 3\sqrt{33}} - \sqrt[3]{19 - 3\sqrt{33}} + \sqrt{3}\left(\sqrt[3]{19 + 3\sqrt{33}} - \sqrt[3]{19 - 3\sqrt{33}}\right) i\right), \text{and}$$

$\alpha_3 = \overline{\alpha_2}$ (i.e., the complex conjugate of α_2)

are the roots of $x^2 - x - 1 = 0$ given by [215]. Then, for our training corpus of $\mathcal{C} = \{\mathbf{s}^1, \ldots, \mathbf{s}^{14,271}\}$ primary peptide sequences, the total number of sentences generated for training our $ProtVec^{(3)}$ model is

[2] [55] defines $F_1^k = 1$, where we start with $F_1^k = 0$ and $F_2^k = 1$; hence, the exponential term in Eq. (E.3) is to j, rather than [55]'s $j - 1$.

$$\sum_{\mathbf{s}\in\mathcal{C}} count(\mathbf{s}) \tag{E.6}$$

which in practice amounted to $6,476,896$ unique sentences made up of a vocabulary of $8,986$ unique 1-, 2-, and 3-mers.

Bibliography

[1] Mirela Alistar and Urs Gaudenz. Opendrop: An integrated do-it-yourself platform for personal use of biochips. *Bioengineering*, 4(2):45, 2017.

[2] Mirela Alistar and Paul Pop. Synthesis of biochemical applications on digital microfluidic biochips with operation execution time variability. *Integration*, 51:158–168, 2015.

[3] Mirela Alistar, Paul Pop, and Jan Madsen. Synthesis of application-specific fault-tolerant digital microfluidic biochip architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 35(5):764–777, 2016.

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, Portland, OR, USA, 2013. IEEE.

[5] Ahmed M. Amin, Raviraj Thakur, Seth Madren, Han-Sheng Chuang, Mithuna Thottethodi, T. N. Vijaykumar, Steven T. Wereley, and Stephen C. Jacobson. Software-programmable continuous-flow multi-purpose lab-on-a-chip. *Microfluid Nanofluidics*, 15(5):647–659, Nov 2013.

[6] Ahmed M. Amin, Mithuna Thottethodi, T. N. Vijaykumar, Steven Wereley, and Stephen C. Jacobson. Aquacore: a programmable architecture for microfluidics. In Dean M. Tullsen and Brad Calder, editors, *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 254–265, New York, NY, USA, 2007. ACM.

[7] Scott C. Ananian and Arthur C. Smith. *The Static Single Information Form*. PhD thesis, Massachusetts Institue of Technology, 1999.

[8] Vaishnavi Ananthanarayanan and William Thies. Biocoder: A programming language for standardizing and automating biology protocols. *Journal of Biological Engineering*, 4, NOV 2010.

[9] Christian B Anfinsen. Principles that govern the folding of protein chains. *Science*, 181(4096):223–230, 1973.

[10] Christian B Anfinsen, Edgar Haber, Michael Sela, and FH White Jr. The kinetics of formation of native ribonuclease during oxidation of the reduced polypeptide chain. *Proceedings of the National Academy of Sciences of the United States of America*, 47(9):1309, 1961.

[11] Ehsaneddin Asgari and Mohammad RK Mofrad. Continuous distributed representation of biological sequences for deep proteomics and genomics. *PloS one*, 10(11):e0141287, 2015.

[12] Ronald C Backer, Joseph R Monforte, and Alphonse Poklis. Evaluation of the dri® oxycodone immunoassay for the detection of oxycodone in urine. *Journal of analytical toxicology*, 29(7):675–677, 2005.

[13] James K Baker. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*, 65(S1):S132–S132, 1979.

[14] Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature News*, 533(7604):452, 2016.

[15] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[16] Amar S. Basu. Droplet morphometry and velocimetry (dmv): a video processing software for time-resolved, label-free tracking of droplet parameters. *Lab Chip*, 13:1892–1901, 2013.

[17] Cesar VF Batista, Andrea Scaloni, Daniel J Rigden, Lindomar R Silva, Adela Rodrigues Romero, Rina Dukor, Antonio Sebben, Fabio Talamo, and Carlos Bloch. A novel heterodimeric antimicrobial peptide from the tree-frog phyllomedusa distincta. *FEBS letters*, 494(1-2):85–89, 2001.

[18] Kia Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design & Test of Computers*, 17:68–83, 2000.

[19] Raymond Behrendt, Peter White, and John Offer. Advances in fmoc solid-phase peptide synthesis. *Journal of Peptide Science*, 22(1):4–27, 2016.

[20] Biddut Bhattacharjee and Homayoun Najjaran. Droplet sensing by measuring the capacitance between coplanar electrodes in a digital microfluidic system. *Lab Chip*, 12:4416–4423, 2012.

[21] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York*

City, NY, USA, June 19-24, 2016, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2933–2942, New York City, NY, USA, 2016. JMLR.org.

[22] H Bielka, N Sharon, and E. C. Webb. Nomenclature and symbolism for amino acids and peptides. *Pure and Applied Chemistry*, 56:617–619, 1984.

[23] Robert Bogue. Robots in the laboratory: a review of applications. *Industrial Robot: An International Journal*, 2012.

[24] Karl-Friedrich Böhringer. Modeling and controlling parallel tasks in droplet-based microfluidic systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(2):334–344, 2006.

[25] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. SSI properties revisited. *ACM Trans. Embedded Comput. Syst.*, 11(S1):21, 2012.

[26] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[27] Brian L Bray. Large-scale manufacture of peptide therapeutics by chemical synthesis. *Nature Reviews Drug Discovery*, 2(7):587–593, 2003.

[28] Preston Briggs, Keith D Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

[29] James R Broach and Jeremy Thorner. High-throughput screening for drug discovery. *Nature*, 384(6604 Suppl):14–16, 1996.

[30] CDC. Leading causes of death, 2022.

[31] CDC. Select agents and toxins list. federal select agent program, 2022.

[32] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In John R. White and Frances E. Allen, editors, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Massachusetts, USA, June 23-25, 1982*, pages 98–105, Boston, MA, 1982. ACM.

[33] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, 1981.

[34] WCWPD Chan and Peter White. *Fmoc solid phase peptide synthesis: a practical approach*, volume 222. OUP Oxford, 1999.

[35] Erika Check Hayden. The automated lab. *Nature*, 516(7529):131–132, 2014.

[36] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. The rise of deep learning in drug discovery. *Drug discovery today*, 23(6):1241–1250, 2018.

[37] Ying-Han Chen, Chung-Lun Hsu, Li-Chen Tsai, Tsung-Wei Huang, and Tsung-Yi Ho. A reliability-oriented placement algorithm for reconfigurable digital microfluidic biochips using 3-d deferred decision making technique. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(8):1151–1162, 2013.

[38] Minsik Cho and David Z. Pan. A high-performance droplet routing algorithm for digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(10):1714–1724, 2008.

[39] Peter Cooreman, Ronald Thoelen, Jean Manca, M. vandeVen, V. Vermeeren, L. Michiels, M. Ameloot, and P. Wagner. Impedimetric immunosensors based on the conjugated polymer ppv. *Biosens. Bioelectron.*, 20:2151–2156, 2005.

[40] Christopher Curtis and Philip Brisk. Simulation of feedback-driven pcr assays on a 2d electrowetting array using a domain-specific high-level biological programming language. *Microelectronic Engineering*, 148:110–116, 2015.

[41] Christopher Curtis, Daniel T. Grissom, and Philip Brisk. A compiler for cyber-physical digital microfluidic biochips. In Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O'Boyle, editors, *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pages 365–377, New York, NY, USA, 2018. ACM.

[42] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[43] Mauro Dalla Serra, Oscar Cirioni, Rosa Maria Vitale, Giovanni Renzone, Manuela Coraiola, Andrea Giacometti, Cristina Potrich, Elisa Baroni, Graziano Guella, Marina Sanseverino, et al. Structural features of distinctin affecting peptide biological and biochemical properties. *Biochemistry*, 47(30):7888–7899, 2008.

[44] Michael Davies, Rhys DO Jones, Ken Grime, Rasmus Jansson-Löfmark, Adrian J Fretland, Susanne Winiwarter, Paul Morgan, and Dermot F McGinnity. Improving the accuracy of predicted human pharmacokinetics: lessons learned from the astrazeneca drug pipeline over two decades. *Trends in pharmacological sciences*, 41(6):390–408, 2020.

[45] Margaret O Dayhoff, RV Eck, and CM Park. A model of evolutionary change in proteins. *Atlas of protein sequence and structure*, 5(88-99):88–99, 1972.

[46] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Trans. Evolutionary Computation*, 18(4):577–601, 2014.

[47] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

[48] Lei Diao and Bernd Meibohm. Pharmacokinetics and pharmacokinetic–pharmacodynamic correlations of therapeutic peptides. *Clinical pharmacokinetics*, 52(10):855–868, 2013.

[49] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

[50] Joseph A DiMasi, Henry G Grabowski, and Ronald W Hansen. Innovation in the pharmaceutical industry: new estimates of r&d costs. *Journal of health economics*, 47:20–33, 2016.

[51] Huijiang Ding, Saman Sadeghi, Gaurav J Shah, Supin Chen, Pei Yuin Keng, R Michael van Dam, et al. Accurate dispensing of volatile reagents on demand for chemical reactions in ewod chips. *Lab on a Chip*, 12(18):3331–3340, 2012.

[52] Jie Ding, Krishnendu Chakrabarty, and Richard B. Fair. Scheduling of microfluidic operations for reconfigurabletwo-dimensional electrowetting arrays. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(12):1463–1468, 2001.

[53] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 579–587, Lille, France, 2015. JMLR.org.

[54] Shan Dong and David B Searls. Gene structure prediction by linguistic methods. *Genomics*, 23(3):540–551, 1994.

[55] Gregory PB Dresden and Zhaohui Du. A simplified binet formula for k-generalized fibonacci numbers. *J. Integer Seq.*, 17(4):14–4, 2014.

[56] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics letters B*, 195(2):216–222, 1987.

[57] Witold Dyrka, Mateusz Pyzik, François Coste, and Hugo Talibart. Estimating probabilistic context-free grammars for proteins using contact map constraints. *PeerJ*, 7:e6559, 2019.

[58] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[59] Moe Elbadawi, Simon Gaisford, and Abdul W Basit. Advanced machine-learning techniques in drug discovery. *Drug Discovery Today*, 26(3):769–777, 2021.

[60] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1638–1645, Melbourne, Australia, 2017. ijcai.org.

[61] Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Unsupervised learning by program synthesis. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 973–981, Montreal, Quebec, Canada, 2015. MIT Press.

[62] RB Fair, V Srinivasan, H Ren, P Paik, VK Pamula, and MG Pollack. Electrowetting-based on-chip sample processing for integrated microfluidics. In *IEEE International Electron Devices Meeting 2003*, pages 32–5. IEEE, 2003.

[63] Luis M. Fidalgo and Sebastian J. Maerkl. A software-programmable microfluidic device for automated biology. *Lab Chip*, 11(9):1612–1619, May 2011.

[64] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, 1986.

[65] Ryan Fobel, Christian Fobel, and Aaron R. Wheeler. Dropbot: An open-source digital microfluidic control system with precise control of electrostatic driving force and instantaneous drop velocity measurement. *Applied Physics Letters*, 102(19), 2013.

[66] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. *ACM Sigplan Notices*, 51(1):802–815, 2016.

[67] Michal Galdzicki, Kevin P Clancy, Ernst Oberortner, Matthew Pocock, Jacqueline Y Quinn, Cesar A Rodriguez, Nicholas Roehner, Mandy L Wilson, Laura Adam, J Christopher Anderson, et al. The synthetic biology open language (sbol) provides a community standard for communicating designs in synthetic biology. *Nature biotechnology*, 32(6):545–550, 2014.

[68] Jie Gao, Xianming Liu, Tianlan Chen, Pui-In Mak, Yuguang Du, Mang-I Vai, Bingcheng Lin, and Rui P. Martins. An intelligent digital microfluidic system with fuzzy-enhanced feedback for multi-droplet manipulation. *Lab Chip*, 13:443–451, 2013.

[69] Martin Gardner. *The 2nd Scientific American book of mathematical puzzles & diversions*, page 101. University of Chicago Press, 1987.

[70] Lal George and Andrew W. Appel. Iterated register coalescing. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*, 18(3):208–218, 1996.

[71] Charles J Geyer and Elizabeth A Thompson. Annealing markov chain monte carlo with applications to ancestral inference. *Journal of the American Statistical Association*, 90(431):909–920, 1995.

[72] Georges G. E. Gielen, editor. *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6-10, 2006*. European Design and Automation Association, Leuven, Belgium, 2006.

[73] Walter R Gilks, Sylvia Richardson, and David Spiegelhalter. *Markov chain Monte Carlo in practice*. CRC press, 1995.

[74] Andrea Giuliani, Giovanna Pirri, and Silvia Nicoletto. Antimicrobial peptides: an overview of a promising class of therapeutics. *Open Life Sciences*, 2(1):1–33, 2007.

[75] Jian Gong and Chang-Jin Kim. Direct-referencing two-dimensional-array digital microfluidics using multilayer printed circuit board. *J. Microelectromech. Syst.*, 17:257–264, 2008.

[76] Jian Gong and CJ. Kim. All-electronic droplet generation on-chip with real-time feedback control for ewod digital microfluidics. *Lab Chip*, 8:898–906, 2008.

[77] Daniel Grissom and Philip Brisk. Fast online synthesis of generally programmable digital microfluidic biochips. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 413–422, New York, NY, USA, 2012. ACM.

[78] Daniel Grissom and Philip Brisk. Fast online synthesis of digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(3):356–369, 2014.

[79] Daniel Grissom, Christopher Curtis, Skyler Windh, Calvin Phung, Navin Kumar, Zachary Zimmerman, O'Neal Kenneth, Jeffrey McDaniel, Nick Liao, and Philip Brisk. An open-source compiler and pcb synthesis tool for digital microfluidic biochips. *Integration, the VLSI Journal*, 51:169–193, 2015.

[80] Daniel T. Grissom and Philip Brisk. Path scheduling on digital microfluidic biochips. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 26–35, New York, NY, USA, 2012. ACM.

[81] Daniel T. Grissom, Christopher Curtis, and Philip Brisk. Interpreting assays with control flow on digital microfluidic biochips. *JETC*, 10(3):24:1–24:30, 2014.

[82] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[83] Anvita Gupta, Alex T Müller, Berend JH Huisman, Jens A Fuchs, Petra Schneider, and Gisbert Schneider. Generative recurrent networks for de novo drug design. *Molecular informatics*, 37(1-2):1700111, 2018.

[84] Sudheer Gupta, Pallavi Kapoor, Kumardeep Chaudhary, Ankur Gautam, Rahul Kumar, Open Source Drug Discovery Consortium, and Gajendra PS Raghava. In silico approach for predicting toxicity of peptides and proteins. *PloS one*, 8(9):e73957, 2013.

[85] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38, Seattle, WA, USA, 2013. ACM.

[86] Ben. Hadwen, G. R. Broder, D. Morganti, A. Jacobs, C. Brown, J. R. Hector, Y. Kubota, and H. Morgan. Programmable large area digital microfluidic array with integrated droplet sensing for bioassays. *Lab Chip*, 12(18):3305–3313, Sep 2012.

[87] Robert EW Hancock and Hans-Georg Sahl. Antimicrobial and host-defense peptides as new anti-infective therapeutic strategies. *Nature biotechnology*, 24(12):1551–1557, 2006.

[88] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. *BIOMETRIKA*, 57(1):97–&, 1970.

[89] Mark Hewitt, Mark TD Cronin, Steven J Enoch, Judith C Madden, David W Roberts, and John C Dearden. In silico prediction of aqueous solubility: the solubility challenge. *Journal of chemical information and modeling*, 49(11):2572–2587, 2009.

[90] Alison H Holmes, Luke SP Moore, Arnfinn Sundsfjord, Martin Steinbakk, Sadie Regmi, Abhilasha Karkey, Philippe J Guerin, and Laura JV Piddock. Understanding the mechanisms and drivers of antimicrobial resistance. *The Lancet*, 387(10014):176–187, 2016.

[91] Peter Hornbeck. Enzyme-linked immunosorbent assays. *Current protocols in immunology*, pages 2–1, 1991.

[92] Yi-Ling Hsieh, Tsung-Yi Ho, and Krishnendu Chakrabarty. Biochip synthesis and dynamic error recovery for sample preparation using digital microfluidics. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(2):183–196, 2014.

[93] Kai Hu, Bang-Ning Hsu, Andrew Madison, Krishnendu Chakrabarty, and Richard B. Fair. Fault detection, real-time error recovery, and experimental demonstration for digital microfluidic biochips. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 559–564. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[94] Eric H Huang, Richard Socher, Christopher D Manning, and Andrew Y Ng. Improving word representations via global context and multiple word prototypes.

In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 873–882, 2012.

[95] Tsung-Wei Huang and Tsung-Yi Ho. A fast routability- and performance-driven droplet routing algorithm for digital microfluidic biochips. In *27th International Conference on Computer Design, ICCD 2009, Lake Tahoe, CA, USA, October 4-7, 2009*, pages 445–450, New York, NY, USA, 2009. IEEE Computer Society.

[96] Tsung-Wei Huang, Chun-Hsien Lin, and Tsung-Yi Ho. A contamination aware droplet routing algorithm for the synthesis of digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 29(11):1682–1695, 2010.

[97] Tsung-Wei Huang, Chun-Hsien Lin, and Tsung-Yi Ho. A contamination aware droplet routing algorithm for the synthesis of digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 29(11):1682–1695, 2010.

[98] Mohamed Ibrahim and Krishnendu Chakrabarty. Efficient error recovery in cyber-physical digital-microfluidic biochips. *IEEE Trans. Multi-Scale Computing Systems*, 1(1):46–58, 2015.

[99] Mohamed Ibrahim and Krishnendu Chakrabarty. Error recovery in digital microfluidics for personalized medicine. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 247–252, New York, NY, USA, 2015. ACM.

[100] Mohamed Ibrahim, Krishnendu Chakrabarty, and Kristin Scott. Synthesis of cyberphysical digital-microfluidic biochips for real-time quantitative analysis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 36(5):733–746, 2017.

[101] Hisao Ishibuchi, Yuji Sakane, Noritaka Tsukamoto, and Yusuke Nojima. Evolutionary many-objective optimization by NSGA-II and MOEA/D with large populations. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, San Antonio, TX, USA, 11-14 October 2009*, pages 1758–1763, New York, NY, USA, 2009. IEEE.

[102] Anil Jain, Karthik Nandakumar, and Arun Ross. Score normalization in multimodal biometric systems. *Pattern recognition*, 38(12):2270–2285, 2005.

[103] Christopher Jaress, Philip Brisk, and Daniel T. Grissom. Rapid online fault recovery for cyber-physical digital microfluidic biochips. In *33rd IEEE VLSI Test Symposium, VTS 2015, Napa, CA, USA, April 27-29, 2015*, pages 1–6, New York, NY, USA, 2015. IEEE Computer Society.

[104] Mais J. Jebrail, Ronald F. Renzi, Anupama Sinha, Jim Van De Vreugde, Carmen Gondhalekar, Cesar Ambriz, Robert J. Meagher, and Steven S. Branda. A solvent replenishment solution for managing evaporation of biochemical reactions in air-matrix digital microfluidics devices. *Lab Chip*, 15:151–158, 2015.

[105] Erik C. Jensen, Bharath P. Bhat, and Richard A. Mathies. A digital microfluidic platform for the automation of quantitative biomolecular assays. *Lab Chip*, 10(6):685–691, Mar 2010.

[106] Mathew M Jessop-Fabre and Nikolaus Sonnenschein. Improving reproducibility in synthetic biology. *Frontiers in bioengineering and biotechnology*, 7:18, 2019.

[107] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224, Cape Town, South Africa, 2010. ACM.

[108] Yousheng Jiang, Xuanyun Huang, Kun Hu, Wenjuan Yu, Xianle Yang, and Liqun Lv. Production and characterization of monoclonal antibodies against small hapten-ciprofloxacin. *African Journal of Biotechnology*, 10(65):14342–14347, 2011.

[109] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, 26(4):196–204, 2004.

[110] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices*, 37(5):304–314, 2002.

[111] Oliver Keszöcze, Robert Wille, Krishnendu Chakrabarty, and Rolf Drechsler. A general and exact routing methodology for digital microfluidic biochips. In Diana Marculescu and Frank Liu, editors, *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015*, pages 874–881, New York, NY, USA, 2015. IEEE.

[112] Oliver Keszöcze, Robert Wille, and Rolf Drechsler. Exact routing for digital microfluidic biochips with temporary blockages. In Yao-Wen Chang, editor, *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*, pages 405–410, New York, NY, USA, 2014. IEEE.

[113] Dhananjay Kimothi, Akshay Soni, Pravesh Biyani, and James M Hogan. Distributed representations for biological sequence analysis. *arXiv preprint arXiv:1608.05949*, 2016.

[114] Eric Klavins. Aquarium, your protocols will be assimilated. `http://klavinslab.org/aquarium.html`, 2014. Accessed: 2017-11-13.

[115] James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. Searching entangled program spaces, 2022.

[116] Panagiotis-Christos Kotsias, Josep Arús-Pous, Hongming Chen, Ola Engkvist, Christian Tyrchan, and Esben Jannik Bjerrum. Direct steering of de novo molecular

generation with descriptor conditional recurrent neural networks. *Nature Machine Intelligence*, 2(5):254–265, 2020.

[117] Tessa A. Lau, Pedro M. Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In Pat Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 527–534, Stanford, CA, USA, 2000. Morgan Kaufmann.

[118] Antonio Lavecchia. Machine-learning approaches in drug discovery: methods and applications. *Drug discovery today*, 20(3):318–331, 2015.

[119] Rodney Lax. The future of peptide development in the pharmaceutical industry. *PharManufacturing: The international peptide review*, 2:10–15, 2010.

[120] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[121] Thomas Lederer, Stefan Clara, Bernhard Jakoby, and Wolfgang Hilber. Integration of impedance spectroscopy sensors in a digital microfluidic platform. *Microsystem Technologies*, 18(7):1163–1180, Aug 2012.

[122] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449, 2018.

[123] Allen Leung and Lal George. A New MLRISC Register Allocator, 1998.

[124] Guangdi Li and Erik De Clercq. Therapeutic options for the 2019 novel coronavirus (2019-ncov). *Nature reviews Drug discovery*, 19(3):149–150, 2020.

[125] Yiyan Li, Hongzhong Li, and R. Jacob Baker. Volume and concentration identification by using an electrowetting on dielectric device. In *2014 IEEE Dallas Circuits and Systems Conference (DCAS)*, pages 1–4, 2014.

[126] Yiyan Li, Hongzhong Li, and R. Jacob Baker. A low-cost and high-resolution droplet position detector for an intelligent electrowetting on dielectric device. *Journal of Laboratory Automation*, 20(6):663–669, 2015. PMID: 25609255.

[127] Zipeng Li, Kelvin Yi-Tse Lai, John McCrone, Po-Hsien Yu, Krishnendu Chakrabarty, Miroslav Pajic, Tsung-Yi Ho, and Chen-Yi Lee. Efficient and adaptive error recovery in a micro-electrode-dot-array digital microfluidic biochip. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 37(3):601–614, 2018.

[128] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 639–646, Haifa, Israel, 2010. Omnipress.

[129] Chen Liao and Shiyan Hu. Multiscale variation-aware techniques for high-performance digital microfluidic lab-on-a-chip component placement. *IEEE Trans Nanobioscience*, 10(1):51–58, Mar 2011.

[130] Gabriel Lippmann. *Relations entre les phénomènes électriques et capillaires.* PhD thesis, Gauthier-Villars Paris, France:, 1875.

[131] Chia-Hung Liu, Kuang-Cheng Liu, and Juinn-Dar Huang. Latency-optimization synthesis with module selection for digital microfluidic biochips. In Norbert Schuhmann, Kaijian Shi, and Nagi Naganathan, editors, *2013 IEEE International SOC Conference, Erlangen, Germany, September 4-6, 2013*, pages 159–164, New York, NY, USA, 2013. IEEE.

[132] Yu-Chen Lo, Stefano E Rensi, Wen Torng, and Russ B Altman. Machine learning in chemoinformatics and drug discovery. *Drug discovery today*, 23(8):1538–1546, 2018.

[133] Tyson Loveless, Jason Ott, and Philip Brisk. A performance-optimizing compiler for cyber-physical digital microfluidic biochips. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 171–184. ACM, 2020.

[134] Tyson Loveless, Jason Ott, and Philip Brisk. Time-and resource-constrained scheduling for digital microfluidic biochips. In *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems*, pages 198–208, 2021.

[135] L. Luan, R.D. Evans, N.M. Jokerst, and R.B. Fair. Integrated optical sensor in a digital microfluidic platform. *IEEE Sensors*, 8:628–635, 2008.

[136] Lin Luan, Matthew W Royal, Randall Evans, Richard B Fair, and Nan M Jokerst. Chip scale optical microresonator sensors integrated with embedded thin film photodetectors on electrowetting digital microfluidics platforms. *IEEE Sensors Journal*, 12(6):1794–1800, 2012.

[137] Yan Luo, Bhargab B. Bhattacharya, Tsung-Yi Ho, and Krishnendu Chakrabarty. Design and optimization of a cyberphysical digital-microfluidic biochip for the polymerase chain reaction. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(1):29–42, 2015.

[138] Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. Error recovery in cyber-physical digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(1):59–72, 2013.

[139] Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. Real-time error recovery in cyberphysical digital-microfluidic biochips using a compact dictionary. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(12):1839–1852, 2013.

[140] J Ross Macdonald and E Barsoukov. Impedance spectroscopy: theory, experiment, and applications. *History*, 1(8):1–13, 2005.

[141] Elena Maftei, Paul Pop, and Jan Madsen. Tabu search-based synthesis of digital microfluidic biochips with dynamically reconfigurable non-rectangular devices. *Design Autom. for Emb. Sys.*, 14(3):287–307, 2010.

[142] Elena Maftei, Paul Pop, and Jan Madsen. Module-based synthesis of digital microfluidic biochips with droplet-aware operation execution. *JETC*, 9(1):2, 2013.

[143] Venkata RR Malapaka, Albert A Barrese III, and Brian C Tripp. High-throughput screening for antimicrobial compounds using a 96-well format bacterial motility absorbance assay. *SLAS Discovery*, 12(6):849–854, 2007.

[144] Polina Mamoshina, Marina Volosnikova, Ivan V Ozerov, Evgeny Putin, Ekaterina Skibina, Franco Cortese, and Alex Zhavoronkov. Machine learning on human muscle transcriptomic data for biomarker discovery and tissue-specific drug target identification. *Frontiers in genetics*, 9:242, 2018.

[145] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. *ACM Sigplan Notices*, 40(6):48–61, 2005.

[146] Chi-Liang Mao, Keith D Zientek, Patrick T Colahan, Mei-Yueh Kuo, Chi-Ho Liu, Kuo-Ming Lee, and Chi-Chung Chou. Development of an enzyme-linked immunosorbent assay for fentanyl and applications of fentanyl antibody-coated nanoparticles for sample preparation. *Journal of pharmaceutical and biomedical analysis*, 41(4):1332–1341, 2006.

[147] Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Using Large Corpora*, page 273, 1994.

[148] Henry Massalin. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126, 1987.

[149] Mike May. A diy approach to automating your lab. *Nature*, 569(7754):587–589, 2019.

[150] Jeffrey McDaniel, Christopher Curtis, and Philip Brisk. Automatic synthesis of microfluidic large scale integration chips from a domain-specific language. *2013 IEEE Biomedical Circuits and Systems Conference, BioCAS 2013*, pages 101–104, 2013.

[151] Scott McFarling. Procedure merging with instruction caches. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 71–79. ACM, 1991.

[152] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 187–195, Atlanta, GA, USA, 2013. JMLR.org.

[153] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, Columbus, OH, 1994.

[154] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[155] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[156] Tomáš Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 746–751, 2013.

[157] Ben Miles and Peter L Lee. Achieving reproducibility and closed-loop automation in biological experimentation with an iot-enabled lab of the future. *SLAS TECHNOLOGY: Translating Life Sciences Innovation*, 23(5):432–439, 2018.

[158] Elizabeth M Miller, Alphonsus HC Ng, Uvaraj Uddayasankar, and Aaron R Wheeler. A digital microfluidic approach to heterogeneous immunoassays. *Analytical and bioanalytical chemistry*, 399(1):337–345, 2011.

[159] MilliporeSigma. Designing peptides, 2022.

[160] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[161] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 997–1016, Tuscon, AZ, USA, 2012. ACM.

[162] Tom M Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[163] Kenneth T Miyasaki, R Iofel, and RI Lehrer. Sensitivity of periodontal pathogens to the bactericidal activity of synthetic protegrins, antibiotic peptides derived from porcine leukocytes. *Journal of dental research*, 76(8):1453–1459, 1997.

[164] Mohamed F Mohamed, Ahmed Abdelkhalek, and Mohamed N Seleem. Evaluation of short synthetic antimicrobial peptides for treatment of drug-resistant and intracellular staphylococcus aureus. *Scientific reports*, 6(1):1–14, 2016.

[165] Floriane Montanari, Lara Kuhnke, Antonius Ter Laak, and Djork-Arné Clevert. Modeling physico-chemical admet endpoints with multitask graph convolutional networks. *Molecules*, 25(1):44, 2019.

[166] Neeloffer Mookherjee, Marilyn A Anderson, Henk P Haagsman, and Donald J Davidson. Antimicrobial host defence peptides: functions and clinical potential. *Nature reviews Drug discovery*, 19(5):311–332, 2020.

[167] Hyejin Moon, Sung Kwon. Cho, Robin L. Garrell, and Chang-Jin Kim. Low voltage electrowetting-on-dielectric. *J. Appl. Phys.*, 92:4080–4087, 2002.

[168] Frieder Mugele and Jeanchristophe Baret. Electrowetting: from basics to applications. *Journal of Physics: Condensed Matter*, 17:R705–R774, 2005.

[169] Asher Mullard. New drugs cost us $2.6 billion to develop. *Nature reviews. Drug discovery*, 13(12):877, 2014.

[170] Miguel Angel Murran and Homayoun Najjaran. Capacitance-based droplet position estimator for digital microfluidic devices. *Lab Chip*, 12:2053–2059, 2012.

[171] Ananthan Nambiar, Maeve Heflin, Simon Liu, Sergei Maslov, Mark Hopkins, and Anna Ritz. Transforming the language of life: Transformer neural networks for protein prediction tasks. In *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 1–8, 2020.

[172] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 31–44, London, UK, 2020. ACM.

[173] Patrick Ng. dna2vec: Consistent vector representations of variable-length k-mers. *arXiv preprint arXiv:1701.06279*, 2017.

[174] Joo Hyon Noh, Jiyong Noh, Eric Kreit, Jason Heikenfeld, and Philip D. Rack. Toward active-matrix lab-on-a-chip: programmable electrofluidic control enabled by arrayed oxide thin film transistors. *Lab Chip*, 12(2):353–360, Jan 2012.

[175] The European Committee on Antimicrobial Susceptibility Testing. Routine and extended internal quality control for mic determination and disk diffusion as recommended by eucast, version 12.0, 2022.

[176] Kenneth O'Neal, Daniel T. Grissom, and Philip Brisk. Force-directed list scheduling for digital microfluidic biochips. In Srinivas Katkoori, Matthew R. Guthaus, Ayse Kivilcim Coskun, Andreas Burg, and Ricardo Reis, editors, *20th IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC 2012, Santa Cruz, CA, USA, October 7-10, 2012*, pages 7–11, New York, NY, USA, 2012. IEEE.

[177] Kenneth O'Neal, Daniel T. Grissom, and Philip Brisk. Resource-constrained scheduling for digital microfluidic biochips. *JETC*, 14(1):7:1–7:26, 2018.

[178] Jason Ott, Tyson Loveless, Chris Curtis, Mohsen Lesani, and Philip Brisk. Bioscript: programming safe chemistry on laboratories-on-a-chip. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):128, 2018.

[179] Phil Paik, Vamsee K Pamula, and Richard B Fair. Rapid droplet mixers for digital microfluidic systems. *Lab on a Chip*, 3(4):253–259, 2003.

[180] Phil Paik, Vamsee K. Pamula, and Richard B. Fair. Rapid droplet mixers for digital microfluidic systems. *Lab Chip*, 3:253–259, 2003.

[181] Fernando Pereira and Yves Schabes. Inside-outside reestimation from partially bracketed corpora. In *30th Annual Meeting of the Association for Computational Linguistics*, pages 128–135, 1992.

[182] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 275–286, Beijing, China, 2012. ACM.

[183] Malak Pirtskhalava, Anthony A Amstrong, Maia Grigolava, Mindia Chubinidze, Evgenia Alimbarashvili, Boris Vishnepolsky, Andrei Gabrielian, Alex Rosenthal, Darrell E Hurt, and Michael Tartakovsky. Dbaasp v3: database of antimicrobial/-cytotoxic activity and structure of peptides as a resource for development of new therapeutics. *Nucleic acids research*, 49(D1):D288–D297, 2021.

[184] Sudip Poddar, Sarmishtha Ghoshal, Krishnendu Chakrabarty, and Bhargab B. Bhattacharya. Error-correcting sample preparation with cyberphysical digital microfluidic lab-on-chip. *ACM Trans. Design Autom. Electr. Syst.*, 22(1):2:1–2:29, 2016.

[185] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[186] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[187] Michael G. Pollack, Alexander D. Shenderov, and Richard B. Fair. Electrowetting-based actuation of droplets for integrated microfluidics. *Lab on a Chip*, 2(2):96–101, 2002.

[188] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH*

2015, Pittsburgh, PA, USA, October 25-30, 2015, pages 107–126, Pittsburgh, PA, USA, 2015. ACM.

[189] Steven J Projan. Why is big pharma getting out of antibacterial drug discovery? *Current opinion in microbiology*, 6(5):427–430, 2003.

[190] Hong Ren, Richard B Fair, and Micheal G Pollack. Automated on-chip droplet dispensing with volume control by electro-wetting actuation and capacitance metering. *Sensors and Actuators B: Chemical*, 98(2-3):319–327, 2004.

[191] Andrew J. Ricketts, Kevin M. Irick, Narayanan Vijaykrishnan, and Mary Jane Irwin. Priority scheduling in digital microfluidics-based biochips. In Gielen [72], pages 329–334.

[192] Christian P Robert, George Casella, and George Casella. *Monte Carlo statistical methods*, volume 2. Springer, 1999.

[193] Steven L Rohall, Lydia Auch, Jonathan Gable, Jacob Gora, Johanna Jansen, Yipin Lu, Eric Martin, Margaret Pancost-Heidebrecht, Bill Shirley, Nikolaus Stiefl, et al. An artificial intelligence approach to proactively inspire drug discovery with recommendations. *Journal of Medicinal Chemistry*, 63(16):8824–8834, 2020.

[194] Pranab Roy, Hafizur Rahaman, and Parthasarathi Dasgupta. A novel droplet routing algorithm for digital microfluidic biochips. In R. Iris Bahar, Fabrizio Lombardi, David Atienza, and Erik Brunvand, editors, *Proceedings of the 20th ACM Great Lakes Symposium on VLSI 2009, Providence, Rhode Island, USA, May 16-18 2010*, pages 441–446, New York, NY, USA, 2010. ACM.

[195] Pranab Roy, Hafizur Rahaman, and Parthasarathi Dasgupta. Two-level clustering-based techniques for intelligent droplet routing in digital microfluidic biochips. *Integration*, 45(3):316–330, 2012.

[196] Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.

[197] Saman Sadeghi, Huijiang Ding, Gaurav J. Shah, Supin Chen, Pei Yuin Keng, Chang-Jin "CJ" Kim, and R. Michael van Dam. On chip droplet characterization: A practical, high-sensitivity measurement of droplet impedance in digital microfluidics. *Analytical Chemistry*, 84(4):1915–1923, 2012. PMID: 22248060.

[198] Michael I. Sadowski, Chris Grant, and Tim S. Fell. Harnessing qbd, programming languages, and automation for reproducible biology. *Trends in Biotechnology*, 34(3):214 – 227, 2016. Special Issue: Industrial Biotechnology.

[199] Michael J Schertzer, R Ben Mrad, and Pierre E Sullivan. Automated detection of particle concentration and chemical reactions in ewod devices. *Sensors and Actuators B: Chemical*, 164(1):1–6, 2012.

[200] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[201] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 305–316, New York, NY, USA, 2013. ACM.

[202] Eva Sciacca, Salvatore Spinella, Dino Ienco, and Paola Giannini. Annotated stochastic context free grammars for analysis and synthesis of proteins. In *European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, pages 77–88. Springer, 2011.

[203] David B Searls. A primer in macromolecular linguistics. *Biopolymers*, 99(3):203–217, 2013.

[204] Marwin HS Segler, Thierry Kogej, Christian Tyrchan, and Mark P Waller. Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS central science*, 4(1):120–131, 2018.

[205] Steve C. Shih, Irena Barbulovic-Nad, Xuning Yang, Ryan Fobel, and Aaron R. Wheeler. Digital microfluidics with impedance sensing for integrated cell culture and analysis. *Biosens Bioelectron*, 42:314–320, Apr 2013.

[206] Steve C. Shih, Ryan Fobel, Paresh Kumar, and Aaron R. Wheeler. A feedback control system for high-fidelity digital microfluidics. *Lab Chip*, 11:535–540, 2011.

[207] Yong Jun Shin and Jeong Bong Lee. Machine vision for digital microfluidics. *Review of Scientific Instruments*, 81(1), 2 2010.

[208] Jeremy Singer. *Static Program Analysis based on Virtual Register Renaming*. PhD thesis, University of Cambridge, UK, 2005.

[209] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 398–414, San Francisco, CA, USA, 2015. Springer.

[210] Hugo Sinha, Angela B. V. Quach, Philippe Q. N. Vo, and Steve C. Shih. An automated microfluidic gene-editing platform for deciphering cancer genes. *Lab Chip*, pages 11–12, 2018.

[211] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[212] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, pages 404–415, New York, NY, USA, 2006. ACM.

[213] Larisa N. Soldatova, Wayne Aubrey, Ross D. King, and Amanda Clare. The EXACT description of biomedical protocols. In *Proceedings 16th International Conference on Intelligent Systems for Molecular Biology (ISMB), Toronto, Canada, July 19-23, 2008*, pages 295–303, New York, NY, USA, 2008. Oxford University Press.

[214] Ekaterina A Sosnina, Sergey Sosnin, Anastasia A Nikitina, Ivan Nazarov, Dmitry I Osolodkin, and Maxim V Fedorov. Recommender systems in antiviral drug discovery. *ACS omega*, 5(25):15039–15051, 2020.

[215] W.R. Spickerman. Binet's formula for the tribonacci sequence. In *Fibonacci Quarterly*. Citeseer, 1982.

[216] WR Spickerman and RN Joyner. Binet's formula for the recursive sequence of order k. *Fibonacci Quarterly*, 22(4):327–331, 1984.

[217] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, pages 194–210, 1999.

[218] Vijay Srinivasan, Vamsee Pamula, and Richard Fair. Droplet-based microfluidic lab-on-a-chip for glucose detection. *Analytica Chimica Acta*, 507:145–150, 04 2004.

[219] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.

[220] Fei Su and Krishnendu Chakrabarty. Module placement for fault-tolerant microfluidics-based biochips. *ACM Trans. Design Autom. Electr. Syst.*, 11(3):682–710, 2006.

[221] Fei Su and Krishnendu Chakrabarty. High-level synthesis of digital microfluidic biochips. *JETC*, 3(4):1, 2008.

[222] Fei Su, William L. Hwang, and Krishnendu Chakrabarty. Droplet routing in the synthesis of digital microfluidic biochips. In Gielen [72], pages 323–328.

[223] Ian I. Suni. Impedance methods for electrochemical sensors using nanomaterials. *TrAC Trends in Analytical Chemistry*, 27(7):604 – 611, 2008. Electroanalysis Based on Nanomaterials.

[224] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009, pages 264–276, Savannah, GA, USA, 2009. ACM.

[225] Explainer: Peptides vs proteins - what's the difference?, Dec 2020.

[226] William Thies, John Paul Urbanski, Todd Thorsen, and Saman Amarasinghe. Abstraction layers for scalable microfluidic biocomputing. *Natural Computing*, 7(2):255–275, 5 2007.

[227] Michael C Thrun and Alfred Ultsch. Using projection-based clustering to find distance-and density-based clusters in high-dimensional data. *Journal of Classification*, 38(2):280–312, 2021.

[228] John D Turnidge. Susceptibility test methods: general considerations. *Manual of clinical microbiology*, pages 1246–1252, 2015.

[229] Olgierd Unold, Mateusz Gabor, and Witold Dyrka. Unsupervised grammar induction for revealing the internal structure of protein sequence motifs. In *International Conference on Artificial Intelligence in Medicine*, pages 299–309. Springer, 2020.

[230] John Paul Urbanski, William Thies, Christopher Rhodes, Saman Amarasinghe, and Todd Thorsen. Digital microfluidics using soft lithography. *Lab Chip*, 6:96–104, 2006.

[231] Jessica Vamathevan, Dominic Clark, Paul Czodrowski, Ian Dunham, Edgardo Ferran, George Lee, Bin Li, Anant Madabhushi, Parantu Shah, Michaela Spitzer, et al. Applications of machine learning in drug discovery and development. *Nature reviews Drug discovery*, 18(6):463–477, 2019.

[232] Laurens Van Der Maaten. Learning a parametric embedding by preserving local structure. In *Artificial intelligence and statistics*, pages 384–391. PMLR, 2009.

[233] Philippe Q. N. Vo, Mathieu C. Husser, Fatemeh Ahmadi, Hugo Sinha, and Steve C. Shih. Image-based feedback and analysis system for digital microfluidics. *Lab Chip*, 17:3437–3446, 2017.

[234] Peng Wang, Jiaming Xu, Bo Xu, Chenglin Liu, Heng Zhang, Fangyuan Wang, and Hongwei Hao. Semantic clustering and convolutional neural network for short text categorization. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 352–357, 2015.

[235] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[236] Matthew White Royal, Nan M. Jokerst, and Richard Fair. Droplet-based sensing: Optical microresonator sensors embedded in digital electrowetting microfluidics systems. *IEEE Sensors Journal*, 13:4733–4742, 12 2013.

[237] Irith Wiegand, Kai Hilpert, and Robert EW Hancock. Agar and broth dilution methods to determine the minimal inhibitory concentration (mic) of antimicrobial substances. *Nature protocols*, 3(2):163–175, 2008.

[238] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[239] Max Willsey, Ashley P. Stephenson, Chris Takahashi, Pranav Vaid, Bichlien H. Nguyen, Michal Piszczek, Christine Betts, Sharon Newman, Sarang Joshi, Karin Strauss, and Luis Ceze. Puddle: A dynamic, error-correcting, full-stack microfluidics platform. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 183–197, New York, NY, USA, 04 2019. ACM.

[240] Tao Xu and Krishnendu Chakrabarty. Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips. *JETC*, 4(3):11, 2008.

[241] Tao Xu, Krishnendu Chakrabarty, and Fei Su. Defect-aware high-level synthesis and module placement for microfluidic biochips. *IEEE Trans. Biomed. Circuits and Systems*, 2(1):50–62, 2008.

[242] Hailong Yao, Qin Wang, Yiren Shen, Tsung-Yi Ho, and Yici Cai. Integrated functional and washing routing optimization for cross-contamination removal in digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 35(8):1283–1296, 2016.

[243] Hailong Yao, Qin Wang, Yiren Shen, Tsung Yi Ho, and Yici Cai. Integrated Functional and Washing Routing Optimization for Cross-Contamination Removal in Digital Microfluidic Biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(8):1283–1296, 2016.

[244] Jacob Yasonik. Multiobjective de novo drug design with recurrent neural networks and nondominated sorting. *Journal of Cheminformatics*, 12(1):1–9, 2020.

[245] Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. Placement of defect-tolerant digital microfluidic biochips using the t-tree formulation. *JETC*, 3(3):13, 2007.

[246] Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. Bioroute: A network-flow-based routing algorithm for the synthesis of digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(11):1928–1941, 2008.

[247] Kourosh Zarringhalam, David Degras, Christoph Brockel, and Daniel Ziemek. Robust phenotype prediction from gene expression data using differential shrinkage of co-regulated genes. *Scientific reports*, 8(1):1–10, 2018.

[248] Lu Zhang, Jianjun Tan, Dan Han, and Hao Zhu. From machine learning to deep learning: progress in machine intelligence for rational drug discovery. *Drug discovery today*, 22(11):1680–1685, 2017.

[249] Yang Zhao and Krishnendu Chakrabarty. Cross-contamination avoidance for droplet routing in digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(6):817–830, 2012.

[250] Yang Zhao and Krishnendu Chakrabarty. Cross-contamination avoidance for droplet routing in digital microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(6):817–830, 2012.

[251] Yang Zhao, Tao Xu, and Krishnendu Chakrabarty. Integrated control-path design and error recovery in the synthesis of digital microfluidic lab-on-chip. *JETC*, 6(3):11:1–11:28, 2010.

[252] Huanyu Zhou, Li Xi, Daniel Ziemek, Shawn O'Neil, Julie Lee, Zachary Stewart, Yutian Zhan, Shanrong Zhao, Ying Zhang, Karen Page, et al. Molecular profiling of ulcerative colitis subjects from the turandot trial reveals novel pharmacodynamic/efficacy biomarkers. *Journal of Crohn's and Colitis*, 13(6):702–713, 2019.