UNIVERSITY OF CALIFORNIA
RIVERSIDE

Programming Safe Chemistry on Laboratories-on-a-Chip

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

Doctor of Philosophy

in

Computer Science

by

Jason Matthew Ott

December 2019

Dissertation Committee:

Professor Philip Brisk, Chairperson
Professor Vassilis Tsotras
Professor William H. Grover
Professor Mohsen Lesani

The Dissertation of Jason Matthew Ott is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This endeavor has been one of the most formative experiences in my life. To say that this was easy or simple would be nothing short of a lie. This has challenged, stretched, destroyed, and rebuilt me in ways I could never imagine. Thankfully, I am surrounded by advisors, family, and friends who were there to celebrate the successes, mourn the loses, and encourage through the tough. This thesis, while my own work, is a product of the support, mentorship, and grace from the people who surround me; this thesis wouldn't exist without you.

I would like to thank my advisor, Dr. Philip Brisk, I would not be here without his patient mentorship and support through this program. To the rest of my committee: Dr. Mohsen Lesani, who was patient and gentle with his instruction and advice; Dr. Grover, whose questions and insight were always insightful and brought new perspective; Dr. Vassilis Tsotras, whose interest and care for me spanned much more than just a student, thank you.

To Shelley, Ariana, Luciana, and Matteo: you have endured it all and I cannot thank you enough. Your support, love, crafts, smiles, laughs, tears, scrapes, bumps, bruises, and joy are what kept me going. You have been patient and sacrificed so much to participate in this journey. You are deserving of more than acknowledgement.

To my parents and family who have and will always be my cheerleaders and support me through the think and the thin, thank you.

To Dr. Skyler Windh and Dr. Prerna Budhkar, your encouragement and friendship are, at some points what kept me going. I genuinely appreciate your friendship, and will not forget the impact you've had on my life both professionally and personally.

To Tyson, Lindsey, and the "boys": the hours spent at the "farm" enjoying shared food and company kept me (and Shelley) sane and helped refine countless ideas and avenues. Tyson, you are a phenomenal sounding board for all things, and your wisdom is much appreciated.

To Bashar Romanous, Saheli Ghosh, Ravdeep Pasricha, Jill Foster, Gaurav Jhaveri, Ignacio del Castro, Carmen Solanas, Aditya Dhakal, and Nick Derimow: my life-long friends, I cherish our friendship dearly and you guys have been better friends than anyone could ask for.

To my lab mates: Dr. Brian Crites, Dr. Jeffrey McDaniel, Chad Davies, Serhan Gener, Amin Kalantar, Phillip Park, Josh Potter, Maryam Shahcheraghi, Dr. Zachary Zimmerman, and Jose Rodriguez Bourbon thank you for always having a "minute", knowing full well it will take far more time. You also suffered my idiosyncrasies, bad humor, and silly opinions, thank you.

Previous publications were used to create this thesis, portions of them are reprinted using the following permissions.

I would like to thank the following entities for their generous awards, fellowships and stipends which made this research possible.

- UCR for their Dean's fellowship aid, and
- The Department of Education for their Graduate Assistance in Areas of National Need (GAANN) Fellowship.

This dissertation is dedicated to:

- Jesus Christ, my savior, who has provided, to the very finest detail all the provisions necessary to survive this program,
- My wife Shelley, whose patience and understanding cannot be understated and whose selflessness through this process has not been overlooked or ignored,
- My kids: Ariana, Luciana, and Matteo. Let this be a reminder that God is faithful to the very end. He will provide, even when things look terribly bleak, his faithfulness is uncanny and unwaivering.

ABSTRACT OF THE DISSERTATION

Programming Safe Chemistry on Laboratories-on-a-Chip

by

Jason Matthew Ott

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2019
Professor Philip Brisk, Chairperson

Laboratories-on-a-chip (LoC), including their programmable variants (pLoC) (both flow- and droplet-based technologies), promise to fundamentally transform the life sciences through miniaturization and automation. However, adoption of these devices lags given their purported advantages. The lack of adoption is a result of two shortcomings: programming a valve or cell activation sequence is equivalent to that of an assembly language and system LoC designers typically design devices by hand, fundamentally limiting the size and complexity of the designed devices. Both shortcomings are symptomatic of missing abstractions limiting all but the most intrepid of life scientists to leverage (p)LoC devices for simplistic usages that are equivalent to a "hello, world"-level of complexity for their experiments and devices.

This thesis introduces the necessary abstractions providing life science practitioners the ability to fully utilize the potential (p)LoC devices promise. The first abstraction is a robust cookbook-style domain specific language (DSL) that allows practitioners to program and execute experiments on droplet-based (p)LoC devices. The second is an extension

to the DSL allowing flow-based (p)LoC system designers the ability to specify an experiment and fabricate the corresponding device capable of executing the experiment, even generating valve activations if desired. The third is a system that can provably protect a life scientist from inadvertently mixing, storing, and/or disposing of chemicals that would otherwise generate an explosion or lethal gasses. These abstractions allow practitioners to fully leverage the promised potential (p)LoC devices have on their discipline in a safe and easy manner.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Laboratories-on-a-Chip (LoC), including their programmable counterparts (pLoC) (both flow- and droplet-based) have been promising to revolutionize biological and chemical experimentation for decades. Their promise to reduce waste, increase safety, increase reproducability, and reduce experimentation cost is consistently verified, but has yet to be realized. In part, this is a result of their inherent complexity and difficulty to leverage in a meaningful manner that sufficiently augments the scientist — in short, they are difficult to integrate into the laboratory. The difficulty in finding the right application for scientists to make use of (p)LoC devices resides in the simple fact that the necessary abstractions required to ease the use of p(LoC) technologies hasn't existed. Historically, designing an LoC device, and if necessary, subsequently programming it, uses what amounts to cumbersome computer aided design (CAD) software and assembly-level programming, respectively. Both require a significant amount of domain-area expertise, where a biologist or chemist might have experience in CAD software, they most likely have no exposure to assembly-

level programming languages. Moreover, these CAD tools and assembly-level programming languages make no guarantees to safety.

With the missing abstractions, it is no surprise that the promised benefits of LoC devices haven't been realized. Simply put: it is faster for scientists to manually mix, heat, measure, or pipette than to use a (p)LoC device. This highlights the need for building these abstractions so that scientists can safely and confidently use p(LoC) devices as their primary tool for building and executing experiments.

In this dissertation we detail the necessary abstractions required to enable scientists to widely and easily apply (p)LoC devices in their laboratory. We begin by introducing a simple-to-use "cook-book" style domain specific language (DSL), *BioScript*, with it's provably safe type system in Chapter 3. *BioScript* uses an intuitive syntax that reads like a recipe, making it easy to express complicated and nuanced assays that can execute on droplet-based pLoC devices. The simplicity and robustness *BioScript* exhibits also makes it a viable candidate as a standardized means to disseminate assays within the scientific community. *BioScript*'s type system also guarantees that no two chemicals that shouldn't be combined are combined. It also guarantees that no chemical is used more than once; an interesting constraint not found in traditional computer science. We showcase *BioScript* by detailing how effective its type system is at preventing incompatible chemicals from mixing, showing its capability at preventing many reported laboratory incidents. We also compare *BioScript* to other languages targeting (p)LoC devices and demonstrate its ability to express the same assays using 67% less code. Finally, we demonstrate the utility of

BioScript by executing a standard biological assay on an industry-standard (p)LoC device: Appendix A.1.

Chapter 4 details necessary extensions to the *BioScript* syntax to fully support functions, including recursion. Because pLoC devices operate on reconfigurable computing platforms, there are a myriad of data-flow analysis that must occur for *BioScript* to fully support functions, including recursion. We also extend the *BioScript* compiler to include *Single-Instruction Multiple Data* (SIMD) semantics; allowing the compiler to fully utilize the inherent parallelizable properties (p)LoC devices enjoy. Part of the inclusion of the SIMD semantics include introducing a new instruction which allows scientists to easily create dilution trees with a single instruction. These enhancements complete the necessary abstractions for targeting droplet-based pLoC devices resulting in a feature-rich, fully functional programming language.

In Chapter 5, we introduce Xylograph, a workflow enabling *BioScript* programs to target flow-based (p)LoC devices. This workflow extends the *BioScript* compiler to include the necessary data-flow analysis and targeting engine enabling a scientist to express an assay in *BioScript*, synthesize, and fabricate the requisite device from the input program. This work removes the CAD tool domain-area expertise enabling more scientists to leverage (p)LoC technologies in their laboratories. The contribution this chapter brings is time saved designing and fabricating flow-based (p)LoC devices. Xylograph is capable of designing flow-based LoC devices in a matter of minutes; where a similar design would take an expert hours if at all. We prove that Xylograph is able to convert a *BioScript* program into a valid flow-based (p)LoC design and subsequently run the *BioScript* assay in Appendix A.1.

Finally, Chapter 6 details a provably safe storage and disposal system, *ChemStor*. *ChemStor* extends *BioScript*'s type system applying it to a wider class of problems that a scientist endures in a laboratory setting: how to safely store or dispose of chemicals used in the daily work of scientist. There have been countless home and laboratory incidents resulting from improperly storing or disposing chemicals. We demonstrate the efficacy of *ChemStor* by showing it could prevent numerous real-world reported incidents and a myriad of synthetic tests designed to push the boundaries of its capabilities.

Chapter 2

Background & Related Work

2.1 Digital Microfluidic Biochips (DMFBs).

This paper targets a specific class of programmable LoCs that manipulate discrete droplets of fluid via electrostatic actuation. Fig. 2.1 illustrates the electrowetting principle [108, 127]: applying an electrostatic potential to a droplet modifies the shape of the droplet and its contact angle with the surface. As shown in Fig. 2.2, droplet transport can be induced by activating and deactivating a sequence of electrodes adjacent to the droplet [138]; the ground electrode, on top of the array, improves the fidelity of droplet motion and reduces the voltage required to induce droplet transport.

Fig. 2.3 depicts a programmable 2D electrowetting array, called a "Digital Microfluidic Biochip (DMFB)." A DMFB can support five basic operations, shown in Fig. 2.4: transport (move a droplet from position $(x,\ y)$ to $(x',\ y')$), split (create 2 droplets out of 1), merge/mix (combine 2 droplets into 1, and, optionally, rotate them in a rectangular motion), and store (place a droplet at position $(x,\ y)$ for later u). A DMFB is reconfigurable,

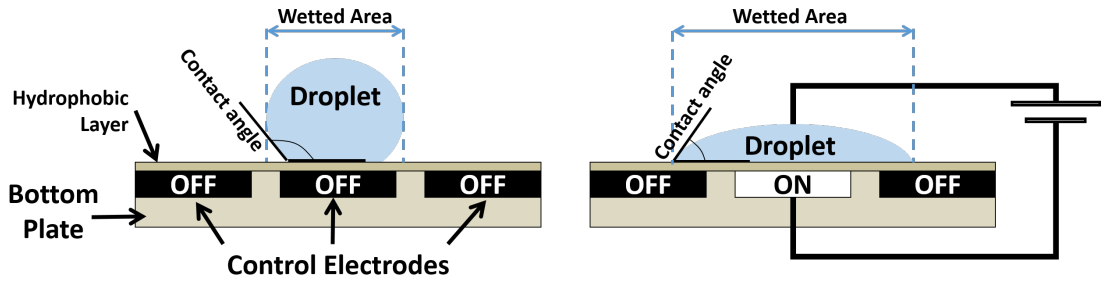


Figure 2.1: The electrowetting principle [108, 127]: applying an electrostatic potential to a droplet at rest reduces the contact angle with the surface, thereby increasing the surface area in contact with the droplet

as these operations can be performed anywhere on the array, and any given electrode can be used to perform different operations at different times. Droplet I/O is performed using reservoirs on the perimeter of the chip, which are not depicted in Fig. 2.5.

The DMFB ISA can be extended by integrating sensors [140, 15, 38, 163, 104, 151, 144, 103, 65, 90, 18, 130, 150, 105, 4], optical detectors [159, 110, 111, 176], heaters [112], or online video monitoring capabilities [152, 16, 81, 61, 173, 106]. Sensors and actuators create a "cyber-physical" feedback loop between the host PC controller and the DMFB. The ability to perform sensing, computation, and actuation based on the results of the computation adds control flow to the instruction set of the DMFB. Prior work has applied feedback-control for precise droplet positioning [151, 130, 18, 105, 16, 61, 81, 173, 106, 4] and online error detection and recovery [188, 113, 114, 80, 91, 5, 86, 87, 6, 137, 88, 106]; efforts to leverage these capabilities to provide control flow constructs at the language syntax level have been far more limited [73, 44, 45].

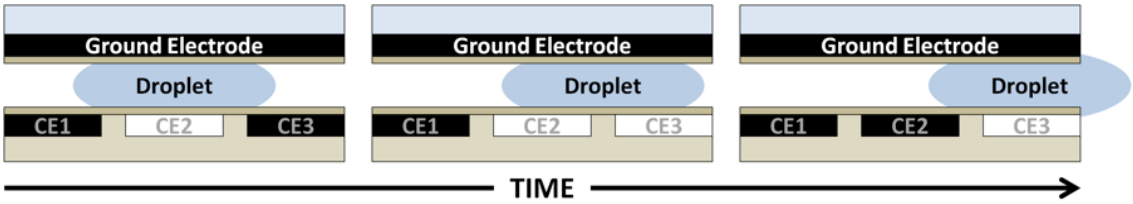


Figure 2.2: A droplet is transported from control electrode CE2 to neighboring electrode CE3 by activating CE3, and then deactivating CE2 (white: activated electrode; black: deactivated electrode).

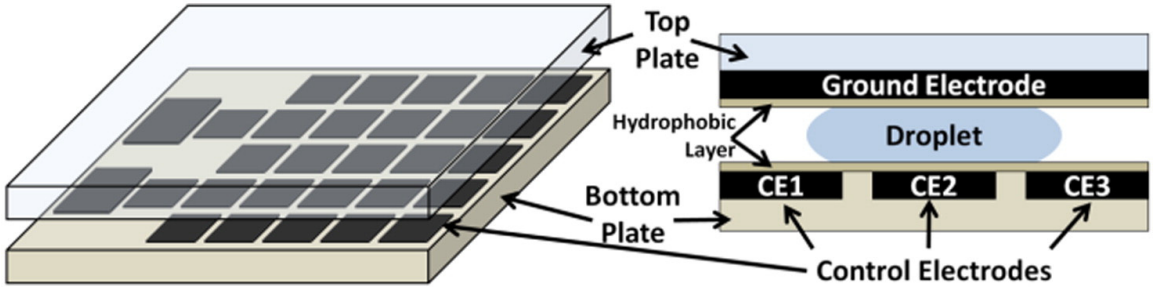


Figure 2.3: Left: A DMFB is a planar array of electrodes [138, 126, 70, 131, 76, 4]. Right: Cross-sectional view.

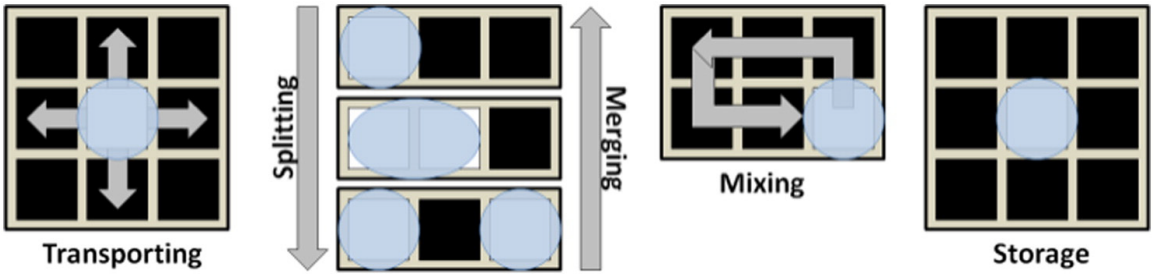


Figure 2.4: The DMFB ISA supports five basic operations: transporting, merging, splitting, mixing and storage, in addition to I/O on the perimeter of the array.

Figure 2.5: A DMFB (a) and it's reconfigurable instruction set (b).

2.2 DMFB Compilation

Compilation targeting DMFBs without control flow is mature. The input is effectively a fluidic variation of a traditional data dependence graph, where vertices represent fluidic operations and edges represent "fluidic dependencies", i.e., an edge (u_i, u_j) indicates that operation u_i produces a droplet $d_{i,j}$ that is used (consumed) by operation u_j. As shown in Figure 2.6, a compiler for a fluidic data dependence graph must solve three interdependent NP-complete problems: operation scheduling [49, 141, 161, 71, 132, 109], reconfigurable module placement [160, 185, 180, 181, 115, 107, 35, 116, 72], and droplet routing [162, 22, 36, 186, 82, 142, 143, 97, 96].

The scheduler must determine the time steps at which each biochemical operation occurs, while satisfying droplet dependency constraints and physical resource constraints of the device. The placer determines the location on the 2D electrode array where each operation is performed as the reaction progresses over time. The router ensures that droplets are transported from/to their start/stop positions (as determined by the placer) at appropriate times (as determined by the scheduler), while ensuring that droplets do not inadvertently collide with one another or interrupt any other ongoing operations on the chip during transport. If needed, the droplet router may introduce wash droplets to remove residue left by "functional" droplets that travel over the surface of the chip [83, 187, 183].

Compiling a *Control Flow Graph (CFG)* onto a DMFB is an active area of research; however, the two techniques proposed to date still compile basic blocks individually, and therefore lack a global scope. The first is to dynamically interpret the CFG by just-in-

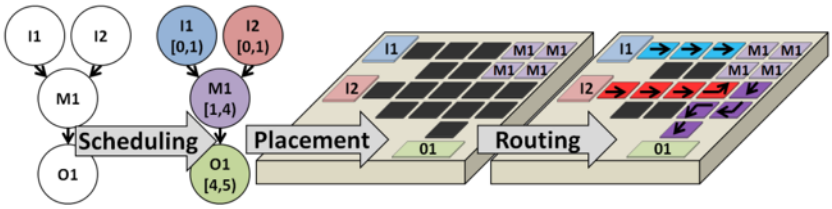


Figure 2.6: A DMFB compiler for biochemical programs without control flow.

time (JIT) compiling each basic block on-the-fly as the program executes [73]: as each basic block executes, the runtime performs computation on sensory data acquired from the device, which resolves conditions and determines the next basic block to JIT-compile.

The alternative is to compile the CFG statically. To date, the only technique which has been proposed compiles each basic block in isolation, using any of the scheduling, placement, and routing algorithms listed above[45]; however, the approach to placement taken by that compiler introduces a number of otherwise unnecessary droplet transport operations, which the placer introduced in this paper effectively eliminates. We provide a detailed discussion of the key differences here.

Similar in principle to graph coloring register allocation [33, 32, 26, 67] two placed operations or fluidic variables "interfere" if their lifetimes overlap, and interfering operations or variables must be placed at non-overlapping positions on the spatial 2D array to prevent inadvertent mixing and cross-contamination of fluids. Operations are defined atomically within basic blocks (i.e., an operation such as "mix" cannot start in one basic block and finish in another; further, the compiler introduced by [45] splits all fluidic variable live ranges at basic block boundaries, which serves to localize all interferences to operations and variables whose lifetimes start and end within the same basic block. This ensures that fully correct CFG compilation can be achieved by compiling each basic block in isolation

and inserting additional droplet transport operations at CFG boundaries to ensure that all droplets have the same starting positions along all possible paths leading into all basic blocks. As noted earlier, this yields a correct executable "program," but does nothing to eliminate or reduce the number of droplet transport operations that the compiler inserts.

For example, consider a fluidic dependence edge (u_i, u_j). If operations u_i and u_j are placed at distinct on-chip locations $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, then the compiler must insert an operation to transport droplet $d_{i,j}$ from p_i to p_j. On the other hand, if the placer can ensure that $p_i = p_j$, then the transport distance becomes 0, eliminating the need to insert the operation. This is identical, in principle, to coalescing performed during register allocation: assigning two variables involved in a copy operation to the same register creates an identity operation (a copy from a register to itself), which can be eliminated.

A similar observation holds for the ϕ- and π-functions of the Static Single Assignment (SSA) [47] and Static Single Information (SSI) [10, 154, 23] Forms. Without loss of generality, consider fluidic operations u_i and u_k, and a ϕ-function ϕ_j, which we denote using subscript-j to ensure notational consistency: u_i produces a droplet $d_{i,j}$, which is read by ϕ_j, and ϕ_j produces a droplet $d_{j,k}$, which is read by u_k; the exact statement of the ϕ-function is therefore $d_{j,k} \leftarrow \phi(\ldots, d_{i,j}, \ldots)$. SSA elimination replaces ϕ_j with a copy operation $d_{j,k} \leftarrow d_{i,j}$, which the compiler converts to a droplet transport operation. If u_i and u_k are placed at positions p_i and p_k, then the transport distance becomes 0 and the transport operation can be eliminated if the placer can ensure that $p_i = p_k$. Once again, this is analogous to how a traditional compiler attempts to coalesce $d_{i,j}$ and $d_{j,k}$ into a single variable to remove the copy operation during SSA elimination [158].

In short, the placer presented in this paper applies techniques derived from coalescing to minimize the number of droplet transport operations that it inserts; moreover, when droplet transportation operations are inserted, the placer attempts to minimize the overall transport distance while incorporating a static estimate of the criticality of the transport operation to overall assay execution time. In contrast, prior work on static DMFB compilation [45] emphasized correctness (i.e., the ability to statically compile a CFG), but did not attempt to reduce the number of length of droplet transport operations that were inserted.

Prior work on microfluidic placement has taken inspiration from spatial computing: [49, 181, 72] and have adapted placement algorithms originally introduced for dynamically reconfigurable FPGAs [17] to the microfluidic context. While practical and useful, these algorithms assume that tasks that are compiled onto a dynamically reconfigurable FPGA do not communicate, and thus do not effectively reduce droplet transport latencies when applied to microfluidics.

Unsurprisingly, there are also many principle similarities between microfluidic and data flow compilation, both of which entail placement and routing problems [156]. One key difference is that a data flow compiler must adhere to the microarchitectural details of the processing elements and interconnect architecture of the data flow target, which are considerably more intricate (enabled in no small part by multi-layer metallization) than the architecture of a DMFB (which is inherently planar). One key similarity is that both microfluidic and data flow compilation can improve performance by respectively minimizing data and fluidic transport distances. Additionally, many techniques to extract parallel execution regions from sequential code can generalize from data flow compilation to DMFBs.

One important caveat is that DMFBs lack any notion of a memory address space and/or off-chip memory hiearchy. As a result, fluidic pointers do not exist, which eliminates the need for a microfluidic compiler to tackle technical challenges such as memory disambiguation and memory access ordering.

2.2.1 Interprocedural Register Allocation in High Level Synthesis

Register allocation is a classic problem compilation faces. In traditional compilation, the number k of registers is fixed by the target architecture and a scalar is either allocated to a register or "spilled" into memory. Solving register allocation is traditionally accomplished by attempting to color an interference graph $G = (V, E)$ with k colors, where V contains the set of all variables declared at a given point in a program and E contains all the interferences between variables (u, v).

A counterpart similar to register allocation is necessary in DMFB synthesis as well. With no external memory, a DMFB has a ceiling on the number of locations that can be used to store droplets, which, as a result of the cohesive property of fluids, is significantly smaller than the total number of electrodes comprising the chip, as depicted in Figs. 4.6a and 4.6b. Hence, allocating resources for storage on a DMFB can be treated as register allocation without the ability to spill.

Although graph coloring in the general case is known to be NP-Complete [12, 58, 29], there have been successful techniques which reduce the complexity in common cases. Programs utilizing Static Single Assignment (SSA) form produce interference graphs that are chordal [27], which are optimally colorable in $O(|V| + |E|)$ time [66]. Similarly, Static Single Information (SSI), an extension to SSA form which places σ-functions at some split

points of a control-flow graph, is an interval graph [24] and also proven to be optimally colorable in $O(|V| + |E|)$ time [28]. SSI form allows variables to be renamed in every conditional context where used, resulting in a linearlized def-use chain, a property useful in the context of fluidic variables that no longer exist after they have been used, e.g., through a mix operation.

In order to account for interprocedural analysis, the creation of an *interprocedural interference graph (IIG)* is necessary [28]. This graph is either chordal or interval depending on which form a program takes – SSA or SSI, respectively. The IIG is defined for the whole program, i.e., interferences between variables defined locally in discrete procedures: if variable v defined in procedure P_i is live across a call c_m to P_j, then v interferes with every variable defined that is reachable from c_m.

2.3 Continuous Flow Devices

Continuous flow microfluidic devices have been demonstrated to span a wide variety of biological and chemical applications which includes, but are not limited to: protein crystallization[77], single-cell mRNA isolation and DNA synthesis [118], single-cell imaging [57], and interrogation of protein-DNA interactions[178]. Like their digital counterparts, continuous flow devices promise the same benefits to the biologist or chemist: higher experimentation throughput, reduced human error, and reduced sample/reagent usage relative the benchtop work they aim to replace. A result of their size, scale, and automated nature lends these devices to perform more complex and sophisticated experiments than their benchtop counterparts are capable of. Further, these attributes lead to direct impacts on global health care: mi-

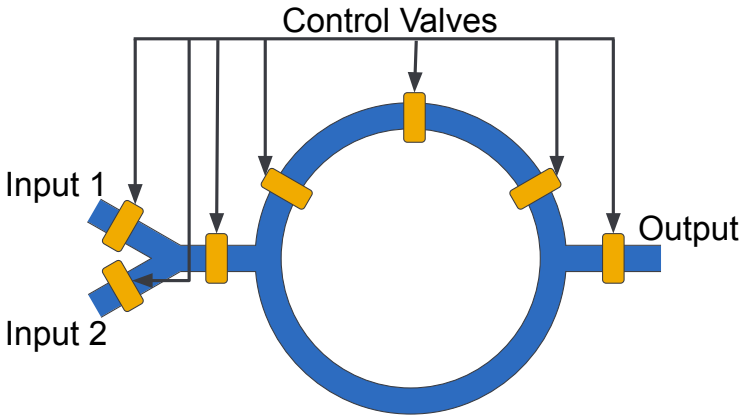
crofluidic devices create low-cost point-of-care devices that can have dramatic effects in both developed and developing nations [182].

There are two major categories within the continuous flow microfluidic taxonomy: *passive* and *active* devices. Passive microfluidic devices, in their simplest form, are channels either etched or carved into a rigid substrate[53, 84, 135] or imprinted in a flexible polymer[179] then mounted to a rigid substrate. The device functions by applying a pressure, either an external pump, gravity, or vacuum which moves fluids through the channels. As fluid flows through these channels, fluidic computation, of sorts, occurs.

While a discrete channel might not accomplish anything in particular, for a device to be of use, it must perform some function. The various functions a passive-flow device can take is dependent upon the component(s) included in the flow path. These components are channels comprised of sets of specific geometries that manipulate fluid in a desired manner. For example, the passive device depicted in Fig. 2.7a which performs a mix of two fluids, employs a serpentine channel mixer (a series of turns designed to agitate the input fluids). While we present a simple, single function device, devices are quickly increasing in complexity and sophistication and are capable of performing increasingly complex biological and chemical experiments.

Active flow, the second category in the continuous flow taxonomy, utilizes control valves to pump fluid through channels. These valves are analogous to the transistor, the fluidic variant. The components included in an active device still rely on specific geometries designed to perform their given function. Figure 2.7b depicts an active flow device with valves placed at specific points whose activation allow for the movement of fluid through the

(a)



(b)

Figure 2.7: Fig. 2.7a depicts a passive flow device that mixes two chemicals. The shape of the channels perturbs the fluids inducing turbulence, causing them to mix. A rotary mixer, Fig. 2.7b, uses control valves to circulate the input fluids resulting in the fluids mixing by traveling through the mix component.

device. The valves placed on the device can be controlled by an external pressure source. This allows for greater control of how fluid moves through the device and increases the complexity of operations that a device is capable of performing[168].

2.4 High-level Languages for Programmable Chemistry

Languages for programmable chemistry, including but not limited to microfluidics, typically fall into one of three categories: ontologies, laboratory automation, and device-specific languages. For a more general review, we refer the interested reader to [145].

2.4.1 Ontologies

Ontologies in synthetic biology, such as *Synthetic Biology Open Language (SBOL)* [63] or *EXACT* [157] aim to standardize how biochemical scientists discuss and disseminate information in a standardized form. They describe experiments and models in a common language, but cannot directly execute the experiment.

2.4.2 Laboratory Automation

Aquarium [100] specifies and composes laboratory workflows using a standard inventory, combining formal and informal statements with photographs. Processes are formed from individual protocols and are then parallelized and scheduled on the available laboratory equipment. In principle, the inventory could be expanded to include a programmable LoC programmed using *BioScript* or any other appropriate domains-specific language.

Cloud-based automation allows scientists to remotely execute biological procedures in robot-run laboratories over the Internet. Experiments are described using laboratory-specific domain-specific languages, such as *Transcriptic*'s *Autoprotocol* [1] and *Synthace*'s

[1] http://autoprotocol.org

Antha [2]. These languages could be extended to encompass LoCs as laboratory components, but would still need to interact with a high-level language to program the devices.

2.4.3 Device-Specific Languages for LoCs

BioStream targeted a programmable LoC designed primarily for serial dilution protocols which coupled a fluidic mixer to a fluidic memory [171, 166]. *BioStream* abstracted away the device-level details from the programmer and included algorithms to automatically generate serial dilution protocols from a set of user-specific target concentrations; however, after the initial publication, the specification and compiler were never released.

Aquacore [9, 8] is a programmable LoC comprising a collection of microvalve-based components connected to a centralized bus, which is programmed using the assembly-like *AquaCore Instruction Set (AIS)*. A high-level language like *BioScript* could be specialized for compatibility with Aquacore's components, and a *BioScript* to AIS compiler, although not presently under development, is certainly feasible.

BioCoder began as an ontology [11] and was later extended to target programmable LoCs [123, 73, 44, 45]. Although useful as a proof-of-concept, *BioCoder*'s syntax is unintuitive and it lacks a type system and formal semantics. *BioScript* is introduced here as a long-term replacement for *BioCoder*, as it is much closer to a natural language and is likely to be easier for a biologist to learn how to program.

[2] https://docs.antha.com

2.5 Fabrication Methods

The fabrication of flow-based microfluidic devices is a rich research area, with significant contributions, some borrowed from traditional computer architecture fabrication. While these thesis doesn't claim any contributions for fabrication technique, as this thesis describes a top-to-bottom high-level language down to fabrication tool, we provide a cursory overview of fabrication techniques.

2.5.1 Laminates

Laminate fabrication is a technique that involves bonding independently cut channels in some medium. When bonded these layers comprise the vias that allow fluid to flow through the device. Computer Numerical Control (CNC) milling is one application of this fabrication technique. A 3D rendering is provided to Computer-Aided Design (CAD) software which is capable of converting the rendering into *G-code*, a programming language capable of controlling a CNC mill. That G-code controls the CNC mill which etches a negative in acrylic or some other polymer. This method is considered the fastest and cheapest way to fabricate microfluidic devices[64].

2.5.2 Molding

Fabrication using molding relies upon photolithography to generate a positive mold. One can then use PDMS or some other liquid-set polymer to create a negative mold. Once the PDMS has cured, it can be bonded with glass creating a microfluidic device.

2.5.3 3D Printing

3D printing extrudes different layers of the mold onto a 2D plane. 3D printing is considered a fast and cheap fabrication method making design iterations fast and easy. Currently, 3D printing is limited in how small devices can be fabricate and it suffers from leaky vias that arise from interfacing layers of different materials. However, there are interesting endeavors to overcome these barriers using techniques like laser etching to melt plastic[184] or applying stereo lithography techniques to the 3D printing process[19].

2.5.4 Nanofabrication

Nanofabrication is a borrowed fabrication technique from the microprocessor. This technique works by adding photoresist material to a substrate and applying an ultra-violet light to remove any area not covered by the photoresist material and applying the functional material in the newly etched area(s)[25, 89]. The process repeats itself until all the photoresist layers and functional materials have been placed. This process is very expensive and time-intensive, but provides the most accurate and the smallest devices available[64].

Each of these fabrication methods have their strengths and weaknesses. However, they all share the same common denominator: they all require the manual placement of components and routing between them. This bottleneck demonstrates a clear and present need for an automated tool that handles the synthesis of a device from a high-level language.

2.6 Automated Laboratory Safety

Efforts to improve chemical safety generally fall into two categories—*system-based approaches* and *behavioral approaches*—both of which have limitations in avoiding storage- and disposal-related incidents:

- **System-based solutions** focus on building systems (often computer-based) that aid research labs in managing many different administrative functions involving chemicals. These functions may include generating various state and federal compliance reports, automatically issuing a purchase order should inventory of a chemical fall below preset levels, and sharing inventory among collaborating laboratories.[68, 41, 147, 62, 139] Some Chemical Inventory Management Systems (CIMS) employ simple safety features, like the ability to parse a chemical's Materials Safety Data Sheet (MSDS) to inform the researcher how to properly store the chemical. *Chempliance*[139] offers some guidelines on chemical disposal practices but is lacking any guarantees with respect to safety and does not track the volumes of chemicals.

- **Behavioral systems** focus on training and the human aspects of safety. Some of these systems train employees to focus on "safety first,"[98, 40] while others aim to identify disparities between institutional and individual beliefs about safety[153, 39]. These approaches focus on systemic issues that, while important, are sometimes relatively abstract and far removed from the specific day-to-day decisions faced by a researcher, like where to store a particular new chemical, or where to dispose of a certain waste chemical.

In summary, existing methods for improving chemical safety can inform researchers about general best practices, but fail to provide real-time guidance on specific storage and disposal decisions.

Chapter 3

BioScript

3.1 Introduction

The last two decades have witnessed the emergence of software-programmable laboratory-on-a-chip (LoC) technology, enabled by technological advances in microfabrication coupled with scientific understanding of microfluidics, the fundamental science of fluid behavior at the micro- to nano-liter scale. The net result of these collective advancements is that many experimental laboratory procedures have been miniaturized, accelerated, and automated, similar in principle to how the world's earliest computers automated tedious mathematical calculations that were previously performed by hand. Although the vast majority of microfluidic devices are effectively Application Specific Integrated Circuits (ASICs), a variety of programmable LoCs have been demonstrated [138, 171, 9, 93, 59, 8].

With a handful of exceptions, research on programming languages and compiler design for programmable LoCs has lagged behind their silicon counterparts. To address this need, this paper presents a domain-specific programming language, type system, and

compiler for a specific class of programmable LoCs that manipulate discrete droplets of liquid on a two-dimensional grid [138, 126, 70, 131, 76, 4]. The basic principles of the language, type system, and compiler readily generalize to programmable LoCs in general, realized across a wide variety of microfluidic technologies.

The presented language, *BioScript*, offers a user-friendly syntax that reads like a cookbook recipe. *BioScript* features a combination of fluidic/chemical variables and operations that can be interleaved seamlessly with computation, if desired. Its intended user base is not traditional software developers, but life science practitioners, who are likely to balk at a language that has a steep learning curve.

BioScript's type system ensures that each fluid is never consumed more than once, and that unsafe combinations of chemicals —those belonging to conflicting reactivity groups, as determined by appropriately qualified government agencies—never interact on-chip; *BioScript*'s type system is based on union types and was designed to ensure that type inference is decidable. This will set the stage for future research on formal validation of biochemical programs.

The *BioScript* compiler exploits the parallelism provided by the target platform to execute as many concurrent chemical operations as possible. Of particular importance, here, is a proper formulation of the problem of microfluidic placement in the scope of a control flow graph, rather than an individual basic block. The problem formulation presented here borrows ideas from graph coloring register allocation as well as spatial/data flow compilation; placement problem instances are solved using an evolutionary heuristic.

Table 3.1: *BioScript* supported fluidic operations.

Target	**Features**
Core	Material Decl
	Mix
	Output
	Store
	Repeat
Control Flow	Branch
	Loop
Digital	Detect
	Heat
	Split

The *BioScript* language, type system, and compiler are evaluated using a set of benchmark applications obtained from scientific literature. We use a microfluidic simulator to assess performance under ideal operating conditions, and also execute them on a real device, which is much smaller and supports a subset of *BioScript*'s operational capabilities. This result establishes the feasibility of high-level programming language and compiler design for programmable chemistry, and opens up future avenues for research in type systems and formal verification techniques within this non-traditional computing domain.

3.2 Overview

3.2.1 *BioScript* Syntax and Semantics

BioScript is a language for programmable microfluidics whose syntax aims to be palatable to life science practitioners, most of whom are not experienced programmers. We desired a syntax and semantics that expresses operations in a manner that closely resembles plain English. To keep the language small, we do not include operations in the language syntax

that can automatically be inferred by the compiler and/or execution engine. For example, the compiler can automatically infer implicit fluid transfers for a mix operation. *BioScript* features a semantics that targets (p)LoC technologies ranging from simplest to the most complicated. The syntax and semantics of *BioScript*'s type system are formally described in 3.3.

We divide *BioScript*'s fluidic operations into three categories, as shown in Table 3.1. The core of the language contains generic operations that are effectively common to all LoCs, such as declaration of fluidic variables and storage. *BioScript* also supports control flow operations, as well as DMFB-specific operations including sensing (detect) and actuation (heat and split). The detect instruction measures properties of droplets such as temperature or volume and the split instruction splits a droplet into multiple parts.

We begin with a self-contained example to illustrate the expressive capabilities of *BioScript*.

3.2.2 Example: PCR with Droplet Replenishment

Fig. 3.1 presents a *BioScript* specification for a DMFB-compatible implementation of the *Polymerase Chain Reaction (PCR)*, used to amplify DNA [129]. This specific example was obtained from the scientific literature [92], and expressed in *BioScript*.

PCR involves *thermocycling* (repeatedly heating then cooling) a droplet containing the DNA mixture undergoing amplification [lines 5-17]. In this implementation, thermocycling may cause excess droplet evaporation. This implementation uses a weight sensor to detect the droplet volume after each iteration [line 8]; if too much evaporation occurs [line 9] the algorithm injects a new droplet to replenish the sample volume [line 10-11], preheating

```
1  // PCRMasterMix is a commercially available
2  // pre-mixed concentrated solution that has
3  // all required components to perform PCR
4  // that are specific to the sample.
5  PCRMix = mix PCRMasterMix with Template for 1s
6  repeat 50 times {
7    heat PCRMix at 95C for 20s
8    volumeWeight = detect Weight on PCRMix
9    if (volumeWeight <= 50uL) {
10     replacement = mix 25uL of PCRMasterMix with 25uL of
         Template for 5s
11     heat replacement at 95C for 45s
12     PCRMix = mix PCRMix with replacement for 5s
13   }
14   heat PCRMix at 68C for 30s
15   heat PCRMix at 95C for 45s
16 }
17 heat PCRMix at 68C for 5min
18 save PCRMix
```

Figure 3.1: PCR with droplet replenishment [92]. It uses the target-specific save instruction.

```
1 step_1 = mix hydroxylamine_hydrochloride with toluene
2 heat step_1 at 105C for 24h
```

Figure 3.2: Safe assay for synthesizing acetaminophen [95]

a template solution [line 12] to ensure that replenishment does not affect the temperature of the DNA.

3.2.3 Example: Synthesizing Acetaminophen

Chemistry is an enormous space; chemists conservatively theorize that the number of pharmacologically active molecules is on the order of 10^{60}[51]. Cheminformatics is a field where chemists rely on computers to manage drug and compound discovery, a process whereby

```
step_1 = mix 10uL of acetic_acid with 10uL of
    tetrahydrofuran
step_2 = mix step_1 with 10uL of water
step_3 = mix step_2 with 10uL of acetonitrile
heat step_3 at 20C for 12h
```

Figure 3.3: Unsafe assay synthesizing acetaminophen [167]. Mixing water with acetonitrile creates hydrogen cyanide, an extremely poisonous and flammable gas. Hence, this reaction may be unsafe.

chemical libraries are used to screen and identify substances that have desired therapeutic effects, which are then tested on a biological cell. Although cheminformatics offers automation, many cheminformatic solutions may be unsafe when translated to the laboratory. *BioScript*'s type system can differentiate between safe and unsafe protocol specifications.

The search space explored by Cheminformatics includes all molecular combinations to synthesize concrete materials. In contrast, the static interaction table for type checking is limited to the reactivity groups of materials, which is necessarily conservative.

Acetaminophen, discovered in 1886[30], is a common pain medication used today, and its synthesis has been extensively studied. Reaxys[55], a leading chemical reaction database, details 275 different ways to synthesize acetaminophen, but ignores factors such as safety and efficiency. As an example, Figures 3.2 and 3.3 report *BioScript* specifications of two of the documented 275 paths: the one shown in Fig. 3.2 is safe, while its counterpart in Fig. 3.3 is unsafe and potentially dangerous. This distinction was made by *BioScript*'s type system. Leveraging the type system could further reduce the search space for viable screening procedures by partitioning the Reaxys database between safe and unsafe protocol specifications.

3.2.4 Type Systems and Safety

The Environmental Protection Agency (EPA) and National Oceanic and Atmospheric Administration (NOAA) have categorized 9,800 chemicals into 68 reactivity groups [56], defined by common physical properties of discrete chemicals. It is known that mixing materials from certain reactivity groups can produce materials from other reactivity groups; for example mixing acids and bases induces a strong reaction that produces salt and water. *BioScript*'s type system models reactivity groups as types. As a material can belong to multiple reactivity groups, a union type is associated with a material. Using standard reaction corpora, we calculate the type signature of the mix operation, which is fundamental throughout chemistry, as a table of abstract reactions between pairs of types, which results in a union of types.

At the same time, reactions vary in terms of safety. The EPA/NOAA categorization assigns one of three outcomes to the combination of chemicals: *Incompatible*, *Caution*, or *Compatible*. If the union type resulting from a mix operation includes a hazardous type, then the corresponding cell in the table is marked as being unsafe. Any biochemical procedure, or *assay*, specified in *BioScript* is allowed to execute only if it is safe. The signature of the mix operation does not include unsafe abstract reactions, which correspond to unsafe table cells. Therefore, the type system exclusively type-checks mix statements that do not produce hazardous materials. This is fundamental to the soundness of *BioScript*'s type system: it only type-checks assays that do not produce unsafe materials.

BioScript allows, but does not require, type annotations, saving the programmer from the burden of annotating programs with overly complicated union types. The assay

specifications presented in Figs. 3.1 to 3.3 do not use type annotations. *BioScript*'s type inference system can automatically infer types. Since, the EPA/NOAA classification begins with a finite set of material types, type inference can be reduced to efficiently decidable theories. We prove that the inference is sound: if a typing assignment is inferred, it can be used to type-check the assay. We also prove that it is complete: if there is a typing assignment with which the assay can be type-checked, the inference will discover it. Otherwise, the assay is rejected and marked as a potential hazard if no typing assignment can be inferred for it. Our experiments show that the type system is expressive enough to reject hazardous and to accept safe assays.

3.2.5 Software & Hardware Architecture

The *BioScript* compiler and runtime system is comprised of three discrete modules, as shown in Fig. 3.4: the compiler, the execution engine, and the DMFB. The front-end performs lexical analysis, parsing, abstract syntax tree (AST) generation, and AST to CFG conversion. The front-end inlines all function calls, noting that *BioScript* does not presently support recursion, and then passes the CFG to the back end.

The back-end converts the CFG to *Static Single Information (SSI)* form [10, 154, 23], under which each definition of a variable dominates each use, and each use of a variable post-dominates its definition, which linearizes def-use chains. The compiler executes a type inference algorithm (described in the next section) in the back end, rather than the front end, by gathering constraints and passing them to an SMT solver to infer types. If the *BioScript* program is typeable, the compiler passes the SSI-based CFG to the execution engine, which performs code generation (scheduling, placement, and routing), which may

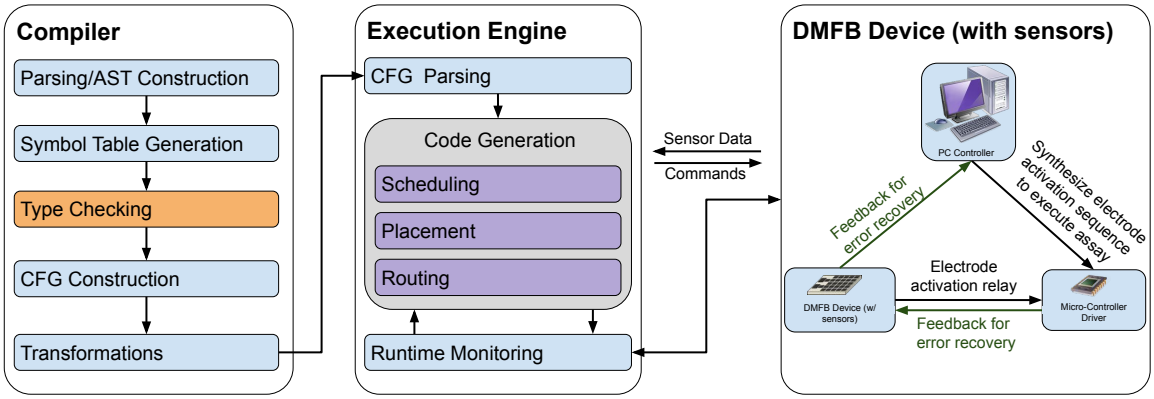


Figure 3.4: The *BioScript* compiler, execution engine and DMFB device.

be performed either statically [45] or dynamically [73]. The execution engine processes sensory feedback produced by the DMFB, including dynamic error detection and recovery; it may be necessary to re-compile parts of the assay, especially if a hard fault has been detected, rendering a portion of the device unusable; prior work has covered dynamic error recovery in detail [188, 113, 114, 80, 91, 5, 86, 87, 6, 137, 88, 106]. The execution engine terminates successfully when the control flow reaches the CFG exit node or unsuccessfully if the error recovery mechanism fails for any reason.

3.3 Type System

This section presents the core *BioScript* language, its semantics, the type checking and inference systems, and their guarantees. We present the core language and type system to showcase ideas and have implemented the type system for our full language.

We first present the *BioScript* syntax and its operational semantics that models the runtime execution of assays on pLoCs. Next, we present the type checking system, which guarantees that well-typed assays never perform unsafe operations at run time. Unsafe op-

erations include dangerous material interactions and attempts to access materials that have already been consumed. We establish the soundness of the type system as tandem progress and preservation properties. We then present the type inference system, which can automatically infer types of variables in assays. We establish the soundness and completeness of type inference with respect to type checking. Type inference succeeds if and only if there are types for the variables that make the assay type-check.

3.3.1 Syntax

Figure 3.5 represents the core language syntax. The language is imperative and a statement is a sequence of effectful instructions that involve side-effect-free terms.

A term t is one of: a variable x, a math operation, detection of a property for a material, or a value v. The set of variables and values are respectively denoted by $\mathcal{X}$ and $\mathcal{V}$. Math operations "$t_1 \oplus t_2$" are parametric in terms of the math operation $\oplus$. The DMFB has a set of detection modules, module_1, .., module_n, each of which measures a property of a material. A detect term "$\mathsf{detect\ module\ on}\ x\ \mathsf{for}\ t$" returns the property that module detects for the material represented by the variable x after a measurement for t time units. A value v is a material value mat, a real number r or natural number n. We use Mat, $\mathbb{N}$ and $\mathbb{R}$ to denote the set of materials, the set of natural and real numbers. Volume and other fluidic properties are captured in the full language, but not in the core language.

A statement s is either the sequence of an instruction i and another statement or the terminal skip statement. An instruction is either an assignment, material mixing, material splitting, or a conditional or loop control operation. An assignment "$x := t$"

assigns term t to variable x. The type system checks assignments to prevent aliasing for material variables. A mix instruction "$x :=$ mix x_1 with x_2 for t" mixes the two materials represented by the variables x_1 and x_2 for t time units and assigns the resulting material to the variable x. The type system checks the safety of mixing x_1's reactivity group with that of x_2. The split instruction "$\langle x_1, .., x_n \rangle :=$ split x into n" splits the material represented by the variable x into n parts and assigns them to variables x_0, .., x_n. The conditional and loop instructions are standard.

3.3.2 Operational Semantics

We model execution of *BioScript* assays on a DMFB as an operational semantics. A store σ maps the set of variables to a set of values. The state of the transition system is a pair (σ, s), where s is a statement. We now present the transition rules. If the conditions of no rule is satisfied for a term (other than values) or a statement (other than skip), its evaluation is *stuck*. A stuck term or statement models an error.

Figure 3.6.(a) presents the evaluation rules for terms. Terms are side-effect-free and leave the store unchanged; therefore, the transition rules for terms are of the form $(\sigma, t) \rightarrow t'$. The rule E-VAR checks that the store σ holds the value of a variable; if so, the variable's value is read from the store.

Rules E-MATHR1 and E-MATHR2 evaluate the first and second operands of a math operation in-order; then the rule E-MATH applies the operation to the arguments if both are numeric values. The rule E-DETECTR evaluates the time term t until it is reduced to a value. Then, the rule E-DETECT reduces a detect term if the value of the variable in

the store is a material. Property measurement by modules is modeled as the function detect that given the material, the module and the measurement time returns the property value. If the value of the variable in the store is not a material, the detect term is stuck.

Figure 3.6.(b) presents the evaluation rules for statements. The rule E-AssignR evaluates the right-hand side term (if it is not a variable); the rule E-Assign assigns the value to the variable in the store. The rule E-Assign′ transfers a material from the right-hand side variable to the left-hand side variable. The value of a variable is consumed when it is assigned to another variable. This restriction is necessary for material variables but can be easily lifted for numeric variables.

The rule E-MixR evaluates the time term; then, the rule E-Mix reduces a mix instruction if the values of both variables in the store are materials and their interaction is safe. Run time material interactions are represented by the concrete interaction function interact. Given input materials mat_i and mat_j and the interaction time r, the function interact returns $\lightning$ if mixing mat_i and mat_j for r time units is unsafe; otherwise, it returns the resulting material. The used variables x_1 and x_2 are removed from the store and the variable x is mapped to the resulting material. The evaluation of a mix instruction is *stuck* if either of the two variables does not represent a material value, or is already used and removed from the store, or the interaction of the two is unsafe.

The rule E-Split reduces a material split instruction by removing the input variable from the store and mapping the output variables to the splits. The function split models splits: given a material mat and the number of splits n, it split mat to n parts of equal volume and returns a part. Rule E-IfR reduces the condition term; rules E-IfTrue

and E-IfFalse reduce an if instruction to either the then or else statement depending on the value of the condition. The operator $\bullet$ on statements unrolls the second statement after the first as a sequence of instructions. The rule E-While unrolls the while statement once to an if statement.

3.3.3 Type Checking System

This section presents *BioScript*'s type system and its guarantees. We first consider the typing judgments, and the typing rules for terms and statements, and then establish progress and preservation properties. Using an abstract model of chemical interactions, the type system guarantees that any program that type-checks will never cause an unsafe material interaction at runtime. *BioScript*'s type system also guarantees that an operation is never applied to mismatching or missing values. In the evaluation of a type-checked program, no material variable is evaluated to an already used material.

Figure 3.5 represents the syntax of types. A type T is either the union of the scalar types $\cup\overline{S}$ or a type variable V; type variables are used for type inference. The overline notation $\overline{S}$ denotes multiple scalar types S. A scalar type is one of the material types Mat_1, .., Mat_n or real $\mathbb{R}$ or natural $\mathbb{N}$ number types. Each material type Mat_i represents a group of similar materials. A material value mat can be a member of one (or multiple) material types Mat_i written as $mat \in Mat_i$. Membership is trivially lifted to union types. The type Mat is defined as the union of all material types $Mat_1 \cup .. \cup Mat_n$. If a type T is a single scalar type, we elide the union symbol. For example, a union type with the single natural number type $\mathbb{N}$ can be denoted as $\mathbb{N}$. A scalar type S is a member of a type $T = \cup\overline{S}$, written as $S \in T$, iff S is one of the scalar types $\overline{S}$. A type T is a subset of

another type T', written as $T \subseteq T'$, iff any scalar type in T is also in T'. Similarly, union, intersection and equality of types are defined. A type environment Γ is a mapping from variables to types. An empty environment is denoted by $\emptyset$; an environment that includes the mapping from variable x to T is denoted by $\Gamma; x : T$. Since the *BioScript* assays are written as scripts, Γ contains the type assignment for all variables in the assay.

The type system uses interact-abs, the abstract interaction function, which accepts two scalar material types as arguments and returns a union type. The abstract interaction interact-abs is conservative with respect to the concrete interaction interact. If two material values mat_i and mat_j are members of two material types Mat_i and Mat_j, and the concrete interaction of mat_i and mat_j is unsafe, then the abstract interaction of Mat_i and Mat_j is undefined; otherwise, the result of the concrete interaction is a member of the type resulting from the abstract interaction of Mat_i and Mat_j. A discussion of how the interact-abs function is used is presented in 3.4.

The typing judgment for terms is $\Gamma, X \vdash t : T$, which states that under the typing environment Γ and set of available variables X, the term t has type T. The type system keeps track of available variables in the set X. These are the variables with unused values. Figure 3.7.(a) presents the type checking rules for terms.

Rule T-VAR states that a variable x has type T if it is typed as such in environment Γ and it is available ($x \in X$). Rule T-MATH states that if the terms t_1 and t_2 have the same numeric type, then the operation $t_1 \oplus t_2$ has the same type. Rule T-DETECT states that a detect term has the real return type if the following conditions hold: only properties of materials can be detected; thus, the input variable x should be typed as the union of only

material types. In addition, the term t for time should be typed as a real number. Rule T-Mat states that a material literal has the union type of material types that it belongs to. A material literal may belong to multiple material types. Rules T-Nat and T-Real type natural and real number literals.

The typing judgment for statements, $\Gamma, X \vdash s, X'$, states that term s is typed under typing environment Γ and available variable set X, yielding updated available variable set X'. A similar typing judgment, $\Gamma, X \vdash i, X'$, exists for instructions. Figure 3.7.(b) lists type checking rules for statements.

Rule T-Inst states that sequence $i; s$, comprising instruction i and statement s, is typed if both i and s are typed. The resulting available variable set from i is the input available variable set for s. Rule T-Skip unconditionally types the terminating statement skip. Rule T-Assign-1 types an assignment of a value to a variable. The type of the assigned value should be a subset of the type of the variable. The variable is also added to the set of available variables. Rule T-Assign-2 types assignment of a variable to another. The right-hand side variable is consumed and the left-hand side variable is added to the set of available variables. This rule prevents aliasing of materials: two variables may not represent the same material. (At the cost of brevity, the rule can be easily relaxed to not remove numeric variables from the available set.) Rule T-Assign-3 types assignments where the assigned term has a numeric type. This rule restricts aliasing through reading from terms to only numeric values. (With the current set of typing rules for terms, any term other than variables and values can be typed only as a numeric type. However, we keep the numeric type condition in this rule for future extensions of terms.)

Rule T-Mix types a mix instruction if the following conditions hold: only materials can be mixed; thus, both input variables x_1 and x_2 should be typed as the union of only material types; term t (time) should be typed as a real number; typing fails if the abstract interaction function is undefined for any pair of material types in the union types for x_1 and x_2; if material types are defined for all such pairs, then the result represents a safe material; as the result is assigned to x, the type of x in the environment should be a superset of the resulting material types; since the materials represented by x_1 and x_2 are consumed, x replaces x_1 and x_2 in the set of available variables.

Rule T-Split types a split instruction of variable x into variables x_1, .., x_n. Only materials can be split; thus, variable x should be typed as the union of material types. The split parts are assigned to variables x_1, .., x_n; thus, the type of each x_i should be a superset of the type of x in the environment; x_1, .., x_n replace x in the set of available variables.

Rule T-If types an if instruction; the output available set is conservatively the intersection of the output available sets of the then and else statements. Rule T-While types a while loop of a statement s. Since the output available set of one iteration can be the input available set of the next, the output available set for s should be a superset of its input available set X. Since the condition may fail and the loop may not execute, the output available set of the while statement is conservatively X.

As classical results establish, there is correspondence between type systems and data-flow analysis[134]. The type system corresponds to flow-sensitive analysis for defined variables and flow-insensitive analysis for interaction safely.

We now state progress and preservation lemmas that together state that well-typed programs never execute unsafe operations. As explained for the operational semantics, there is no reduction rule for unsafe operations, that is unsafe operations are stuck. The following lemmas state that well-typed programs never get stuck.

To state the lemmas, we first define the consistency invariant between the runtime store σ and the static type environment Γ and the set of variables X. A store σ is consistent with the type environment Γ and the set of variables X, if every variable that is in X and Γ, has a value in σ whose type complies with Γ.

Definition 1 *For all* Γ, X *and* σ, *consistent*$(\Gamma,\ X,\ \sigma)$ *iff for all* x *and* T *such that* $x \in X$ *and* $(x : T) \in \Gamma$, *we have* $\sigma(x) \in T$.

The following progress lemma states that well-typed programs are not stuck, i.e., they can take a step. More precisely, if a statement is typed, then it is either the terminal statement skip or it can make a step with every consistent store. The proofs for these Lemmas are available in the supplemental material.

Lemma 1 (Progress) *For all* Γ, X, s *and* X', *if* $\Gamma, X \vdash s, X'$ *then either* s *is* skip *or for all* σ *such that consistent*$(\Gamma,\ X,\ \sigma)$, *there exists* σ' *and* s' *such that* $(\sigma,\ s) \to (\sigma',\ s')$.

The following preservation lemma states that if a well-typed program steps, the resulting program is also well-typed. More precisely, if a statement s is typed and with a consistent store σ steps to a statement s' and a store σ', then s' is typed and σ' is consistent as well.

Lemma 2 (Preservation) *For all* Γ, X, σ, s, X'', σ', s', *if* $\Gamma, X \vdash s, X''$, *consistent*(Γ, X, σ) *and* $(\sigma,\ s) \rightarrow (\sigma',\ s')$ *then there exists an* X' *such that* $\Gamma, X' \vdash s', X''$ *and consistent*(Γ, X', σ').

3.3.4 Type Inference System

We now present the type inference system. A type can be not only a union of scalar types but also an unknown type variable. The type inference rules match the corresponding type checking rules but restate the conditions as constraints. After the type inference system derives the constraints for a program, a satisfying model for the constraints yields types for the variables of the program.

The type inference judgment for terms is $\Gamma, X \vdash t : T \mid C$ that states that under the typing environment Γ and available variables X, the term t is typed as T if the constraints C are satisfied. The constraints C are quantifier-free set theory formulae. Since we have a finite set of scalar types, the constraints can be reduced to quantifier-free formula in the theory of equality. The type inference rules for terms are presented in Figure 3.8.(a)

The rule CT-Var introduces no constraints. The rule CT-Math introduces constraints requiring the two arguments to be of the same numeric type. The rule CT-Detect introduces constraints that require the type of the material variable x not to include numeric types. This is because, as mentioned for the rule T-Detect, only properties of materials can be detected; thus, the type of the material variable should only include material types. The rules CT-Mat, CT-Real and CT-Nat type literal values without constraints.

The type inference judgment for statements is $\Gamma, X \vdash s, X' \mid C$, which states that under typing environment Γ and available variables X, statement s is typed and the resulting available variables are X' if all conditions C are satisfied. There is also a similar type inference judgment for instructions: $\Gamma, X \vdash i, X' \mid C$. Figure 3.8.(b) presents type inference rules for statements. Rules CT-Assgn-1, CT-Assgn-2 and CT-Assgn-3 type the assignment instruction. If the right-hand side is a material value, then rule CT-Assgn-1 introduces a constraint that requires the type of the right-hand side to be a subset of the type of the left-hand side. Rule CT-Assgn-2 and CT-Assgn-3 similarly mirror rules T-Assign-2 and T-Assign-3. Rules CT-Mix, and CT-Split restate the conditions of their corresponding typing rules as constraints. Rules CT-If, and CT-While introduce a constraint that requires the condition term to be of natural number type.

To infer types for program s, we first check if the type inference judgment $\Gamma_0, \emptyset \vdash s, X' \mid C$ is derivable; Γ_0 maps each variable x in s to a fresh type variable V_x, and the initial set of available variables is empty. We note that to support optional type annotations for variables, Γ_0 can map an annotated variable to the concrete type annotation instead of a type variable. If the judgment cannot be derived, the program may access an uninitialized variable or an already used material variable. Thus the program is rejected. If the judgment can be derived and constraints C are satisfiable, then any model m of C provides the typing $[\overline{x \mapsto m(V_x)}]$ for the program.

We now state the soundness and completeness of the type inference system, which collectively state that types can be inferred for a program iff it is typeable. The soundness lemma states that, if the type inference system infers types for a program, then with the

inferred types, the type checking system can type-check the program. More precisely, if under a type environment Γ with type variables, the type inference system derives the set of type constraints C for a statement s and C is satisfiable with a model m, then applying m to Γ yields the concrete type environment $m(\Gamma)$ under which the type checking system can type-check s.

Lemma 3 (Soundness) *For all* Γ, X, s, C, X', *and* m, *if* $\Gamma, X \vdash s, X' \mid C$ *and* m *is a model for* C *then* $m(\Gamma), X \vdash s, X'$.

The completeness lemma states that if, for a program, there exist types for variables under which the type checking system can type-check the program, then the type inference system can infer those types. More precisely, if there exists a model m that maps type variables in a type environment Γ to concrete types such that under the concrete type environment $m(\Gamma)$, the type checking system can type-check s, and the type inference system, under Γ, derives the constraints C for s, then C is satisfiable and m is a model for it.

Lemma 4 (Completeness) *For all* Γ, X, s, X', X'', C, *and* m, *if* $m(\Gamma), X \vdash s, X'$ *and* $\Gamma, X \vdash s, X'' \mid C$ *then* m *is a model for* C.

The proofs are available in Appendix B.

3.4 Implementation

This section describes the underlying details of implementation of the *BioScript* language, its type system and its code generator.

3.4.1 *BioScript*

The *BioScript* language was implemented as described in Section 3.2. As DMFBs do not offer external fluidic storage, there is no possibility to implement a stack or heap of substantial size. For these reasons, *BioScript* provides *inline* functions exclusively and does not support recursion; similarly, *BioScript* does not support arrays, even of constant size, as doing so would significantly inhibit portability. We hope to address these issues in greater detail in a future publication. *BioScript* handles variable assignment implicitly, e.g., Fig. 3.10d. However, the scientist declares a `manifest` of chemicals that is used throughout the assay ("blood" and "water", in this case) and the *BioScript* compiler infers the *dispense* and *move* operations.

3.4.2 The Type System

BioScript's type system utilizes static type checking, which runs during compilation. The type system automatically infers types using an abstract interaction function that is a conservative over-approximation of the resulting chemical types of each interaction. The type system uses the 68 EPA/NOAA reactivity groups as the material types $\overline{Mat_i}$, that together with natural $\mathbb{N}$, and real $\mathbb{R}$ numbers, constitute the set of scalar types S.

We calculate the abstract interaction function interact-abs (used in Section 3.3) as a table that is indexed by two material types and stores union types. Each reactive group or type Mat_i comprises a non-empty set of chemicals C_i. Abstract mixing of a pair of material types Mat_i and Mat_j effectively mixes each pair of chemicals (c_i, c_j) in the cross product $C_i \times C_j$. If any interaction is *Incompatible*, the table entry for (Mat_i, Mat_j) is marked as hazardous (or undefined, as modeled in Section 3.3). Otherwise, if the mix operation yields a new chemical c_k, we use the industry-standard *ChemAxon* [34] computational chemistry library to assign a union type $\cup\overline{Mat_k}$ to c_k, which are added to the union type of the cell for Mat_i and Mat_j. In practice, molecules of c_i and c_j will remain after mixing c_i and c_j, even if a reaction occurs, and the presence of extra molecules at the micro-liter scale, or smaller, may have a non-negligible impact on the underlying chemistry or biology [1]. To account for this fact, Mat_i and Mat_j are also added to the cell. Since type assignment to concrete chemicals is conservative and we include the input types in the resulting union type, the types in the table represent an over-approximation of the chemicals that can result from concrete interactions.

The type system implements Hindley-Milner type reconstruction [124]. Constraints are gathered from the CFG according to the type inference rules. Constraints are encoded in the SMTLib2 format and passed to Z3 [48] for satisfiability. In cases that no type can be inferred, deciding which part of the program is to blame is a classical problem[174]. Life scientists using *BioScript* would benefit from localized typing errors; including the necessary heuristics is left for future work.

[1] This was confirmed by a collaborator in the Bioengineering department of the authors' institution.

There may be instances where scientists need to create hazardous reactions, which the type system would correctly reject. For example, mixing ammonia with bleach yields chlorine gas, which is deadly. The type system correctly prevents mixing ammonia and bleach; however, in organic chemistry and industrial manufacturing, chlorine is a foundational material, and many consumer products rely on chlorine for production, e.g. PVC piping and cleaning agents. Thus, a scientist may wish to create chlorine to use elsewhere in the assay. In this case, the type system generates all relevant errors and warnings, but allows the programmer an override to finish compilation and execute the assay.

While not necessary, the execution engine is capable of performing dynamic checks as well. Before each interaction, it consults the EPA and NOAA categorization: if the interaction is *Incompatible*, then execution is halted or the user is prompted to override necessary safety precautions in order to proceed. This is a final safety check, given the safety-critical nature of the domain.

3.4.3 Code Generation

The code generator presently targets three DMFB back-ends. The first two are simulators that can statically compile [45] or dynamically interpret [73] assays featuring control flow operations. 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.

The code generator also targets a real-world platform called DropBot [61], shown in Fig. 3.9a. Although DropBot features real-time object tracking, it does not, at present, support execution of assays that feature control flow. The DropBot interface allows the user to specify an electrode activation sequence using either a graphical interface, shown in or through a JSON file. We modified the code generator to produce a DropBot-compatible JSON file. Fig. 3.9 shows DropBot's graphical interface while manipulating droplets on a real-world device.

3.5 Evaluation

The objectives of *BioScript* are to reduce the time and costs of scientific research and to provide a safe execution environment for chemists and biologists with respect to chemical interactions. As noted earlier, *BioScript* is a DSL that enables high-level programming and direct execution of bioassay on (p)LoCs. These objectives inform our selection of metrics to evaluate *BioScript*.

3.5.1 Language

Compared to other languages, *BioScript* offers an intuitive and readable syntax and a type system. As a point of clarification, we do *not* claim that *BioScript* offers a performance advantage with respect to other languages; performance primarily depends on the algorithms implemented in the compiler back-end and execution engine, which are compatible, in principle, with any language and front-end. Hence, our evaluation emphasizes qualitative metrics of the language.

First, we compare *BioScript*'s syntax to three other languages: the *AquaCore Instruction Set (AIS)*, a target-specific assembly-like language [9]; *Antha*, a language for cloud-based laboratory automation [164]; and BioCoder, a C++ library that has been previously specialized for DMFBs [11, 73, 44]. We review these three languages in greater detail in Chapter 2 and Appendices C and E. Our comparison uses a set of compact, yet representative, bioassays taken from published literature. As an illustrative example, Fig. 3.10 shows a simple assay (a Mix followed by a Heat instruction) in all four languages; *BioScript*, by far, has the shortest description.

- The *BioCoder* specification (Fig. 3.10a) is written as a C++ program. It does not require awareness of the underlying physical resources of the target device, but does require explicit statements to synchronize time-steps and to terminate the assay.

- The *Antha* specification (Fig. 3.10b), is imperative, but involves unintuitive notation such as []* and . operators (e.g., []*wtype.LHComponent).

- The *AIS* specification (Fig. 3.10c) operates at a lower level of abstraction. AIS is an assembly language, that requires resource awareness, which inhibits retargetability. The programmer must explicitly declare fluids, manually bind fluidic operations to resources, and explicitly transfer fluids between resources. Our back-end can automate this process.

- The *BioScript* specification (Fig. 3.10a) requires 3 lines of code. The specification is compact and declares variables implicitly.[2]

[2]In fact, the only arguably superfluous keywords are *of* (preposition), *with* (preposition), *at* (preposition) and *for* (preposition or conjunction), which bring the language closer to written English than to an imperative programming language.

Figure 3.11 compares the number of lines of code required to specify seven representative bioassays using the four languages; three of the seven assays were not compatible with *AIS* (which is tethered to a specific pLoC [9]) and *Antha* (which is tethered to a cloud laboratory), so we only report four assays for those languages. We do not count empty lines (for spacing/aesthetic purposes) or lines that contain comments. We wrote each assay based on our notion of human readability, which generally meant one statement/operation per line for *AIS*, *BioCoder*, and *Antha*. As shown in Fig. 3.10d, the mixture statement in *BioScript* succinctly encompasses two implicit variable declarations with fluid type and volume information.

Across the four compatible assays, *BioScript* required 68% fewer lines of code than *AIS* and 73% fewer lines of code than *Antha*. Across all seven assays, *BioScript* required 65% fewer lines of code than *BioCoder*, which can target DMFBs, [73, 44], unlike *AIS* and *Antha*. Although these results do not account for subjective experience, we believe that they convey the same basic sentiments as Fig. 3.10: *BioScript* has an intuitive syntax and will be far easier for scientists to learn and use compared to existing languages in the same space. Source code for all implementations of the bioassays reported in Figure 3.11 are included in Appendices C and E.

3.5.2 Type System Evaluation

BioScript's type system's main purpose is to prevent inadvertent production of hazardous chemicals. We evaluate its ability to detect hazardous mixing in *BioScript* descriptions of 5 reported real-world incidents [7, 21], as well as several hand-generated examples. To the

best of our understanding, *BioScript*'s type system is first-of-its kind, so there are no prior type systems to compare against.

Table 3.3 summarizes the results of our experiments. The first four tests are taken from documented real-world situations in which chemists ignored safety precautions while carrying out experiments. The first three are incidents documented by the *American Industrial Hygiene Association (AIHA)* [7]. *Mustard gas* refers to a documented situation where an individual mixed two common reagents used to clean swimming pools, inadvertently creating mustard gas. *SafetyZone* refers to a documented explosion where a student mixed a sulfuric acid/hydrogen peroxide mixture with acetone [50] (it remains unknown whether this explosion was intentional or accidental). The type system correctly identified the presence of safety hazards in all of these cases.

We also tested the type system on 14 assays that were known to be safe; *BioScript*'s type system successfully inferred types in all of these cases. We have intentionally chosen to express only the assays in Table 3.2, noting the limited benchmarks. These assays are currently being used in the bio-chemical sciences today. By demonstrating *BioScript*'s ability to express, type-check, and execute these assays, we demonstrate the power that *BioScript* provides scientists. We could have created synthetic benchmarks, but all would be derivative of the presented assays and, ostensibly, would not provide as compelling an argument to a life scientist of *BioScript*'s capabilities.

Table 3.2: Compile time, the number of constraints gathered, and simulated execution times for the safe and unsafe assays.

Benchmark	**Compilation Time** (sec)	**Constraint Solving** (sec)	**Gathered Constraints**	**Execution Engine Time** (m:s:ms)	**Execution Time** (h:m:s)
AIHA 1 [7]	0.012	0.936	70	N/A	N/A
AIHA 2 [7]	0.012	1.648	68	N/A	N/A
AIHA 3 [7]	0.014	1.214	17	N/A	N/A
Broad Spectrum Opiate [14, 117, 94]	0.011	0.887	11	0:18:55	0:23:21
Ciprofloxacin [94]	0.023	1.722	14	101:31:80	128:54:32
Diazepam [79]	0.024	1.007	14	96:48:13	121:01:39
Dilution [79]	0.014	0.892	9	0:21:05	0:26:33
Fentanyl [117]	0.018	0.900	13	126:32:40	158:10:80
Full Morphine [79]	0.048	4.188	19	127:16:78	159:06:17
Glucose Detection [9]	0.012	1.633	14	0:23:77	0:29:73
Heroine [79]	0.020	1.553	13	126:32:40	158:10:80
Image Probe Synthesis [9]	0.015	2.181	13	8:38:96	10:47:50
Morphine [79]	0.018	1.026	13	126:32:40	158:10:80
Mustard Gas [21]	0.015	1.433	83	N/A	N/A
Oxycodone [14]	0.026	0.959	13	126:32:40	158:10:80
PCR [9]	0.032	3.534	8	11:16:12	14:36:29
Safety Zone [50]	0.013	1.341	76	N/A	N/A
Cancer-detection via gene-edits [155]	0.016	1.637	16	1920:08:01	N/A

3.5.3 Compilation Time

We compiled the safe and unsafe assays described in the previous subsection, targeting the DropBot platform, which is a 4×15 array (ignoring I/O reservoirs on the perimeter), assuming the default electrode actuation time of 750ms. The experiments were run on a 2.7 GHz Intel™ Core i7 processor, 8GB RAM, machine running macOS™. We include a video of assay running imaging probe synthesis[3] on a DropBot device in Appendix A.1.

Construction of the the type system's *abstract interaction table* took 31 minutes running on a 2.53 Ghz Intel™ Xeon™ processor, with 24GB RAM, running CentOS 5. In this case, performance was limited to a single execution thread, as per *ChemAxon*'s documentation [34]. Constructing the *abstraction interaction table* using a multithreaded implementation of *ChemAxon* would significantly reduce construction time.

Table 3.2 reports the compilation time, constraint solving time, number of constraints gathered, time spent in the execution engine performing code generation, and total assay execution for each of the benchmarks. The unsafe assays were unable to run, so their execution times are reported as N/A. On average, each material defined in the benchmarks belonged to 3.015 distinct reactive groups; average benchmark compilation time was 0.0190 seconds; and the average time spent solving constraints was 1.594 seconds. Execution times, in these cases, depended on the assay specifications (e.g., PCR spends a lot of time thermocycling, which cannot be optimized away) and the effectiveness of the code generation algorithms.

[3]Imaging probe synthesis is a technique using radioactivity or fluorescence to tag a DNA or RNA fragment to detect complimentary nucleotide substances.

BioScript assays, along with several additional synthetic benchmarks, are made available in Appendix C.

Table 3.3: Experimental tests validating *BioScript*'s type system; parentheses denote reactive group(s) assigned to chemicals. Tests are documented incidents that could have been prevented. *I* denote Incompatible errors (dangerous) based on the EPA/NOAA reactive groups.

Name	**Result**	**Description**
AIHA 1 [7]	FAIL - I	Mix Nitric Acid (2) and Tetrachloroethylene (17,28)
AIHA 2 [7]	FAIL - I	Mix Nitric Acid (2) and Methanol (4)
AIHA 3 [7]	FAIL - I	Mix Potassium Hydride (35, 21) and Diaminopropane (7)
Mustard Gas [21]	FAIL - I	Mix Calcium Hypo (1) and Dichlor (17)
Safety Zone [50]	FAIL - I	Mix Hydrogen Peroxide (44) and Sulfuric Acid (2), then mix Acetone (19)

3.5.4 Simulation Results

As DropBot cannot execute assays that feature control flow, we evaluated the impact of our global placement techniques using a cycle-accurate DMFB in the preceding section. Table 3.4 reports the simulated execution times for several benchmarks that feature control flow. These benchmarks were compiled using the optimized global placement strategy presented in this paper, and compared against a more naive approach that compiles each basic block individually, introducing droplet transport operations when needed, at basic block boundaries [45]. Identical random number seeds were used when executing each benchmark using the two placement strategies.

The results show small, but consistent, improvements in assay execution time. The explanation is that placement can optimize droplet transport times, which are generally much shorter than the time required for mixing or heating/cooling. Although improved

scheduling could potentially reduce the latency of the scheduled operations, there is no way that existing code generation techniques, for example, could reduce the amount of time that a PCR implementation spends on thermocycling. Although the results of the optimized placement are not necessarily optimal (noting that placement is NP-complete), these results do quantify the limitations and capabilities of placement on real-world benchmarks that have been extracted from the scientific literature.

Table 3.4: The impact of the proposed global placement method in comparison to a prior approach that computes placement for each scheduled basic block in isolation [45].

Benchmark	**Global Placement** (mm:ss)	**Prior Placement** (mm:ss)
PCR w/droplet replenishment	38:16	40:44
Probabilistic PCR(full)	11:17	11:19
Probabilistic PCR(early exit)	7:20	7:21
Opiate immunoassay (positive)	399:54	405:30
Opiate immunoassay (negative)	100:16	101:48

3.6 Conclusion and Future Work

This paper has established the viability of high-level programming languages and type systems for programmable LoCs, and has properly formulated the problem of global placement, on the granularity of CFGs, for digital microfluidics. This paper reports a full system implementation, which can compile and type-check a high-level language program and execute it on the real-world DropBot platform by transmitting commands (electrode actuation sequences) via the DropBot software interface.

In the future, we hope to extend the *BioScript* language with support for non-inlined functions, arrays, SIMD operations, and some notion akin to processes or threads.

We view the type system as a starting point for a much deeper foray into formal verifications, e.g., to ensure that biological media always experience physical properties such as temperature or pH levels within a user specified range. We also plan to investigate more efficient heuristics for global placement compared to NSGA-II. Lastly, we will continually scour the scientific literature on microfluidics to find new and relevant benchmarks that exploit novel device-level capabilities, especially involving control flow. Long-term, we hope to expand the language and compiler to target a wider variety of microfluidic technologies and programmable LoC platforms.

3.6.1 Type System

Being nascent, *BioScript*'s type system statically type-checks only chemical reactivity groups. Extending the type system, introducing dependent types to account for properties such as temperature, pH, volume, or concentration is a natural next step. For example, material types can be dependent on concentration and volume. The split rule will keep the concentration the same but lower the volume. The mix rule should use an extended 4 dimensional abstract interaction table that in addition to the reactivity groups is dependent on the concentration and volume. To have a finite table, properties can be divided to ranges such as, low, medium and high concentration. However, available datasets such as chemicals/pubchem [99] do not report these properties. A large dataset is needed to calculate the dependent abstract interaction table. In conjunction with extending the type system's capabilities, providing meaningful error messages will help life scientists understand problematic portions of their *BioScript* program. Long-term, this type system could

be generalized into a generic type system for cyber-physical systems, transcending even (p)LoC-based biochemistry.

3.6.2 Compiler

We aim to support both *compilation* and *synthesis*. Compilation targets a pLoC which has already been fabricated, while synthesis converts a *BioScript* program into an optimized application-specific LoC prior to fabrication. Likewise, we aim to target two technologies: DMFBs and continuous fluid flow technologies. At present, our compiler targets a specific hardware, we aim to extend support to more devices than just DMFB devices.

BioScript enables scientists to express assays in a comfortable manner, similar in principle to laboratory notebooks. Its type system, which defines the operational semantics of *BioScript*, can provide safety guarantees when potentially hazardous chemicals are used. *BioScript* is extensible, allowing it to target pLoC compilation and LoC synthesis across multiple technologies. *BioScript* and its software stack pave the way for many life science subdisciplines to increase productivity due to automation and programmability.

$t ::=$		Terms:
|	$x \in \mathcal{X}$	Variable
|	$t_1 \oplus t_2$	Math operation
|	detect *module* on x for t	Detect
|	$v \in \mathcal{V}$	Value
$v ::=$		Values:
|	*mat*	Material value
|	r	Real number
|	n	Natural number
module $::=$		
|	$\mathsf{module}_1 \mid .. \mid \mathsf{module}_n$	Sensor module(s)
$s ::=$		Statements:
|	$i;\ s$	Sequencing
|	skip	Skip
$i ::=$		Instructions:
|	$x := t$	Assignment
|	$x :=$ mix x_1 with x_2 for t	Mixing
|	$\langle x_1, .., x_n \rangle :=$ split x into n	Splitting
|	if t then s_1 else s_2	Conditional
|	while t s	Loop
$T ::=$		Union Types:
|	$\cup \overline{S}$	Union type
|	V	Type variables
$S ::=$		Scalar types:
|	$Mat_1 \mid .. \mid Mat_n$	Material types
|	$\mathbb{R}$	Real number
|	$\mathbb{N}$	Natural number
$\Gamma ::=$		Context:
|	$\emptyset$	Empty context
|	$\Gamma, x : T$	Variable type binding
X		Set of variables x
C		Constraints

Figure 3.5: Syntax of *BioScript*'s type system.

$$\text{E-Var}\ \frac{x \in \mathsf{dom}(\sigma)}{(\sigma, x) \to \sigma(x)} \qquad \text{E-MathR1}\ \frac{(\sigma, t_1) \to t_1'}{(\sigma,\ t_1 \oplus t_2) \to t_1' \oplus t_2} \qquad \text{E-MathR2}\ \frac{v \in \mathbb{N} \vee v \in \mathbb{R} \qquad (\sigma, t_2) \to t_2'}{(\sigma,\ v \oplus t_2) \to v \oplus t_2'}$$

$$\text{E-Math}\ \frac{(v_1 \in \mathbb{N} \wedge v_2 \in \mathbb{N}) \vee (v_1 \in \mathbb{R} \wedge v_2 \in \mathbb{R}) \qquad v_1 \oplus v_2 = v}{(\sigma, v_1 \oplus v_2) \to v} \qquad \text{E-DetectR}\ \frac{(\sigma, t) \to t'}{(\sigma,\ \mathsf{detect\ module\ on}\ x\ \mathsf{for}\ t) \to \mathsf{detect\ module\ on}\ x\ \mathsf{for}\ t'}$$

$$\text{E-Detect}\ \frac{\sigma(x) \in Mat \qquad r_2 = \mathsf{detect}(\sigma(x),\ \mathsf{module},\ r_1)}{(\sigma,\ \mathsf{detect\ module\ on}\ x\ \mathsf{for}\ r_1) \to r_2}$$

(a) Evaluation rules for terms

$$\text{E-AssignR}\ \frac{(\sigma, t) \to t' \qquad t \notin \mathcal{X}}{(\sigma,\ x := t;\ s) \to (\sigma,\ x := t';\ s)} \qquad \text{E-Assign}\ \frac{}{(\sigma,\ x := v;\ s) \to (\sigma[x \mapsto v],\ s)} \qquad \text{E-Assign}'\ \frac{\sigma' = (\sigma \setminus \{x'\})[x \mapsto \sigma(x')]}{(\sigma,\ x := x';\ s) \to (\sigma', s)}$$

$$\text{E-MixR}\ \frac{(\sigma, t) \to t'}{(\sigma, x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t;\ s) \to (\sigma, x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t';\ s)}$$

$$\text{E-Mix}\ \frac{\sigma(x_1) \in Mat \qquad \sigma(x_2) \in Mat \qquad \mathsf{interact}(\sigma(x_1),\ \sigma(x_2),\ r) \neq \lightning \qquad \sigma' = (\sigma \setminus \{x_1, x_2\})[x \mapsto \mathsf{interact}(\sigma(x_1),\ \sigma(x_2),\ r)]}{(\sigma, x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ r;\ s) \to (\sigma',\ s)}$$

$$\text{E-Split}\ \frac{\sigma(x) \in Mat \qquad \sigma' = (\sigma \setminus \{x\})\overline{[x_i \mapsto \mathsf{split}(\sigma(x),\ n)]}}{(\sigma,\ \langle x_1, .. x_n\rangle := \mathsf{split}\ x\ \mathsf{into}\ n;\ s) \to (\sigma',\ s)}$$

$$\text{E-IfR}\ \frac{(\sigma,\ t) \to t'}{(\sigma,\ \mathsf{if}\ t\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2;\ s) \to (\sigma,\ \mathsf{if}\ t'\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2;\ s)} \qquad \text{E-IfTrue}\ \frac{n \neq 0}{(\sigma,\ \mathsf{if}\ n\ s_1\ \mathsf{else}\ s_2;\ s) \to (\sigma,\ s_1 \bullet\ s)} \qquad \text{E-IfFalse}\ \frac{}{(\sigma,\ \mathsf{if}\ 0\ s_1\ \mathsf{else}\ s_2;\ s) \to (\sigma,\ s_2 \bullet\ s)}$$

$$\text{E-While}\ (\sigma,\ \mathsf{while}\ t\ s_1;\ s_2) \to (\sigma,\ \mathsf{if}\ t\ \mathsf{then}\ (s_1 \bullet \mathsf{while}\ t\ s_1;\ s_2)\ \mathsf{else}\ s_2)$$

$$\mathsf{skip} \bullet s = s \qquad\qquad (i;\ s) \bullet s' = i;\ (s \bullet s')$$

(b) Evaluation rules for statements

Figure 3.6: Evaluation rules

$$\text{(T-Var)}\ \frac{x : T \in \Gamma \qquad x \in X}{\Gamma, X \vdash x : T} \qquad \text{(T-Math)}\ \frac{\Gamma, X \vdash t_1 : T \qquad \Gamma, X \vdash t_2 : T \qquad T = \mathbb{N} \vee T = \mathbb{R}}{\Gamma, X \vdash t_1 \oplus t_2 : T} \qquad \text{(T-Detect)}\ \frac{\Gamma, X \vdash x : \cup\overline{Mat_i} \qquad \Gamma, X \vdash t : \mathbb{R}}{\Gamma, X \vdash \mathsf{detect\ module\ on}\ x\ \mathsf{for}\ t : \mathbb{R}}$$

$$\text{(T-Mat)}\ \frac{\overline{mat \in Mat_i}}{\Gamma, X \vdash mat : \cup\overline{Mat_i}} \qquad \text{(T-Nat)}\ \Gamma, X \vdash n : \mathbb{N} \qquad \text{(T-Real)}\ \Gamma, X \vdash r : \mathbb{R}$$

(a) Typing rules for terms

$$\text{(T-Inst)}\ \frac{\Gamma, X \vdash i, X' \qquad \Gamma, X' \vdash s, X''}{\Gamma, X \vdash i;\ s, X''} \qquad \text{(T-Skip)}\ \frac{}{\Gamma, X \vdash \mathsf{skip}, X}$$

$$\text{(T-Assign-1)}\ \frac{x : T \in \Gamma \qquad \Gamma, X \vdash v : T' \qquad T' \subseteq T}{\Gamma, X \vdash x := v, X \cup \{x\}} \qquad \text{(T-Assign-2)}\ \frac{x : T \in \Gamma \qquad \Gamma, X \vdash x' : T' \qquad T' \subseteq T}{\Gamma, X \vdash x := x', X \setminus \{x'\} \cup \{x\}}$$

$$\text{(T-Assign-3)}\ \frac{\begin{array}{c} x : T \in \Gamma \\ t \notin \mathcal{V} \cup \mathcal{X} \qquad \Gamma, X \vdash t : T' \\ T' = \mathbb{R} \vee T' = \mathbb{N} \qquad T' \subseteq T \end{array}}{\Gamma, X \vdash x := t, X \cup \{x\}} \qquad \text{(T-Mix)}\ \frac{\Gamma, X \vdash x_1 : \cup\overline{Mat_i} \qquad \Gamma, X \vdash x_2 : \cup\overline{Mat_j} \qquad \Gamma, X \vdash t : \mathbb{R} \qquad \mathsf{interact\text{-}abs}(Mat_i, Mat_j) \subseteq \Gamma(x) \ \text{ for each } i \text{ and } j}{\Gamma, X \vdash x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t, X \setminus \{x_1, x_2\} \cup \{x\}}$$

$$\text{(T-Split)}\ \frac{\Gamma, X \vdash x : \cup\overline{Mat_i} \qquad \Gamma(x) \subseteq \Gamma(x_1), .., \Gamma(x) \subseteq \Gamma(x_n)}{\Gamma, X \vdash \langle x_1, .., x_n \rangle := \mathsf{split}\ x\ \mathsf{into}\ n, X \setminus \{x\} \cup \{x_1, .., x_n\}}$$

$$\text{(T-If)}\ \frac{\Gamma, X \vdash t : \mathbb{N} \qquad \Gamma, X \vdash s_1, X' \qquad \Gamma, X \vdash s_2, X''}{\Gamma, X \vdash \mathsf{if}\ t\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, X' \cap X''} \qquad \text{(T-While)}\ \frac{\Gamma, X \vdash t : \mathbb{N} \qquad \Gamma, X \vdash s, X' \qquad X \subseteq X'}{\Gamma, X \vdash \mathsf{while}\ t\ s, X}$$

(b) Typing rules for statements

Figure 3.7: Type checking rules.

$$\text{(CT-Var)}\quad \frac{x : T \in \Gamma \qquad x \in X}{\Gamma, X \vdash x\colon T \mid \emptyset}$$

$$\text{(CT-Math)}\quad \frac{\Gamma, X \vdash t_1 : T_1 \mid C_1 \qquad \Gamma, X \vdash t_2 : T_2 \mid C_2}{\begin{array}{c}\Gamma, X \vdash t_1 \oplus t_2 : T_1 \mid \\ C_1 \cup C_2 \cup \{T_1 = T_2 = \mathbb{N} \vee T_1 = T_2 = \mathbb{R}\}\end{array}}$$

$$\text{(CT-Detect)}\quad \frac{\Gamma, X \vdash x : T_1 \mid C_1 \qquad \Gamma, X \vdash t : T_2 \mid C_2}{\begin{array}{c}\Gamma, X \vdash \mathsf{detect}\ \mathit{module}\ \mathsf{on}\ x\ \mathsf{for}\ t : \mathbb{R} \mid \\ C_1 \cup C_2 \cup \{T_1 \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset, T_2 = \mathbb{R}\}\end{array}}$$

$$\text{(CT-Mat)}\quad \frac{\dfrac{}{mat \in Mat_i}}{\Gamma, X \vdash mat : \cup\overline{Mat_i} \mid \emptyset} \qquad \text{(CT-Real)}\quad \Gamma, X \vdash r : \mathbb{R} \mid \emptyset \qquad \text{(CT-Nat)}\quad \Gamma, X \vdash n : \mathbb{N} \mid \emptyset$$

(a) Type inference rules for terms

$$\text{(CT-Inst)}\quad \frac{\Gamma, X \vdash i, X' \mid C_1 \qquad \Gamma, X' \vdash s, X'' \mid C_2}{\Gamma, X \vdash i;\ s, X'' \mid C_1 \cup C_2} \qquad \text{(CT-Skip)}\quad \frac{}{\Gamma, X \vdash \mathsf{skip}, X \mid \emptyset}$$

$$\text{(CT-Assgn-1)}\quad \frac{x\colon T \in \Gamma \qquad \Gamma, X \vdash v : T' \mid C'}{\Gamma, X \vdash x := v, X \cup \{x\} \mid C' \cup \{T' \subseteq T\}}$$

$$\text{(CT-Assgn-2)}\quad \frac{x\colon T \in \Gamma \qquad \Gamma, X \vdash x' : T' \mid C'}{\Gamma, X \vdash x := x', X \setminus \{x'\} \cup \{x\} \mid C' \cup \{T' \subseteq T\}}$$

$$\text{(CT-Assgn-3)}\quad \frac{x\colon T \in \Gamma \qquad t \notin \mathcal{V} \cup \mathcal{X} \qquad \Gamma, X \vdash t : T' \mid C'}{\Gamma, X \vdash x := t, X \cup \{x\} \mid C' \cup \{T' = \mathbb{R} \vee T' = \mathbb{N}, T' \subseteq T\}}$$

$$\text{(CT-Mix)}\quad \frac{\begin{array}{c}\Gamma, X \vdash x_1 : T \mid C \qquad \Gamma, X \vdash x_2 : T' \mid C' \\ \Gamma, X \vdash t : T'' \mid C''\end{array}}{\begin{array}{c}\Gamma, X \vdash x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t, \\ X \setminus \{x_1, x_2\} \cup \{x\} \mid C \cup C' \cup C'' \cup \\ \{T \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset,\ T' \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset,\ T'' = \mathbb{R}, \\ \text{for each } i, j\colon\ Mat_i \in T \wedge Mat_j \in T' \Rightarrow \\ \mathsf{interact\text{-}abs}(Mat_i, Mat_j) \subseteq \Gamma(x)\}\end{array}}$$

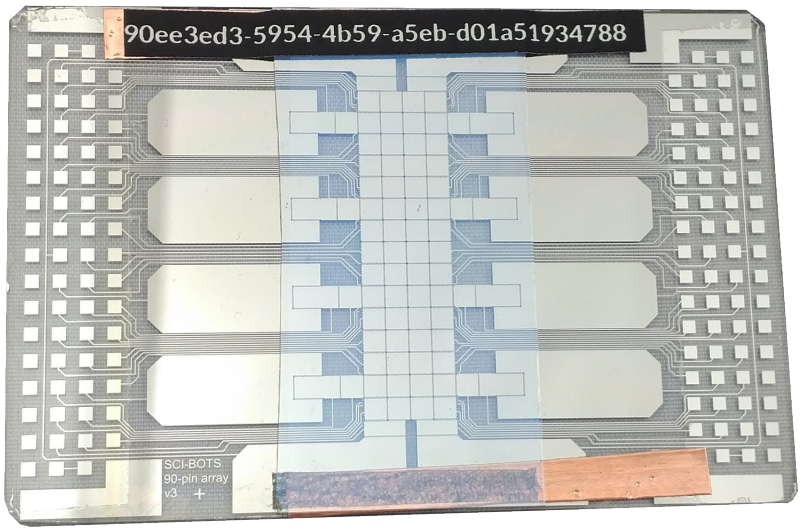
$$\text{(CT-Split)}\quad \frac{\Gamma, X \vdash x : T \mid C}{\begin{array}{c}\Gamma, X \vdash \langle x_1, .., x_n\rangle := \mathsf{split}\ x\ \mathsf{into}\ n, \\ X \setminus \{x\} \cup \{x_1, .., x_n\} \mid \\ C \cup \{T \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset, T \subseteq \Gamma(x_1), .., T \subseteq \Gamma(x_n)\}\end{array}}$$

$$\text{(CT-If)}\quad \frac{\begin{array}{c}\Gamma, X \vdash t : T \mid C \\ \Gamma, X \vdash s_1, X' \mid C_1 \qquad \Gamma, X \vdash s_2, X'' \mid C_2\end{array}}{\begin{array}{c}\Gamma, X \vdash \mathsf{if}\ t\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \\ X' \cap X'' \mid C \cup C_1 \cup C_2 \cup \{T = \mathbb{N}\}\end{array}}$$

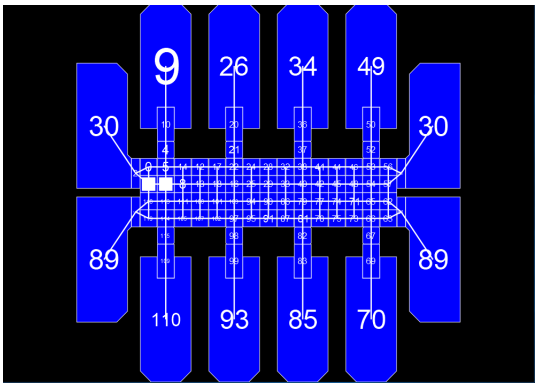
$$\text{(CT-While)}\quad \frac{\begin{array}{c}\Gamma, X \vdash t : T \mid C \\ \Gamma, X \vdash s, X' \mid C' \qquad X \subseteq X'\end{array}}{\begin{array}{c}\Gamma, X \vdash \mathsf{while}\ t\ s, \\ X, C \cup C' \cup \{T = \mathbb{N}\}\end{array}}$$

(b) Type inference rules for statements

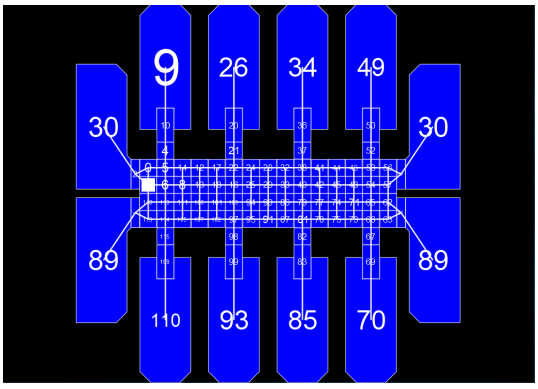
Figure 3.8: Type inference rules



(a)



(b)



(c)

Figure 3.9: DropBot device (a) and example execution, where (b) & (c) depict mixing two droplets.

```
/*Initialization Omitted*/
b.first_step();
b.measure_fluid(blood, tube);
b.measure_fluid(water, tube);
b.next_step();
b.tap(tube, tenSec);
b.next_step();
b.incubate(tube, 100, tenSec);
b.end_protocol();
```

(a)

```
/*Initialization Omitted*/
smpl := make([]*wtype.LHComponent, 0)
Bld  := mixer.SampleForTotalVolume(Blood, BldVol)
smpl  = append(smpl, Bld)
Wtr  := mixer.Sample(Water, WtrVol)
smpl  = append(smpl, Wtr)
rctn := MixInto(OutPlate, "", smpl...)
r1   := Incubate(rctn, mltTemp, InitDenatime, false)
```

(b)

```
/*Initialization Omitted*/
input s1, ip1
input s2, ip2
move mixer1, s1 ;
move mixer1, s2 ;
mix mixer1, 10 ;
move heater1, mixer1;
incubate heater1, 100, 10;
```

(c)

```
/* Initialization Omitted */
mixture = mix 10uL of water with 10uL of blood for 10s
heat mixture at 100C for 10s
```

(d)

Figure 3.10: Example assay specified using Biocoder(Fig. 3.10a)[44, 72], Antha(Fig. 3.10b)[164], AIS(Fig. 3.10c)[9], and *BioScript*(Fig. 3.10d).

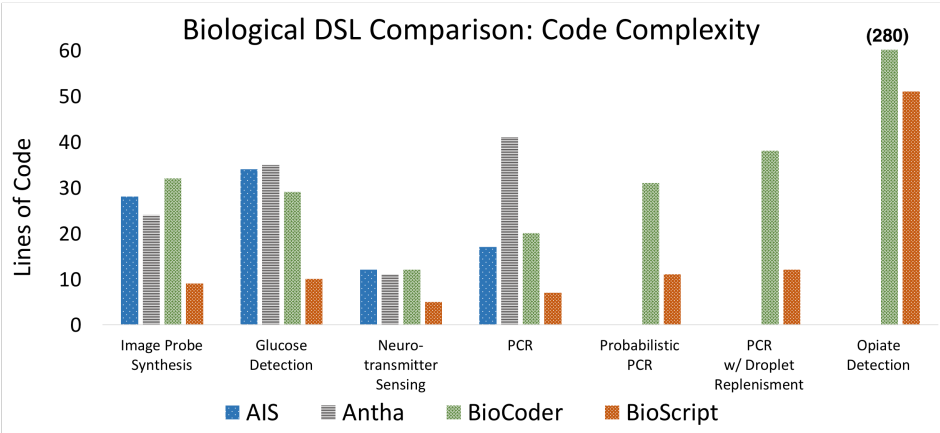


Figure 3.11: The number of lines of code to specify Image Probe Synthesis, Glucose Detection, Neurotransmitter Sensing, PCR[9], Probabilistic PCR[112], PCR w/ Droplet Replacement[92], and Opiate Detection[14, 117, 94] in AIS [9], BioCoder [11, 73, 44], Antha [164], and *BioScript*. We were unable to specify the latter three assays in AIS and Antha.

Chapter 4

Extensions to BioScript

4.1 Introduction

Digital Microfluidic Biochips (DMFBs) are poised to transform biological and chemical sciences through automation and miniaturization by reducing fluidic volume to the micro- or nano-liter scale. These devices operate by manipulating discrete fluidic droplets on a two-dimensial grid [138, 126, 70, 131, 76, 4]. Automation at the microfluidic scale accelerates and reduces waste in laboratory experimentation. DMFBs, a member of programmable laboratory-on-a-chip devices [138, 171, 9, 93, 59, 8], are taxonomically similar to Application Specific Integrated Circuits and Field-Programmable Gate Arrays. Programming a DMFB relies on the same high-level synthesis (HLS) techniques used in other reconfigurable computing domains whereby a user provides a high level description of a desired behavior and the synthesis tool generates the corresponding digital hardware matching the specified behavior. Where DMFBs differ from their reconfigurable counterparts and traditional Von

Neumann architectures is the lack of a memory hierarchy, leaving no ability to store data in off-chip memory.

Synthesis frameworks targeting DMFBs use either static [74, 75] or just-in-time [73, 177] (JIT) compilation techniques. JIT frameworks targeting DMFBs include support for the use of functions in their high level specification; a feature not currently supported in static frameworks. However, the use of JIT compilation can lead to programs that halt mid-way through execution, wasting time and resources. This outcome is easily avoided by using static synthesis techniques to determine if a target architecture is capable of supporting the specification *before* execution.

A DMFB operates by mapping operations onto portions of the device for each timestep of execution. This behavior allows for any number of operations to be mapped at any given timestep given the sum of the areas of the operation don't exceed the area available for computation. This property allows them to run many operations in parallel. The HLS tools handle the scheduling and placement of the operations. Currently, no programming language targeting DMFB devices[177, 133] support parallelization techniques within the programming language itself. They rely on the HLS tools to handle parallelization for them.

This paper describes extensions to a leading static synthesis framework [74] which provides support for functions within the framework and a domain specific language for programming DMFB devices to include support for Single Instruction Multiple Data (SIMD) operations. By cleverly applying data flow techniques to the compilation and synthesis process, the static framework overcomes the issues of wasted time and resources that plague dynamic approaches.

In 4.2 we describe how the system works, followed by its implementation in 4.3. We conclude this paper by discussing future research directions for this system in 4.5.

4.2 Overview

Synthesizing an executable for a DMFB device is comprised of two phases: the *compilation phase* is responsible for classic compiler techniques while the *synthesis phase* synthesizes the input program into an electrode activation sequence. This process is depicted in Fig. 4.1.

4.2.1 Compilation Phase

In phase 1, the compiler converts an input program into SSI form, builds a Control-Flow Graph (CFG), and runs various data-flow analysis: variable liveness analysis, reaching definitions, and context sensitive call graph analysis. As DMFBs manipulate discrete droplets, the idea of a pointer is obtuse: there is a one-to-one mapping between addresses (droplet location) and their values (droplets). As the program is in SSI form, context insensitive analysis is sufficient to calculate space requirements necessary to store droplets alive across function calls [78, 28], except for cases when head-recursion is present, which we discuss below.

4.2.2 Synthesis Phase

During the second phase, we attempt to synthesize an executable as an electrode activation sequence from the CFG provided from phase 1. The CFG is parsed into into a series of Directed Acyclic Graphs (DAGs), each of which is synthesized separately. The synthesis phase

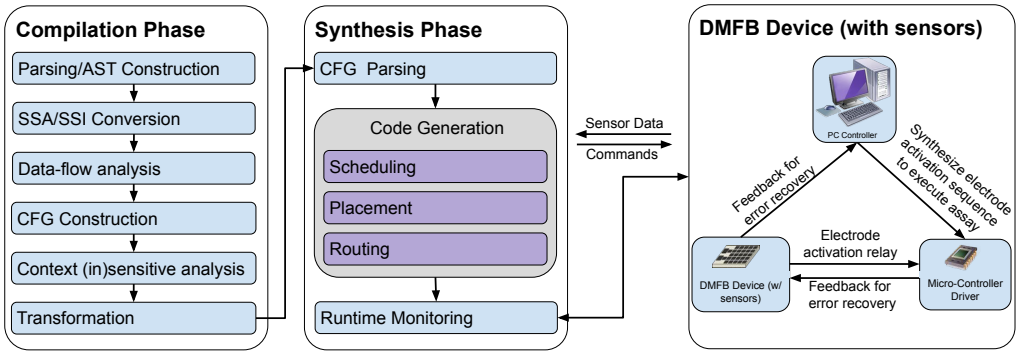


Figure 4.1: Synthesis is split into two phases: the *Compiler* phase is responsible for traditional compilation techniques: transformation into SSI form, data-flow analysis, CFG analysis, etc. The *Synthesis* phase translates an intermediate representation to the necessary *electrode activation sequence* provided it can solve the scheduling, placement, and routing problems.

proceeds to solve the scheduling, placement, and routing problems discussed in Chapter 2 by first converting the CFG-linked DAGs into the schedule of operations. After scheduling completes, placement begins, binding modules (mixers, detectors, etc.) to particular chip regions. When a valid placement has been determined, the router attempts to route the necessary droplets to their appropriate modules, as depicted in Fig. 2.6. The synthesis phase creates an executable in the form of an electrode activation sequence corresponding to the droplet manipulations that encode the high level description of the program.

4.3 Compiler Design

To help detail the algorithms used, we explore our implementation with the aid of an example, depicted in Figs. 4.4 to 4.6. In general, any language supporting the DMFB ISA can be compiled and run on a DMFB. As pointers are not supported by DMFBs there is no need for resolving points-to analysis, as there are no function pointers nor any notion of

dynamic dispatch. To begin, the program is converted into SSI form where a combination of context insensitive and sensitive analysis occurs.

An input program, Fig. 4.4, details an arbitrary program targeting a DMFB utilizing functions. As shown in Fig. 4.4a, the program begins with a series of dispenses, denoted in lines 2-7. We assume that, prior to line 8, all of these droplets are alive on the chip. In line 8, the program invokes function foo, defined in Fig. 4.4b. bar is subsequently invoked in line 9, which also invokes foo. The program finishes with several mixes (lines 10 & 11).

4.3.1 Context Insensitive Call Graph Analysis

In order to determine if a program has adequate space to execute on a device, the compiler uses context insensitive analysis to calculate live ranges and allocate storage for variables live across function calls; because the program is in SSI form as discussed in Chapter 2, there are optimal linear time solutions [27, 28]. In Fig. 4.4, the variables alive across foo (line 8) are $\{main_a, main_b, main_c, main_f\}$. The variables alive across bar are $\{main_a, main_b, main_c\}$; however, due to the imprecise nature of context insensitive analysis, the chip must be able to support storage of the maximum number of live variables across all function calls, including nested calls. In the case of this program, that is 4. Using a chip size of 5×8, as depicted in Fig. 4.6, this program is able to be executed on the device.

4.3.2 Context Sensitive Call Graph Analysis

Context sensitive analysis is required in order to statically synthesize the electrode activation sequence that is correct with respect to the input program. Fig. 4.4 demonstrates a program that necessitates context sensitive analysis. On line 2 of Fig. 4.4b there is an I/O operation: c is assigned from an input fluid dispensed from the edge of a device. foo is called from two different locations: lines 7 and 8 in Figs. 4.4a and 4.4c, respectively. The interprocedural control path originating from line 8 of main, depicted by the dotted lines in Fig. 4.5 might yield the placement and routing plan depicted in Fig. 4.6a. Unfortunately, as a result of the dispense present in foo, this cannot be blindly copied and repeated throughout the execution of the program.

The dashed lines in Fig. 4.5 depict the interprocedural control path originating from line 9 of main. This calls bar which may subsequently call foo. Because bar is called *before* foo in this calling context, Fig. 4.6b depicts a valid placement and routing plan generated by the synthesis phase. In more complicated programs, the shape as well as the location of basic blocks and their modules may change as a result of differing calling contexts.

4.4 SIMD Semantics

Flynn's taxonomy describes several classifications of hardware architectures[52, 60]. One classification, Single-Instruction-Multiple-Data (SIMD), manipulates multiple data elements with a single instruction; exploiting data-level parallelism. In traditional computation models, SIMD instructions lend themselves to image-processing and other multimedia applica-

tions, e.g., changing the brightness level of an image; where a constant value can be added or subtracted from rows of pixels at a time.

In the life sciences, a practitioner may wish to react a sample with a range of different concentrations. This is analogous to applying a constant to a row of pixels in an image. The scientist, hoping to understand how a protein sample denatures in the presence of various concentrations of acid, as presented in Fig. 4.2a, historically, would have to write a *BioScript* equivalent to that of Fig. 4.2b. DMFB devices, being reconfigurable, lend themselves to parallelization. Thus, SIMD instructions can both increase the complexity of assays that *BioScript* is capable of expressing while decreasing the overall size of a *BioScript* program by 80%, as depicted in Fig. 4.2c.

To model SIMD semantics in *BioScript*, we leverage techniques from array programming [172]. Each variable is an array implicitly and the compiler infers SIMD operations using inferred context from the instruction, as depicted in Fig. 4.3a. This allows the scientist to write assays exploiting SIMD semantics without having to explicitly use them. On line 3 of Fig. 4.3a, the *BioScript* compiler is able to automatically mix 2 droplets of water with both values (in this case water) stored in y. *BioScript* also supports direct indexing, as expressed in lines 3 and 4 of Fig. 4.3b. Both pieces of code in Fig. 4.3 are semantically identical.

This is largely a usability construct that affords an easier syntax for scientists unaccustomed to programming. Allowing scientists to express more complicated and that are more representative with their traditional laboratory workflow.

4.4.1 Implementation

We extend [46], a Java-based SSI-form compiler for DMFB devices, to support both static compilation of programs utilizing functions and SIMD semantics. Context insensitive analysis uses the algorithm described in [28]. The required IIG is built and optimally colored. Thus, if the coloring exceeds the number of regions capable of storing fluids, then the compilation halts, notifying the user that the target device cannot execute the input program, saving time and resources; otherwise, compilation continues onto the synthesis phase. For context sensitive analysis the compiler uses the functional approach [149], choosing accuracy at the expense of time, noting that the size of existing programs are not much larger or more complex than Fig. 4.4[1]. If analysis discovers a head-recursive function, in which droplets created within a function remain alive across recursive calls,detected by the cardinality of the liveness set increasing, that calling context is marked as requiring JIT compilation.

The synthesis implementation extends MFSim [74], a C++-based DMFB synthesis framework, to include full support of recursion. Using the context sensitive analysis, we must determine different placement and routing plans for each calling context, as they may differ for each calling context, as noted in 4.3.2. Recognizing that statically computing precise recursion depth is intractable even in traditional architectures, we rely on MFSim's JIT runtime to handle programs featuring unbounded head recursion. Thus, where there are no cases of head recursion this solution uses statically-compiled CFGs. If there are

[1]To further understand the complexity of chemical experiments, we refer the reader to the supplemental material produced in [133]. Which detail experiments that are representative of what scientists conduct in a cookbook-style syntax.

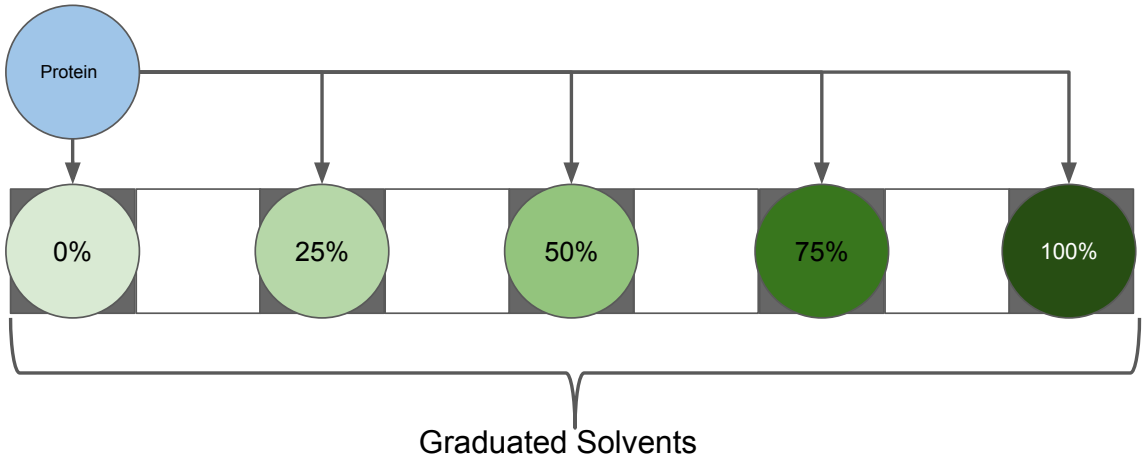
instances of head recursion the synthesis uses statically-compiled CFGs where able and JIT compilation at runtime to resolve the head recursion.

As DMFBs are nascent, the complexity of standardized chemical experiments is minimal, not exceeding trivial conditional statements, like that of Fig. 4.4[1]. There are, to the best of our knowledge, no experiments that necessitate the use of functions, much less recursion. We do acknowledge that several assays might benefit from a reduction in assay size should they employ functions. Reporting the results of synthetic assays do not sufficiently demonstrate the efficacy and potential this implementation yields other than to demonstrate functionality. In any event, reporting results amounts to providing an electrode activation sequence for an experiment or reporting failure; as we make no claims about performance or efficiencies.

4.5 Conclusion

DMFBs stand to accelerate and reduce waste in laboratory experimentation through automation and miniaturization. While the current complexity of uses on DMFBs do not require the usage of high level abstractions such as functions, more complex uses will require the reworking of current synthesis techniques to allow for abstractions which enable users to specify their usage clearly. This paper describes an implementation that statically compiles and synthesizes DMFBs from high level descriptions utilizing functions, providing for a clear path for specifying more complex experimentation and research and introducing SIMD semantics, allowing programmers to exploit the inherent parallelism endemic to

DMFB devices. Current DMFB architectures lack I/O channels native to traditional Von Neumann architectures, which poses interesting future work which could ease scheduling and placement during synthesis and allow for the completely static synthesis of head-recursive functions should off-chip storage be manifested.



(a)

```
// Initialization omitted
// Prepare the 25% and 75%
half = mix acid with buffer
temp = split half into 2
twenty_five = mix temp[0]
   with buffer
seventy_five = mix temp[1]
   with acid
fifty = mix acid with
   buffer
zero = dispense buffer
one_hundred = dispense acid
// Denature the protein
p_0 = mix protein with zero
p_25 = mix protein with
   twenty_five
p_50 = mix protein with
   fifty
p_75 = mix protein with
   seventy_five
p_100 = mix protein with
   one_hundred
```

(b)

```
// Initialization omitted
ranges = gradient acid with
    buffer range 0, 100 at
   25
denature = mix protein with
    ranges
```

(c)

Figure 4.2: Fig. 4.2a depicts a scientist who wishes to observe how a protein denatures in the presences of differing concentrations of acid. Without SIMD operations, a scientists would have to write the code depicted in Fig. 4.2b. Fig. 4.2c illustrates how SIMD operations simplifies the same experiment. Reducing the number of lines of code from 12 to just 2.

```
1 x = dispense water
2 y = split x into 2
3 z = mix y with water
```

(a)

```
1 x = dispense water
2 y = split x into 2
3 z = mix y[0] with water
4 a = mix y[1] with water
```

(b)

Figure 4.3: Two programs, Fig. 4.3a depicts *BioScript*'s SIMD semantics, and Fig. 4.3b depicts an assay using direct indexing. These two programs are semantically identical.

```
main() {
    a = ...
    b = ...
    c = ...
    d = ...
    e = ...
    f = ...
    g = foo(d, e)
    h = bar(f, g)
    i = mix(a, h)
    j = mix(b, c)
}
```

(a)

```
foo(a, b) {
    c = ...
    if (...) {
        m = mix(a, c)
        dispose b
    } else {
        m = mix(b, c)
        dispose a
    }
    return m
}
```

(b)

```
bar(a, b) {
    c = mix(a, b)
    if (...) {
        return c
    } else {
        d = ...
        e = foo(c, d)
        return e
    }
}
```

(c)

Figure 4.4: A simple program targeting a DMFB device. The mix function combines the two fluids passed as an argument. Every variable declaration triggers an I/O request, drawing liquid from a reservoir, except for variables declared as a result of the mix operation.

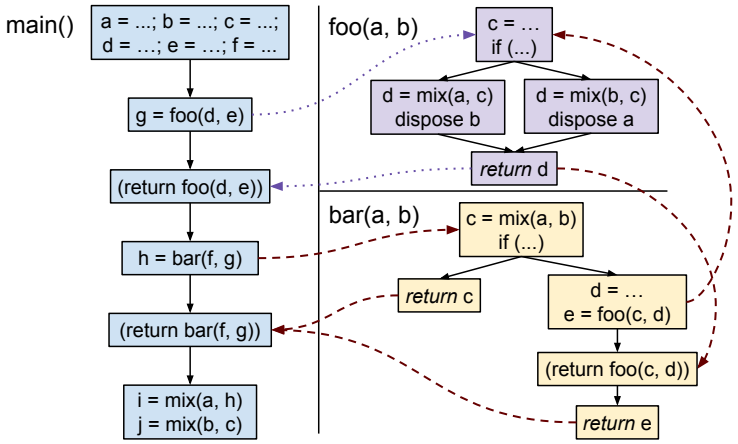
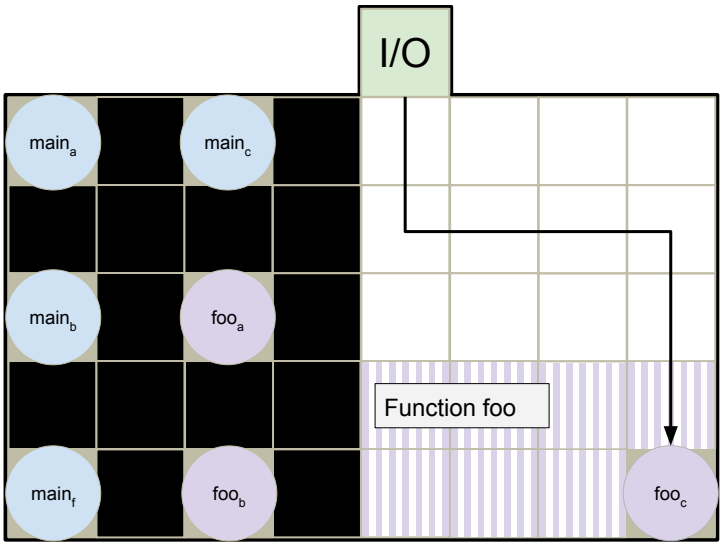
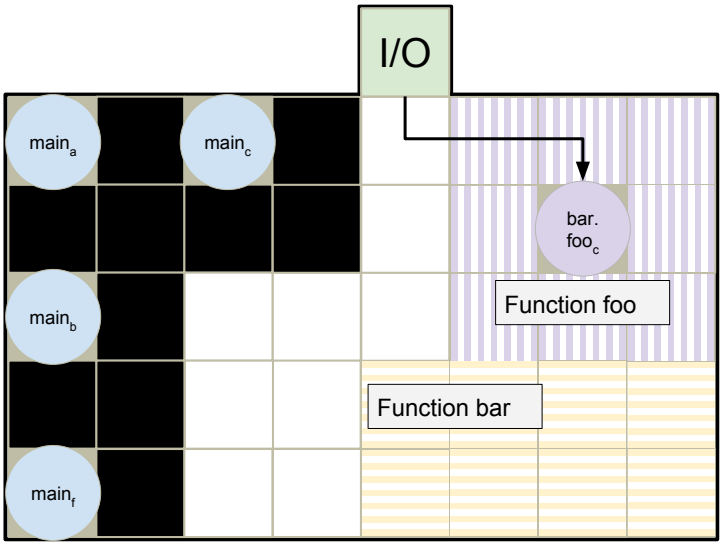


Figure 4.5: The abbreviated context sensitive call graph corresponding to Fig. 4.4. For brevity and clarity, we omit explicit procedure entry and exit points, only denoting return of control from a procedure being called and don't include call string information on edges. As this is context sensitive, a call path cannot traverse both dotted and dashed lines.



(a)



(b)

Figure 4.6: Fig. 4.6a depicts where function foo might initially be placed when called from line 8 in Fig. 4.4a. However, because different calling contexts will yield different chip allocations, calling foo from a different location may force foo to be placed in a different location. In this case, routing must be aware of the changes to placement. Fig. 4.6b depicts what the calling context placed onto the chip would look like at line 9 in Fig. 4.4a. bar is placed where foo was originally placed. Routing must now move the fluidic variable defined in line 2 of Fig. 4.4b to foo's new placement, denoted by the vertical pinstripes. The black electrodes, or registers, are used for storing fluids. They cannot be used for general computation.

Chapter 5

Targeting Continuous-Flow Microfluidic Devices

5.1 Introduction

The design and fabrication of microfluidic devices for use in experimental micro-biology is limited to those with the requisite domain expertise spanning a multitude of disciplines: biology, chemistry, physics, material sciences, and microfabrication techniques, to name a few. The expertise required creates a barrier restricting adoption of microfluidic devices to a few specific applications in academia and industry; even in spite of the technological advances microfluidic devices have experienced.

Historically, the design of flow-based laboratory-on-a-chip (LOC) devices has been by hand, using computer-aided design (CAD) tools to draw a device and then fabricating the design using one of the myriad of fabrication techniques available — a laborious task

requiring significant investment from the scientist for even the simplest of devices. There are two basic categories that describe the designers of microfluidics devices: component and system designers. Component designers explore how fluidic phenomena can be exploited to create new components to achieve new goals or old ones more efficiently. In contrast, system designers explore building large-scale microfluidic devices typically involving hundreds of components and channels. For continuous flow devices, those which rely upon an external pressure or vacuum to drive fluids, system designers have physical constraints to suffice, e.g., maintain an equal relative resistance across the device, otherwise the device may not function as intended. Both size (number of components and connections) and medium (fluidic properties used in the device) further compound complexity and difficulty for system designers to design and fabricate functional devices.

This paper introduces Xylograph, a toolchain designed to reduce the complexity and difficulty system designers face designing and fabricating new microfluidic devices. Xylograph applies concepts native to Computer Science — programming languages, compiler theory, and architectural synthesis — to automate the design and fabrication of microfluidic devices Xylograph is used to design an assay using a cook-book-style programming language. Xylograph uses that program to automatically design the corresponding device that maintains semantic fidelity to the user's assay; which amounts to automatic selection and placement of the required components used in the assay and the routing of the connections between those components. Xylograph makes it easy for experts and amateurs alike to automatically design and fabricate microfluidic devices, including devices that would otherwise be intractable to design by hand.

In this work, we showcase Xylograph's capability at removing the barriers a system designer faces in designing and fabricating microfluidic devices; enabling amateurs to leverage microfluidics. We use Xylograph to automatically design and fabricate 2 microfluidic devices that perform common tasks like mixing reagents or generating reagent concentrations. To demonstrate the time saved, we measure the time it takes to express assays of components ranging from 4 to 1,024 components in Xylograph against estimates from a bioengineer to design the same devices by hand.

5.2 System Design

The discrete steps of Xylograph are depicted in Fig. 5.1, with the architecture workflow described in Fig. 5.3. Xylograph is a series of extensions to prior work easing the burden system designers face when designing new microfluidic devices.

5.2.1 Designing an Assay

A program is composed of a series of instructions conforming to the syntax of a particular programming language. These instructions, before they can be executed on a processor, must be compiled. A compiler translates the programmers code into something the processor is capable of understanding. Once the compilation is complete, the program can then be executed and the desired output or behavior can be observed.

Designing a microfluidic device and writing a program share many similarities: the designer begins with an assay, a series of instructions culminating in some measurable output. The steps range from mixing chemicals, to heating a reagent, to measuring the

pH of reagents, the steps are discrete and clear. Those steps, in the context of microfluidic devices, typically require the use of a component. Thus, when designing a microfluidic device, a designer will place the necessary components and route channels between the components. It can be said that an operation is "executed" when the fluid has moved from the input(s) of a component to it's output(s) — or computation has occurred.

Because of these similarities there have been several attempts at developing programming languages targeting microfluidic devices. These languages, however, only aim to program digital microfluidic devices, doing nothing to address the complexities of designing and fabricating a continuous-flow device from a specification. While these languages might be effective at programming a device, they tend to be arcane and/or tied to a specific device, in some cases necessitating expertise in Computer Science.

For instance, *BioCoder*[44], *Puddle*[177], and *BioScript*[133] all target specific programmable microfluidic architectures. While *BioStream*[171, 166] and *Aqucore Instruction Set* (AIS)[9, 8] are focused on specific customizeable wet-ware architectures. However, the range of expertise required to *use* the languages ranges from the esoteric corners of Computer Science (*BioStream* and *AIS*) to user-friendly (*Puddle* and *BioScript*). Further, these languages are restricted to their specific microfluidic architectures, unable to support the automatic design of microfluidic devices.

In order to federate languages from their specific architectures, computer scientists utilize compilers. A compiler takes a high-level language (e.g., Python or C++), and translates them into sets of low-level *machine code* — instructions a physical architecture is capable of understanding. This process is illustrated in Fig. 5.3 (left of the dotted line),

in which a user provides an input program, written in C++ or Python in this case, and the compiler translates the input into an *intermediate representation*, and then generates the machine code that is specific to the processor on which the program will execute. The translation into the *intermediate representation* might seem unnecessary; however, if it did not exist, there would need to be a mapping from each language to the specific processor architecture. In other words, the translation to an *intermediate representation* introduces a common denominator that allows computer scientists to build one compiler that is, in principle, capable of accepting any language as input and is capable of targeting any processor architecture. *Intermediate representations* also allows a myriad of standardized optimizations to the input code that typically reduces execution time and better utilizes computational resources. Once optimizations occur, the compiler can translate the *intermediate representation* code into the machine code for the processor architecture(s) desired for execution.

Recent research discuses the creation of microfluidic compilers[45, 133, 74]; however, these endeavors target digital microfluidic devices. Interest is lacking in leveraging compilers for continuous-flow devices, much less conjoining compilers with that of design-automation tools.

The *BioScript* language and compiler, (1) in Fig. 5.2 is extended to include a target for continuous flow devices, via the ParchMint[119] interchange format; creating the workflow represented in Fig. 5.3 (right of the dotted line). Part of this extension includes component selection and netlist generation. For this experiment we use the ParchMint microfluidic netlist standard as an interchange format between *BioScript* and Inkwell [43].

5.2.2 Component Selection & Generation

A system designer might begin designing a new device by first selecting the required components. However, those components might require changes as the original component might have been designed with different fluidic properties in mind.

Like the designer, the compiler selects the component that best matches the function of the operation. For Xylograph to correctly map *BioScript* instructions to their corresponding component, it employs a direct mapping of instruction to pre-generated components. For instance, given the mix instruction in line 7 of Fig. 5.1a, the compiler maps all mixes to a pre-generated serpentine mixer. Component selection is parameterized such that if a time definition exists on the mix instruction, component selection will attempt to select the best match; opting to round up when necessary. There is nascent work involving automated and parameterized component generation[146], however, as the *BioScript* compiler currently lacks necessary information, automated component generation is left for future work.

Once components are selected, the compiler builds the connections between these components as a netlist (Fig. 5.1b). A connection is defined as a channel between a component that defines a *variable* (for example, line 5 in Fig. 5.1a defines the variable *acid* with the value *hcl*) and any *uses* of that variable (for example, line 7 in Fig. 5.1a, the variable *acid* is being used in the mix instruction, but is not defined in the statement). *BioScript*'s compilation process finishes by generating a netlist — it's *machine code*.

The netlist represents an abstract description containing what components the device should have and how those components should be connected. This abstract netlist is

not sufficient for fabrication as it does not contain concrete information about where components should be located or what routes connections should take to connect two components. In order to convert an abstract netlist into a concrete netlist containing this information we need to perform two steps: placement and routing.

5.2.3 Designing a Device

Once the system designer has a conceptual idea of the assay and the necessary components they can begin designing the physical device. To do so, they must solve 2 problems considered by computer scientists to be intractable: placement and routing. A designer may be able to solve both of these problems on simple or small designs, those having 5-10 components. However, for devices comprised of hundreds or even thousands of components, this quickly becomes overwhelming; as complexity increases designing by hand is not viable. The only viable way to design and fabricate these devices is by automating the design and fabrication processes.

Since *BioScript* is only concerned with the conversion of a high level description into an abstract netlist, another tool, Inkwell[120], is utilized to perform the synthesis phase: finding a valid placement of components and routing channels between them.

Placement

Two different methods, Planar Placement (Figs. 5.1c to 5.1e) [121] and Directed Placement (Figs. 5.6b and 5.6c) , are used in generating results for this publication. Both of these methods are designed to obtain a placement of components within a 2D plane such that no

components overlap, optionally including an additional user-defined buffer distance between components to facilitate fabrication.

Planar Placement seeks to find a component placement which will support a planar routing, one where no two connections intersect; which is necessary to generate a valid single-layer device with no incidental fluid interactions. Intersection can be dealt with in 2 ways: at any intersection a valve can be placed there to switch which fluid channel is currently flowing or the device can contain multiple levels allowing fluids to route above/below each other. Xylograph, in principle could support either of these methods. Noting that either method dramatically increases the complexity of fabricating and/or operating the device. The Xylograph methods used here only support the design and fabrication of planar device, and all other non-planar inputs are designated as invalid before placement begins.

Determining planarity occurs by utilizing the Chrobak-Payne straight line planar embedding algorithm [37] to find a layout where all connected components can be connected using a straight line. Unfortunately the Chrobak-Payne method is only capable of embedding single points and not components with varying sizes, necessitating additional processing. After initial coordinates are found space is inserted such that sufficient space exists for the coordinate to represent the upper left of the component without introducing any overlap between neighboring components. There are a number of different ways that this space can be inserted, however for this publication we use the naive method [42].

Directed Placement uses an assumed flow paths in the netlist to generate a superior design for certain types of devices. This method starts with the fluid input components and places them on the left-hand side of the device. It then traverses each connection from the

input components and as it does so places each new component it finds in a subsequent lane, so components that are 1 connection from an input are in lane 2, components 2 connections away are in lane 3, etc. After it finds initial lanes for each component it performs a number of optimizations to distribute the components so they do not overlap and they are well aligned with their neighboring components.

Routing

Once a valid placement has been found routes need to be determined to connect the components as described in the netlist, depicted in Fig. 5.1d. In order to create a set of routes that introduce no intersections with components or each other we use the Planar Routing [122] method. This method creates a grid of nodes for each point of unused space (space within the bounds of the device that does not contain a component) which represents the available paths for a route. Using a network-flow based routing algorithm route paths are found between a component and all other components to which its connected, and that process continues for each component until a path for every connection has been found. When a path is found the nodes it uses are removed from consideration by other paths so that they cannot re-use or cross an existing route path. In the case that a valid route, i.e. one that does not intersect another route path, cannot be found then all the paths found so far are removed and the process beings again with the connection which could not be routed going first. In this way connections which move through the most congested part of the device are routed first and connections with more possible choices are routed last.

5.2.4 Device Fabrication & Assay Execution

Once a design is created, either by hand or using design automation tools, it must be reviewed by a domain expert. A domain expert must ensure the geometries, channel widths, etc. are correct with respect to the specification and that the device will operate relative to fluidic attributes. We expect, future research into component generation and placement and routing to address this shortcoming. However, because Inkwell generates an Scalable Vector Graphics (SVG), there is no real time investment for a domain expert. The SVG, in this context, is then sent to Computer-Aided Design (CAD) software, where the device can then be simulated and fabricated using any of the techniques mentioned above. Since *BioScript* nor the netlist do not capture physical attributes of fluids, most devices generated require review from a domain expert.

In principle, the design generated by Xylograph can be fabricated using any popular fabrication method, e.g., Computer numerical control (CNC) milling[101, 102] or photolithography. For the purposes of this work, we use a CNC mill (Bantam Tools© Desktop PCB Milling Machine) to mill a negative in poly(methyl methacrylate) (PMMA, or "acrylic" plastic)[54, 85, 136]. We then bond a glass layer to the PMMA layer. Once the bond has cured, I/O ports are then connected to their respective reservoirs and the device is ready to be used, Fig. 5.1e. The devices are capable of either being powered by gravity-induced head pressure or, as in our case, external syringe pumps.

5.2.5 Application Mapping

Application mapping for passive flow devices amounts to generation of a Directed Acyclic Graph (DAG), selection of components that best match operations and then executing the assay on the device, a process discussed in more detail in earlier in 5.1. Mapping an assay onto a programmable, active flow device is more complicated: while the process remains the same, in principle, an additional step between component selection and execution is inserted into the process: application mapping[125]. This is analogous to the code generation phase of a traditional compiler; where the compiler schedules operations, binds values to registers, and emits processor-specific machine code. However, in this context, it is binds resources to components, using the device's valves to move fluid to the appropriate component.

BioScript generates an activation sequence like that detailed in Table 5.1. This table maps the resources to a component for a given time step. This ultimately results in the valves being activated at a given timestep, as illustrated in Fig. 5.4. *BioScript* implementation of relies on previous work described in [125]. At each time step a resource is scheduled and bound to a component. After this, a path from origin to destination is computed. If a route cannot be found, the assay can not be executed on the device — application mapping fails.

Unfortunately, at time of writing, Inkwell is incapable of supporting device design and fabrication utilizing valve-based pLoC devices. Xylograph will generate the activation sequence based on the timing information provided by the user. The activation sequence can be trivially translated to a tool like Labview©, an industry standard tool capable of physically actuating the valves in accordance with the activation sequence generated by

Table 5.1: The activation sequence at each time step for each valve required to mix two fluids in a circular mixer.

Time Step	**Open**	**Closed**	**Operation**
1	$\{cv_1, cv_3, cv_4, cv_5, cv_6, cv_7\}$	$\{cv_2\}$	Load Fluid 1
2	$\{cv_2, cv_3, cv_7\}$	$\{cv_1, cv_4, cv_5, cv_6\}$	Load Fluid 2
3	$\{cv_4, cv_6\}$	$\{cv_1, cv_2, cv_3, cv_5, cv_7\}$	Mix
4	$\{cv_5\}$	$\{cv_1, cv_2, cv_3, cv_4, cv_6, cv_7\}$	Mix
5	$\{cv_4, cv_6\}$	$\{cv_1, cv_2, cv_3, cv_5, cv_7\}$	Mix
...	$\{...\}$	$\{...\}$	...
n	$\{cv_5\}$	$\{cv_1, cv_2, cv_3, cv_4, cv_6, cv_7\}$	Mix

Xylograph. It should be noted: this is a naive and simple mapping, it does not account for timing issues pertaining to fluidic properties such as viscosity. We leave all extensions involving fluidic properties for future work.

5.3 Results & Discussion

We measured the time it took to program an assay and synthesize a corresponding device design. These assays are arbitrary on purpose; only including primitive operations found in many lab-on-a-chip devices (mix, split, and dispose/dispense). For the purposes of these experiments we assume standard components, e.g., all mix modules behave the same in form (two input, one output), or channel splits can only split into some power of 2 (so one channel can split into two channels, four channels, eight channels, etc.).

We generated Table 5.2 using an Amazon AWS™instance comprised of 8 vCPUs running at 2.2 Ghz and 64 GB of RAM. We acknowledge that these results are in a vacuum — free from comparison of how traditional microfluidic devices are created. We have been unable to find any quantitative information regarding the time required to design and

fabricate these devices. In light of that fact, we consulted with Bioengineers to estimate how long it would take them to build each of the devices in question; which we report in Table 5.2. The 1024 component device is a conservative estimate from Bio-Engineers working with microfluidics. Using Xylograph reduces the amount of time a system designer must invest in the design of a device by 78%.

To empirically prove that Xylograph can reduce the barriers for system designers, we used Xylograph to design 2 devices that perform common tasks like preparing a range of solute concentrations. We also use one of the fabricated devices to automate the steps involved in a standard dilution experiment. We discuss the experimental setup and outcome for each of the devices in 5.3.1 and 5.3.2.

Table 5.2: Results demonstrating the time it takes to express an assay (using *BioScript*) and then using Inkwell to synthesize the corresponding device. The results clearly demonstrate how well this workflow performs at building successively more complicated devices; a task that, when done by hand is exceedingly arduous and onerous. Xylograph is shown to reduce human design time by an average 22%.

	Design time using Xylograph (hh:mm:ss)			Design time by hand (hh:mm:ss)	
Number of Components	Write Assay	*BioScript* & Inkwell	Total	Estimated Total	Xylograph's Time Savings
4	0:26	0:01	0:27	2:00	78%
6	0:45	0:01	0:46	5:00	85%
16	2:48	0:05	2:53	10:00	72%
32	3:03	0:17	3:20	15:00	80%
64	6:49	1:11	8:00	30:00	77%
128	12:36	4:52	17:29	1:00:00	79%
1024	1:42:09	6:22:25	8:04:34	7:00:00	76%

5.3.1 Small Dilution Mixer — the "Mini-Mixer"

We tested our "small" device, depicted in Fig. 5.1e, a CNC-milled PMMA device sealed with Polymerase Chain Reaction (PCR) tape. We used two 50mL fluids colored with 10 drops of blue dye and 60 drops of yellow dye. The colored fluids were pumped through the device at a rate of 0.5ml/min.

We expected the "Mini-Mixer" device to generate a 50% dilution of the yellow and blue colored fluids — in this case we expect to create green water. To confirm this dilution, we took images of the resulting outputs using an off-the-shelf high-resolution camera and using ImageJ, an industry standard image processing tool, we created the RGB histograms Fig. 5.5. The histogram in Fig. 5.5c shows that Xylograph's generated device produces a fluid with the appropriate color profile, the water is green. The histogram for the blue water, Fig. 5.5b, has a large presence of green. To the naked eye, Fig. 5.1e appears slightly cyan — which does includes some amount of green. Figure 5.5c also exhibits a large presence of red. These variances could be a result of differing densities of the dye or imperfections introduced into the device during milling.

5.3.2 Medium Dilution Tree

While a tiny mixer is an intersting proof-of-concept, we wanted to demonstrate Xylograph's ability to produce devices that can reproduce meaningful experiments. To do this, we use Xylograph to build a dilution tree — a standard experiment allowing scientists to quickly generate a series of reagents with ranging concentrations, as depicted in Fig. 5.6. Like the

"Mini-Mixer", described in 5.3.1, this device was fabricated using a CNC-milled PMMA device sealed with PCR tape. Our two inputs are are 50mL of water colored with 10 drops of blue dye and 50mL of water colored with 60 drops of yellow dye. The two fluids were driven through the device using a syringe pump at a rate of 0.125mL/min.

To obtain visual measurements, we used an off-the-shelf high-resolution camera and used ImageJ to measure the RGB values in the resulting solutions. Figure 5.7 depicts ImageJ's generated histograms. In Figs. 5.7a and 5.7b demonstrate the inputs as expected — yellow (Fig. 5.7a) and blue (Fig. 5.7b). The outputs, Figs. 5.7d to 5.7g, demonstrate the expected output for the device. The histograms in Figs. 5.7d and 5.7g show the correct outputs, 100% of both yellow and blue, respectively. The histograms in Figs. 5.7e and 5.7f correlate with the droplets of their respective position in Fig. 5.7c. These histograms show the presence of both green and red in the resulting mixtures, but in different ratios, which demonstrates reasonable rates of mixtures for the expected concentrations. The lack of perfect distributions in the outputs could be a result of milling imperfections or different densities in the dyes used to color the water. To determine the exact concentrations, or in this case, presence of colors, in Figs. 5.7e and 5.7f would require the use of spectroscopy — a tool not readily accessible to computer scientists.

5.4 Future Work

Xylograph is flexible and pliable: there are many directions one can choose to further develop this idea and tool. One orthogonal research avenue would be exploring component reuse and reduction. Current devices are not large, spatially; however, they can be comprised

of dozens of components. Designing these devices doesn't explicitly identify components capable of reuse. Automatically detecting and reducing components can reduce design and fabrication time. This avenue is only possible in active flow devices, as fluid may be flowing backwards through the device.

As noted in 5.3, fluidic properties (e.g., viscosity, concentration, or density) are not captured in the front-end (*BioScript*) or utilized during device synthesis (Inkwell). By capturing these properties, Xylograph could further reduce the need for expertise required in fabricating and verifying more interesting and complicated devices. One interesting application of including fluidic properties is solute separation – making separating blood cells and cancer cells significantly simpler while using far less blood. Including these properties has implications for the inclusion of timing constraints as well. Timing constraints allow a user to specify that a particular fluidic variable must be used within a particular time interval. Capturing fluidic properties also has implications for component generation: without fluidic properties a "best match" component may not be appropriate to achieve homogeneity between the fluid(s) and the component's function.

Component designers include fluidic properties when designing new components. Including fluidic properties during compilation and synthesis would allow for dynamic component generation or selection and deliver the best possible device. This would entail manipulating channel width or geometries, influencing placement and routing, or even simply selecting the best component for the fluid (an improvement over Xylograph's direct mapping approach). However, even by including these properties devices still require validation prior to execution.

Finally, extending the compiler to include active-flow devices, devices that use pumps to move fluid through the channels, instead of back pressure or gravity, would be a natural extension; allowing active control of the device — the ability to change execution based on various readings or inputs. This hass the added benefit of removing the intersecting channel limitation. This would require a further extension to all 4 steps denoted in Fig. 5.2. Component selection and generation would need to include components that are able to be controlled. The compiler would need to know how the selected components behave, and how to actuate them. Fabricating a device with multiple layers is feasible; however, still requires some fashion of human intervention. The different layers would have to be aligned precisely so they can be bonded. This step, in using the cost-effective CNC mill, requires human intervention. Execution of the assay would require a feedback loop that can notify the compiler of any changes to execution.

5.5 Conclusion

Xylograph is the first workflow of its kind: system designers can design assays in a high-level language, and quickly and efficiently derive a design that can be fabricated using one of many fabrication techniques allowing for the creation of far more complicated assays and devices than current practices allow. We then demonstrated Xylograph's capabilities run a dilution assay which correctly generated concentrations at a 33% increments.

Given Xylograph is a first of its kind, there are inherent limitations. Currently, Xylograph only supports a subset of the instructions *BioScript* employs. For instance, Xylograph doesn't support the `detect` or `heat` instructions. This is, in part, a result of both

design libraries not supporting the necessary components and fabrication techniques not supporting off-chip operations; not to mention any of the physical constraints imposed on microfluidic devices, e.g., relative resistances across channels.

While *BioScript* makes it easy to design assays, not all assays can be fabricated by Inkwell. There are certain assays that are correct programs, that can be compiled, and even designed, but are not valid devices.

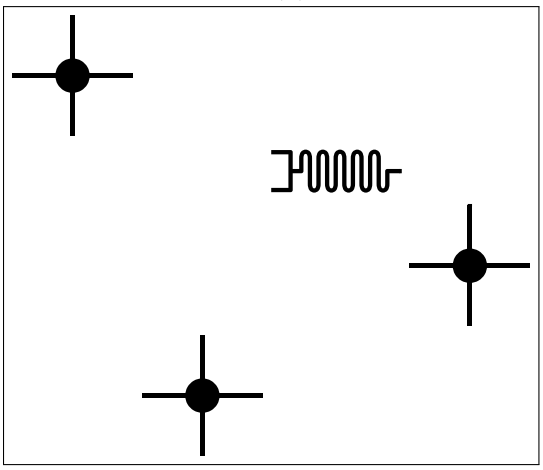
Our experimental setup does not include the time taken to fabricate or execute the device as the fabrication of the device is dependent upon too many variables: the placement and routing algorithms used to synthesize the device, the actual device fabrication technique (photo-lithography, CNC mill, etc), or what type of device is being fabricated (continuous or active flow) in addition to assembly and setup.

```
manifest hcl
manifest buffer

instructions:
acid = dispense hcl
buf = dispense buffer
dilute = mix acid with buf

dispose dilute
```

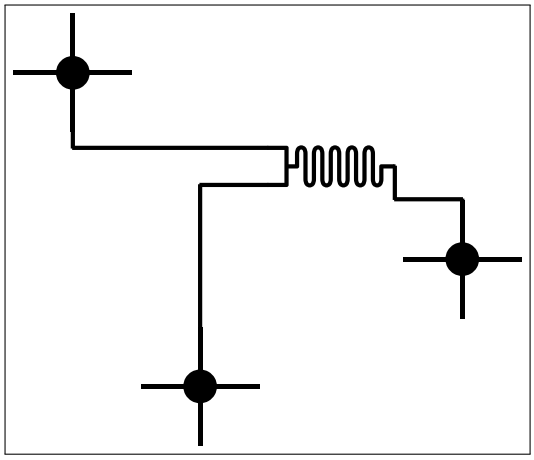
(a)

```
components: {
    input_1,
    input_2,
    mixer_1,
    output_1 },
netlist: {
    (input_1, mixer_1),
    (input_2, mixer_1),
    (mixer_1, output_1) }
```

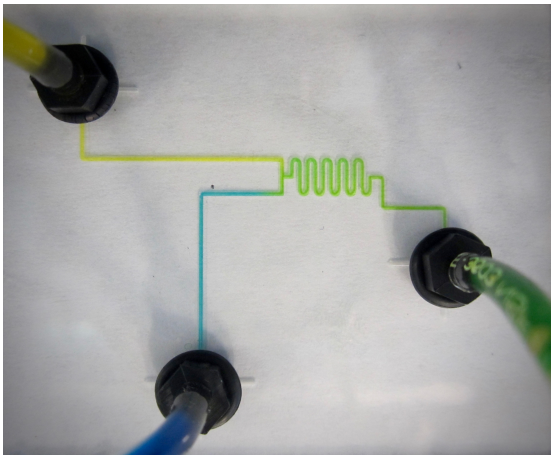
(b)



(c)



(d)



(e)

Figure 5.1: Fabricating flow-based devices begins with a *BioScript* program depicted in Fig. 5.1a. After selecting components, the *BioScript* compiler builds a netlist: Fig. 5.1b. A synthesis tool then attempts to both place the components on a device Figure 5.1c and to route the connections prescribed in the netlist Fig. 5.1d. If both placement and routing are successful, the device can be fabricated (Fig. 5.1e) using any available fabrication processes.

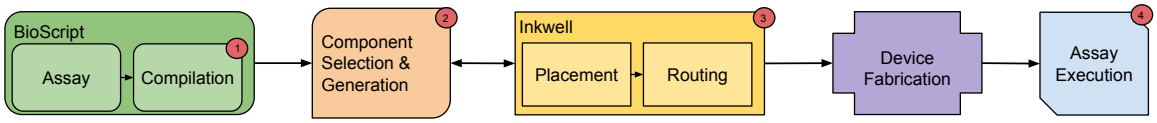


Figure 5.2: Overview of Xylograph[133, 119], which begins with an assay expressed in the *BioScript* programming language. The compiler (1) then selects components (2) while translating the assay into ParchMint, a form ingestible by Inkwell. Inkwell then attempts to solve the placement and routing problems specific to the input assay (3). If the assay yields a valid design, it can then be fabricated and used to execute the original assay (4).

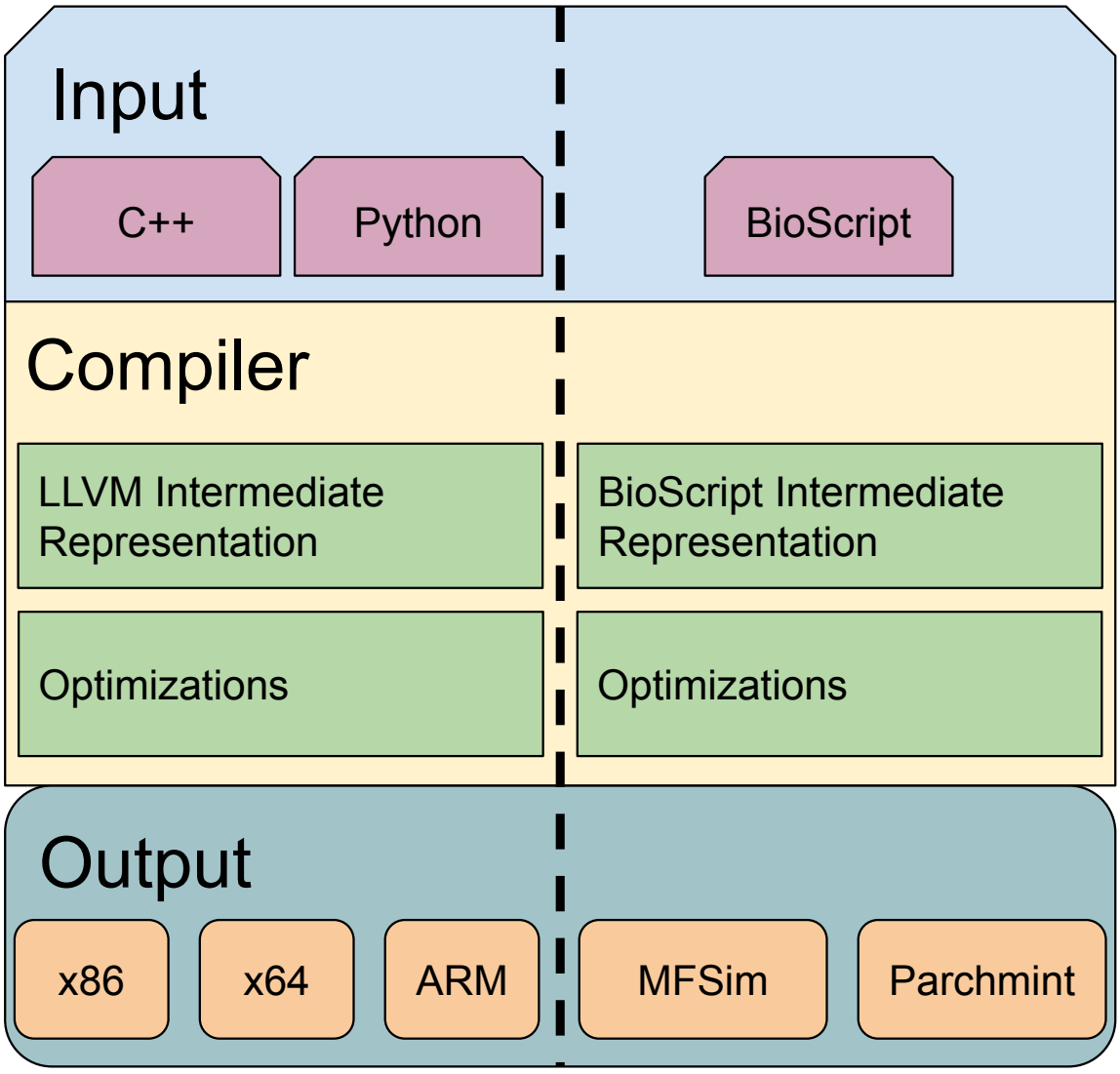


Figure 5.3: The job of a compiler: converting an input language (C++, Python) to machine code a specific architecture (x86, x64, ARM) can execute. The left side of the dotted line compares the compilation process for traditional computer architectures with that of microfluidic architectures. Relying on traditional compiler techniques allows *BioScript* to target different architectures.

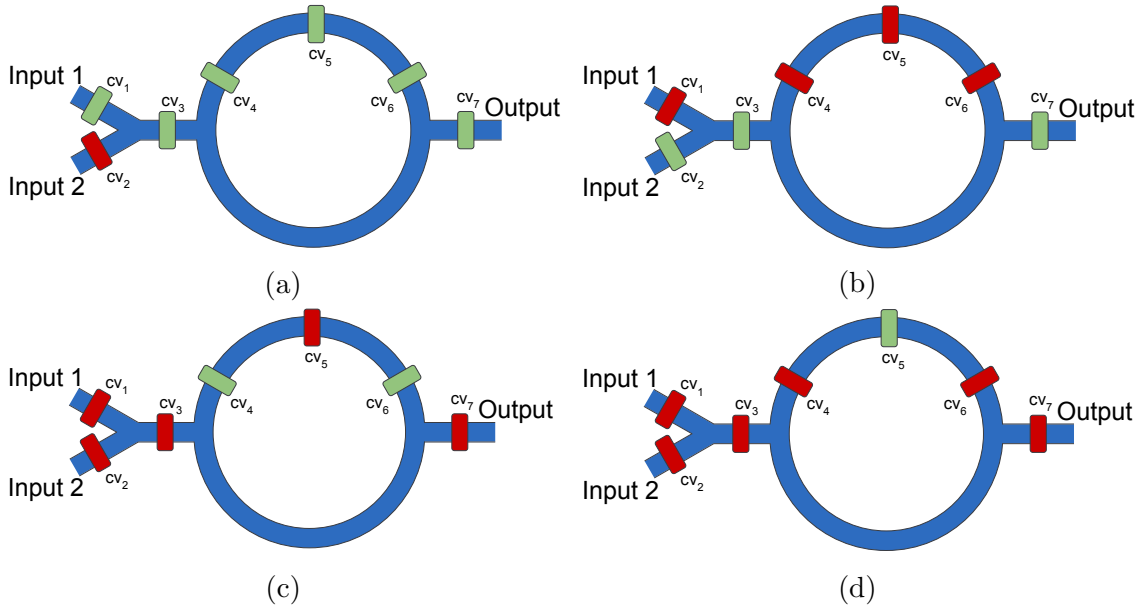


Figure 5.4: The activation sequence required for loading 2 fluids, input 1 and input 2 (Figs. 5.4a and 5.4b) and mixing them (Figs. 5.4c and 5.4d). Red denotes valves that are in a "closed" state while green denote an "open" valve.

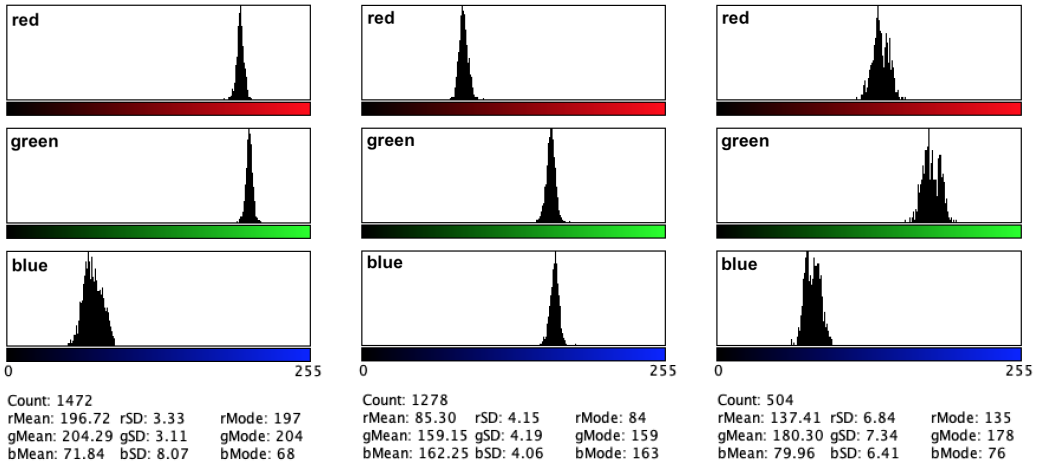
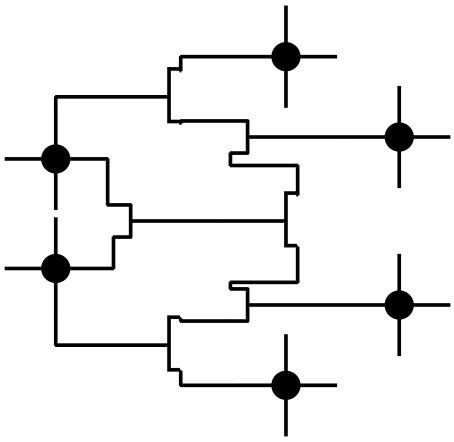


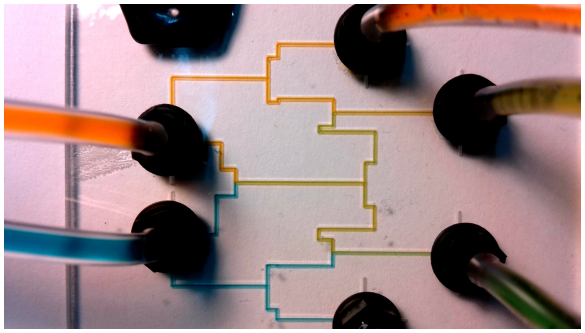
Figure 5.5: Figures 5.5a and 5.5b record the RGB histogram of the yellow and blue channels before the mix operation, respectively. Figure 5.5c shows the color histogram for the resulting mixed fluid.

```
manifest hcl
manifest buffer

instructions:

acid[2] = dispense hcl
buff[2] = dispense buffer
fifty_orig = mix acid[1]
   with buff[1]
fifty = split fifty_orig
   into 2
all_acid = split acid[0]
   into 2
all_buff = split buff[0]
   into 2
acid_66 = mix fifty[0]
   with all_acid[1]
acid_33 = mix fifty[1]
   with all_buffer[1]
dispose all_acid[0]
dispose all_buff[0]
dispose acid_66
dispose acid_33
```

(a)



(b)



(c)

Figure 5.6: Fig. 5.6a depicts the code required to build a dilution tree resulting in fluids of 0%, 33%, 66%, and 100% dilutions. Figure 5.6b is the corresponding Inkwell placement and routing masks. Finally, Fig. 5.6c is the fabricated device.

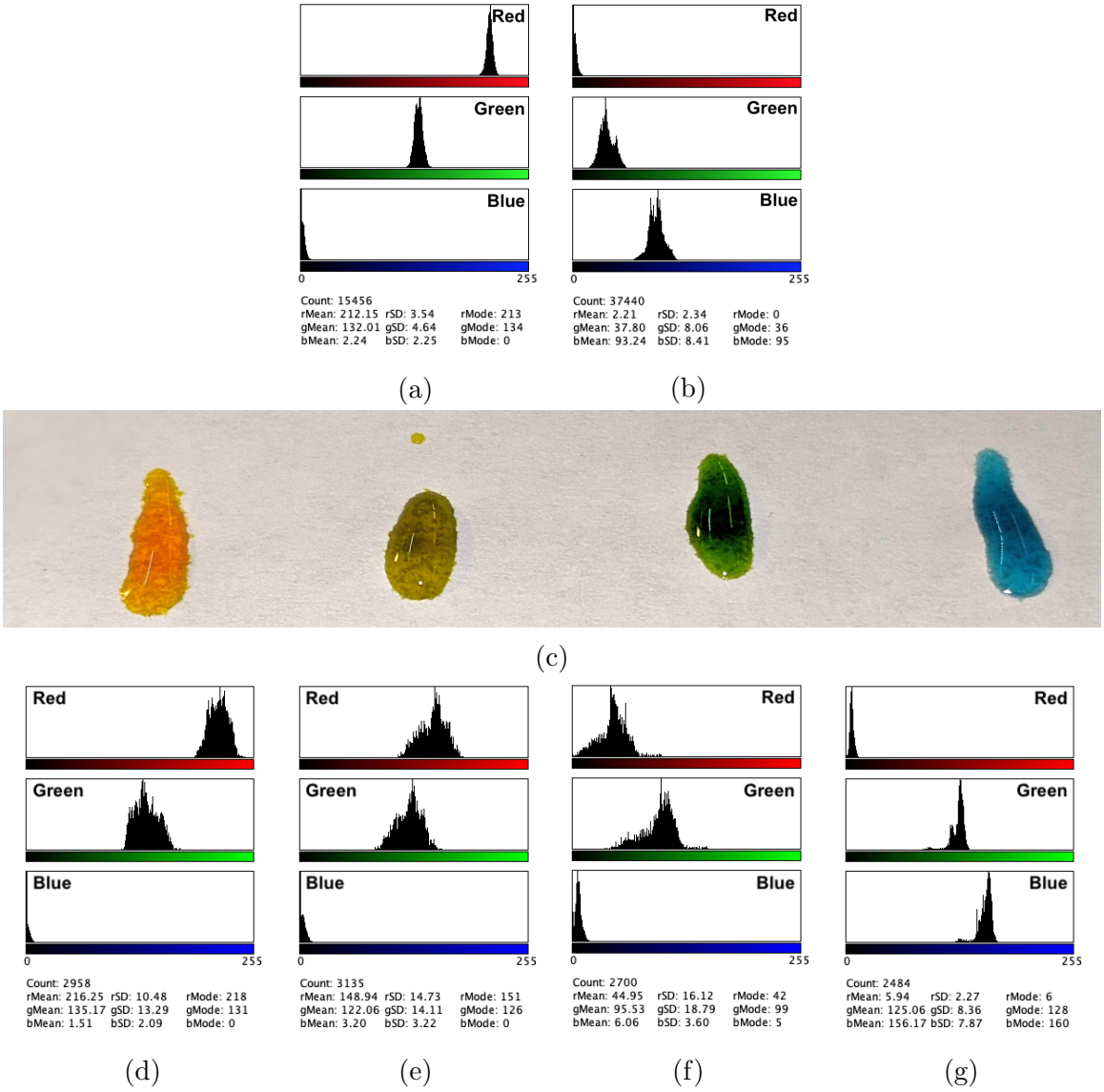


Figure 5.7: Figures 5.7a and 5.7b record the RGB histograms of the input to the dilution mixer. Figure 5.7c provides a visual of the output fluids, while Figs. 5.7d to 5.7g show the color histograms for each of the output ports on the device. Starting with Figs. 5.7a and 5.7d, we see an almost identical histogram; the same holds true for Figs. 5.7b and 5.7g. The histogram for Fig. 5.7e shows a markedly redder mixture, while Fig. 5.7f depicts a markedly greener mixture. As expected, the ranges are distributed correctly, proving correct dilutions at 0%, 33%, 66%, and 100% respectively.

Chapter 6

ChemStor

6.1 Introduction

Many common chemicals can undergo dangerous reactions when combined with incompatible chemicals during storage or disposal. For example, millions of tons of nitric acid are produced every year, making it a ubiquitous chemical in many research and industrial settings. Likewise, millions of tons of organic solvents are produced every year and used in many different applications. But when nitric acid is mixed with organic solvents, hazardous chemical products, fires, and explosions can result. While all chemists are (hopefully!) trained to avoid *intentionally* mixing nitric acid and organic solvents, accidents sometimes occur when these chemicals are *unintentionally* mixed in a chemical storage location or a waste disposal container.[1, 165, 170, 175] Furthermore, nitric acid and organic solvents are just a few of the millions of chemicals that can undergo dangerous reactions when combined during storage or disposal. Incidents due to improper storage or disposal of chemicals occur

with alarming frequency,[13, 148, 2, 169, 3] with consequences like second-degree burns[1] and destroyed laboratories.[148]

This paper introduces *ChemStor*, an open-source, automated chemical storage and disposal system that is provably safe with respect to proper laboratory safety protocols. For a given set of chemicals and containers, *ChemStor* either provides a specific storage or disposal configuration that is safe, or informs the user that no safe storage/disposal configuration is possible. Specifically, *ChemStor* informs users which specific cabinet or bin should be used to store or dispose of each chemical, thereby easing the burden of safety protocols that users must keep in their minds, while simultaneously minimizing the space required for chemical storage and disposal. *ChemStor* can be integrated with electronic laboratory notebooks, voice assistant tools, and many other existing and emerging technologies. Finally, *ChemStor* can enhance safety in a wide range of settings, not only research laboratories and industrial facilities, but also homes (where each year, *mixing incompatible pool cleaning chemicals* leads to an estimated 4,500 injuries alone[31]).

6.2 Overview of *ChemStor*

In this section, we summarize the operation of *ChemStor* in the context of a specific safety incident. In 1997, while one of the authors (WHG) was a student at the University of Tennessee, Knoxville, students in an undergraduate chemistry laboratory performed the Belousov-Zhabotinsky (BZ) reaction:

$$3\,CH_2(CO_2H)_2 + 4\,BrO_3^- \xrightarrow{Ce^{2+}} 4\,Br^- + 9\,CO_2 + 6\,H_2O \tag{6.1}$$

The BZ reaction is commonly studied in laboratory classes due to its unusual oscillatory nature. As is typically done, the students at UT Knoxville combined aqueous solutions of malonic acid and potassium bromate along with a cerium ammonium nitrate catalyst. When the reactants are combined as aqueous solutions, the reaction is benign, and the laboratory class concluded without incident. However, during post-lab cleanup, spilled reactants *in dry form* were swept from around the laboratory balances and into a waste container. This container was subsequently placed beneath a leaky sink, which began adding drops of water to the mixture of dry reactants after the laboratory was empty. The resulting reaction was extremely exothermic and caused a fire that resulted in significant damage to the laboratory[13].

How could *ChemStor* have prevented this incident? More specifically, at the end of the laboratory class, after the dry reagents used in the BZ reaction were combined into the same waste container, how could *ChemStor* warn the teaching assistants that they should avoid any situation that might add water to this container (like placing the container beneath a leaky sink)?

The first step involves defining which chemicals are currently in which storage containers. Fig. 6.1a depicts the chemical storage situation at the end of the class, with the three dry reagents used in the BZ reaction (malonic acid, potassium bromate, and cerium ammonium nitrate) combined in a single chemical storage container, and the teaching assistant contemplating placing this container beneath a sink where water might be added to it.

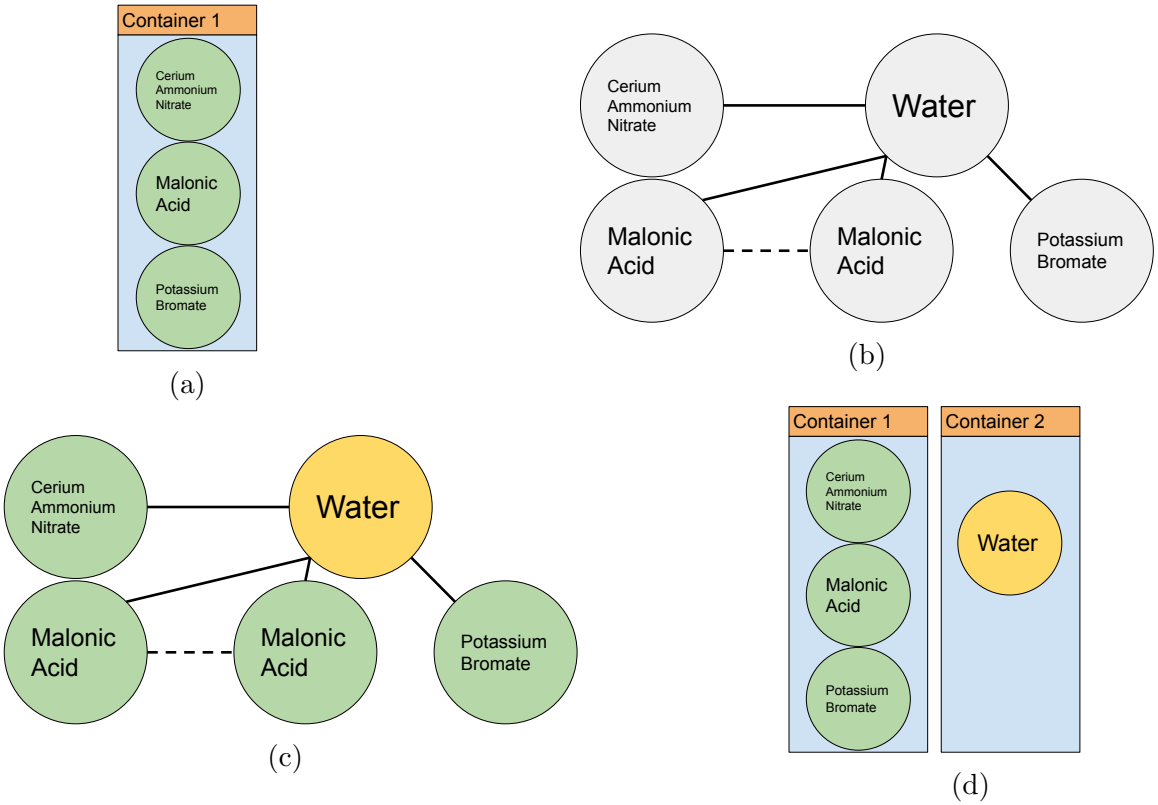


Figure 6.1: Using *ChemStor* to safely dispose of leftover reactants from performing the Belousov-Zhabotinsky (BZ) reaction. This simple scenario (based on real-life events that culminated in a lab-destroying fire [13]) begins with three reactants (cerium ammonium nitrate, malonic acid, and potassium bromate) combined in dry form in a single container **(A)**. A teaching assistant considers placing this container beneath a leaky sink drain, which will add water to the mixture. At this point, *ChemStor* constructs a chemical interaction graph **(B)** containing vertices for each chemical in the proposed mixture. In this graph, chemicals that may react with each other are linked with solid lines, and chemicals that are identical and can be combined are linked with dotted lines. After *ChemStor* calculates the chromatic number of the graph and colors the graph **(C)**, the chemicals can be safely added to different containers based on their vertex colors **(D)**. At this point, *ChemStor* would notify the teaching assistant that water should *not* be added to the container with the BZ reactants, the teaching assistant would avoid placing the container in a wet location, and a significant laboratory accident would have been avoided.

Next, *ChemStor* builds a *chemical interaction graph* G (Fig. 6.1b) containing 3 sets: a set of vertices, V; and two different sets of edges, E and A:

- The vertices, V, are the individual chemicals in this scenario. Specifically, we can define the vertices v = cerium ammonium nitrate, u = malonic acid, w = potassium bromate, and z = water. We can furthermore say that $v \in V$, $u \in V$, $w \in V$, and $z \in V$, or that v, u, w, z are members of, or are "in," the set of vertices V. (The "$\in$" symbol, as well as the other standard Boolean logic and set theory symbols used in this work, are defined in Table 6.1.)

- The set E is the set of edges that represent unsafe combinations of chemicals, or "interference" edges. Because mixing v, u, and w without z will not result in a dangerous reaction, it can be said that $(v, u) \notin E$; or that there is no edge between v and u. Similarly, there is no edge between the other combinations of the dry BZ reactants, so $(u, w) \notin E$ and $(v, w) \notin E$. Interference edges are denoted as solid lines between vertices in Fig. 6.1b. In this example the teaching assistant is considering possibly adding water to a container that already contains the dry BZ reagents (z and $z \in V$). Because z (water) *will* lead to a dangerous reaction when added to the combined dry BZ reagents, there are edges between each of the BZ reactants and water, so $(v, z) \in E$, $(u, z) \in E$, and $(w, z) \in E$, as shown in Fig. 6.1b.

- The set A is the set of edges that represent instances of the same chemical, properly defined as "affine" edges. The dotted line in Figure 6.1b represents an affine edge. In this scenario, the edge represents some additional malonic acid (a) that requires disposal. Thus, $a \in V$, and $(u, a) \in A$ because both u and a are malonic acid and

are chemically identical. The set of affine edges enables *ChemStor* to save chemical storage volume space by combining u and v, assuming the container for u or v has enough space. *ChemStor* represents this as $\forall(u, v) \in A$ ($\forall$ reads "for all").

Now that the chemical interaction graph G has been built, *ChemStor* can calcuate the smallest number of containers required for safe disposal of the chemicals in G. *ChemStor* accomplishes this by calculating the graph's chromatic number, $\chi(G)$, a step in "coloring" the graph. Graph coloring is a mathematical problem that aims to color a graph's vertices with the minimal number of different colors, while guaranteeing that no two vertices that share an edge also share the same color. For safe disposal or storage of chemicals, the only edges that are of consequence are the interference edges, or the solid edges in Fig. 6.1b. This graph is trivially small and can be colored by hand using two colors, as shown in Fig. 6.1c. However, graph coloring belongs to a class of problems that are incredibly difficult to solve efficiently, the so-called *NP-complete* problems, so as more chemicals are added to the graph, the computational effort required to color the graph grows astronomically.

Finally, the graph's chromatic number (the number of different colors on the colored graph) denotes the minimum number of containers required to safely dispose of the chemicals in the graph. In this example, *ChemStor* recommends placing water in a separate container, not in the container with the dry BZ reagents (Fig. 6.1d). If this information was then communicated to the teaching assistant, they might avoid placing the container in a location where water could be added, thereby avoiding a significant laboratory accident.

Table 6.1: Common notation in set theory and Boolean logic.

Notation	**Plain English**	**Example**	**Outcome**
$\|A\|$	Cardinality	$\|A\|$	Number of elements in the set A
$\in$	Set membership	$1 \in \mathbb{N}$	Statement of fact — 1 is a member of the set of natural numbers
$\subseteq$	Subset or equal	$A \subseteq B$	Determines whether all the elements in set A are also in set B
$A \backslash B$	Set difference	$A \backslash B$	Returns the elements in B that are not in A
$\wedge$	Boolean AND	$X \wedge Y$	Evaluates true if and only if both predicates X and Y are true
$\vee$	Boolean OR	$X \vee Y$	Evaluates true if either/and both predicates X, Y are true
$\neg$	Boolean NOT	$\neg X$	Negates the predicate X. If X is true, then $\neg X$ evaluates to false
$\forall$	For all	$\forall a(a > 1)$	Definition of the natural numbers
$\exists$	There exists	$\forall(m \in \mathbb{N})\exists(n \in \mathbb{N})(n > m)$	The set of natural numbers continues *ad infinitum*

6.3 Methods

In this section, we present details on the necessary data structures, algorithms, and constraints used by *ChemStor* to determine the safe disposal and storage of chemicals.

6.3.1 Chemical Compatibility

To obtain information about which types of chemicals are compatible or incompatible with each other, *ChemStor* relies on a chemical classification system created jointly by the US Environmental Protection Agency (EPA) and National Oceanic and Atmospheric Administration (NOAA)[56]. This system categorizes over 9,800 chemicals into 68 reactivity groups that have similar properties. Mixing materials from certain reactivity groups can produce materials from other reactivity groups; for example mixing *acids* and *bases* induces a strong reaction that produces *salt* and *water*. The EPA/NOAA categorization assigns one of three outcomes to the combination of chemicals: *Incompatible*, *Compatible*, or *Caution*. We write $\mathsf{interact}(x, y) = \lightning$ if chemicals x and y cause an adverse reaction ($\lightning$) when mixed, or are *Incompatible*. Chemical combinations that result in *Caution* are either deferred to the expert user or treated as *Incompatible*.

6.3.2 Chemical Interaction Graph

Let $G(V, E, A)$ be the *chemical interference graph* representing a storage problem. V is the set of chemicals we desire to store, E is the set of *interference edges* between incompatible chemicals, and A is the set of *affinity edges* between unique instances of identical chemicals. An interference edge (v, u) is added to E if, for any two chemicals v, u, $\mathsf{interact}(v, u) = \lightning$

as noted in 6.3.1. An affinity edge (v, w) is added to A if v and w represent distinct instances of the exact same chemical (i.e., there is more than one container of a certain chemical we wish to store). The chemical interaction graph may be extended to include a set $P \subseteq V$ of vertices and $Q \subseteq E$ of edges representing chemicals already in storage and their corresponding edges, respectively.

6.3.3 Chemical Storage

Let $C = \{c_1, c_2, \ldots, c_k\}$ be the set of *cabinets* for chemical storage. Each cabinet contains a finite set of *shelves*. Let $S(c_m)$ denote the set of shelves within cabinet $c_m \in C$, where s_n^m denotes the n^{th} shelf in c_m. Each shelf $s_n^m \in S(c_m)$ has an immutable capacity, denoted $\mathsf{maxCapacity}(s_n^m)$ with which it can use to store chemicals. The capacity of each cabinet is the sum of the capacities of its shelves. The capacity of a shelf currently occupied by chemicals is $\mathsf{currCapacity}(s_n^m)$.

Affinity-adjacent chemicals may be combined into the same container. The cost of a container is orders of magnitude less than the cost of a cabinet; as such, we assume an infinite supply of containers but a finite supply of cabinets.

Let $\mathsf{cvol}(x)$ denote the current volume chemical x occupies within its container, and let $\mathsf{combine}(y, z)$ be the volume of the combined quantities of chemicals y and z, which in either case may be less than the volume of their respective containers. Two instances of the same chemical v and w can be combined by coalescing their respective vertices in G.

6.3.4 Chemical Disposal

Let $D = \{d_1, d_2, \ldots, d_k\}$ be the set of *containers* for chemical disposal. Each container has a maximum volume associated with it, denoted $\mathsf{maxVolume}(d_m)$. The current volume of the container d_m is $\mathsf{currVolume}(d_m)$.

Thus, if $(v, u) \in E$, then v and u cannot be combined in the same container d_m (e.g., $\mathsf{interact}(v, u) = \lightning$), so a new container must be added: d_{m+1}. If $(v, u) \notin E \wedge \mathsf{vol}(v) + \mathsf{currVolume}(d_m) \leq \mathsf{maxVolume}(d_m)$, then v can be combined with u in the container d_m. Affinity-adjacent chemicals are assumed to be combined into the same container, assuming the maximum volume of the container allows it.

6.3.5 Characterization of a Solution to the Chemical Storage Problem

Given a chemical interference graph G, *ChemStor* computes a pair of functions $f : V \rightarrow \{1, 2, \ldots |C|\}$ and $g : V \rightarrow \mathbb{N}^+$ which assigns each chemical (vertex) to a specific storage location within a cabinet and on a shelf, or (cabinet, shelf). If $(f(v), g(v)) = (m, n)$, then chemical v is assigned to shelf s_n^m in cabinet c_m, $1 \leq n \leq |S(c_m)|$.

A legal chemical storage solution must satisfy the *Chemical Reactivity Constraint*, which states that two interfering chemicals cannot be stored in the same cabinet:

$$f(v) \neq f(u) \; \forall (v, u) \in E \tag{6.2}$$

A more permissive variant of the Chemical Reactivity Constraint allows two interfering chemicals to be stored in the same cabinet, but on different shelves:

$$\begin{aligned} p_1 &= f(v) \neq f(u) \\ p_2 &= f(v) = f(u) \wedge g(v) \neq g(u) \\ p_{12} &= p_1 \vee p_2 \ \forall (v,u) \in E \end{aligned} \tag{6.3}$$

A legal chemical storage solution must also satisfy the *Storage Capacity Constraint*, which states that the sum of the capacities of the containers assigned to each shelf in each cabinet cannot exceed that shelf's storage capacity:

$$\begin{aligned} \sum_{v \in V | f(v)=m,\ g(v)=n} \mathsf{vol}(v) &\leq \mathsf{maxCapacity}(s_n^m) \\ 1 \leq m &\leq |C| \\ 1 \leq n &\leq |S(c_m)| \end{aligned} \tag{6.4}$$

6.3.6 Satisfiability Modulo Theories

ChemStor uses a class of logical formula solvers, Satisfiability Modulo Theories (SMT), to solve a given storage or disposal problem instance. An SMT solver determines whether a problem instance is "decidable", or can be answered by a simple "true" or "false". SMT problems support linear inequalities (e.g., $x+5y-2z \leq 5$), equalities involving uninterpreted terms or functions (e.g., $f(u,v) = f(g(v),u)$), Boolean logic (e.g., $a \wedge b$), and in some cases quantifiers (e.g., $\forall a(a \in \mathbb{N})(a > 0)$).

SMT-based problems are expressed as a series of mathematical constraints. These constraints define the valid range of values variables can take for a solution. SMT equations

are very expressive and can take one or many of the form(s) noted above. Once these equations are defined, they are used as input to an SMT solver. If the solver can find a solution which satisfies the constraints, it provides a model, or the values of all the variables. If no solution can be found, the solver simply returns "false".

6.3.7 SMT Constraints

To use a SMT solver to solve a *ChemStor* problem instance, we first convert the Chemical Storage Problem, described in 6.3.5, into a set of SMT equations. Each chemical v must be assigned to exactly one shelf $s_{n_v}^{m_v}$ in exactly one cabinet c_{m_v}. We accomplish this using the following constraints:

$$m_v \in \mathbb{Z}, 1 \leq m_v \leq |C| \tag{6.5}$$

$$n_v \in \mathbb{Z}, 1 \leq n_v \leq |S(c_k)| \wedge k == m_v \tag{6.6}$$

If v is a previously stored chemical, then the values for m_v and n_v are known *a priori* and are encoded as SMT constants.

The second constraint, the *Chemical Reactivity Constraint*, guarantees that no pair of chemicals stored in the same cabinet can interact dangerously, and can be expressed as an SMT constraint as follows:

$$\forall u, v \in V | m_u == m_v \; \mathsf{interact}(u, v) \neq \lightning . \tag{6.7}$$

The more permissive variant of this constraint, Eq. (6.3), guarantees that no pair of chemicals stored in the same shelf in the same cabinet can interact dangerously, and can be

expressed as an SMT constraint as follows:

$$\forall u, v \in V | m_u == m_v \wedge n_u == n_v,\ \mathsf{interact}(u, v) \neq \lightning . \tag{6.8}$$

Finally, the *Storage Capacity Constraint* (Eq. (6.4)) expresses the Storage Capacity Constraint in a form that is already SMT-compatible.

6.3.8 Coalescing Strategy

As defined in 6.3.2, an affinity edge $(u, v) \in A$ represents two containers that store identical chemicals. Let $t(u)$ denote the chemical "type" of u. In *ChemStor*'s case, the reactivity groups described in 6.3.1 comprise the different "types" to which chemicals may belong, like "acid" and "base." In Eq. (6.4), $\mathsf{vol}(v)$ represents the volume of the container that holds chemical v. Let $\mathsf{cvol}(v) \leq \mathsf{vol}(v)$ denote the volume of the chemical held in the container. To reduce demands on limited storage space, it may be possible to consolidate multiple instances of the same chemical (u and v) into u's container if

$$\mathsf{cvol}(u) + \mathsf{cvol}(v) \leq \mathsf{vol}(u). \tag{6.9}$$

Here, the user no longer needs to store v's container, and *ChemStor* can eliminate all of v's associated SMT constraints.

We implement this feature as a coalescing (vertex merging) operation applied to the chemical interference graph prior to calling the SMT solver; in practice, coalescing opportunities could be incorporated directly into the SMT formulation as well.

Figure 6.2 illustrates coalescing. Here u represents 150 mL of hydrochloric acid in a 300 mL container and v represents 150 mL of hydrochloric acid in a 300 mL container.

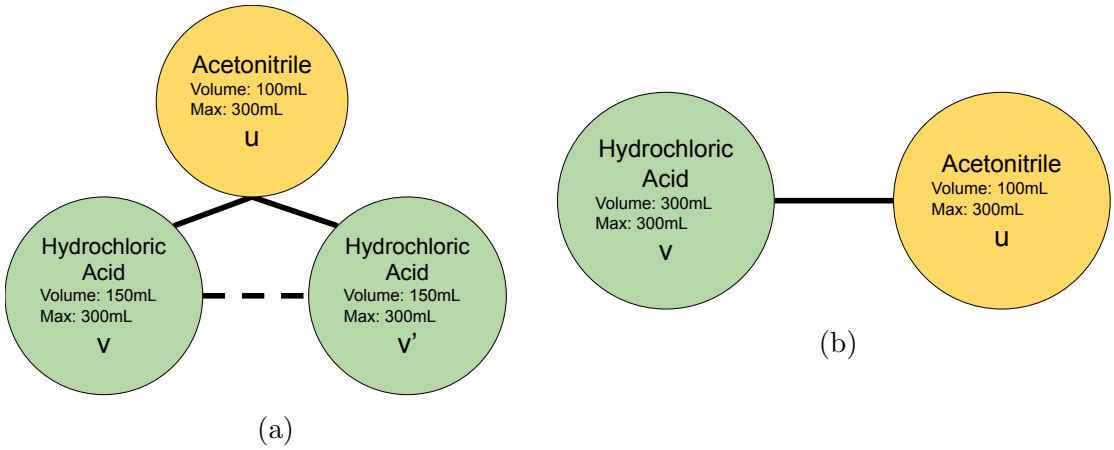


Figure 6.2: Demonstrating the coalescing strategy. The affine edge (dotted line) in **(A)** allows *ChemStor* to combine those chemicals into one vertex as shown in **(B)**, as long as the volumes $v + v' \leq \mathsf{maxVolume}(v, v')$.

Without coalescing, the shelves would use a combined 600 mL of volume to store u and v. However, with coalescing, *ChemStor* combines them into a single container, reducing the storage requirement to 300 mL.

6.3.9 De-Coalescing Strategy

In some cases, it may be necessary to *split* one chemical container into two or more containers; *ChemStor* addresses this behavior through *de-coalescing*. As a motivating example, suppose that a user tries to store 300 mL of hydrocholoric acid in a cabinet with three shelves, as shown in Fig. 6.3. Due to pre-existing chemicals allocated to storage, only 100 mL of space is available on each shelf. In this case, it makes sense to *split* the hydrochloric acid into three 100 mL containers; otherwise, a legal storage solution cannot be found. We implement this strategy using de-coalescing (vertex splitting). *ChemStor* is allowed to

de-coalesce a vertex v into a given number of p parts, each having a volume no more than $\mathsf{maxVolume}(v)/p$.

Let c_m be a cabinet that adheres to the "strict" *Chemical Reactivity Constraint*, as noted in Eq. (6.2). Let Q be the set of shelves in c_m that are not at their maximum capacity. In order to de-coalesce a chemical v, there must be sufficient volume over all the shelves in Q to store all of v, as noted in Eq. (6.10):

$$\mathsf{maxVolume}(v) \leq \sum_{q \in Q} \mathsf{maxVolume}(q) - \mathsf{currVolume}(q) \tag{6.10}$$

If this constraint is met, *ChemStor* finds a partitioning of v into p parts $\{v_1, v_2, .., v_p\}$ by dividing $\mathsf{maxVolume}(v)$ by the greatest common divisor of the available volumes amongst all shelves in Q:

$$p = \left\lceil \frac{\mathsf{maxVolume}(v)}{\mathsf{GCD}(\{\mathsf{currVolume}(q) - \mathsf{maxVolume}(q) \forall q \in Q)} \right\rceil \tag{6.11}$$

Each of these p parts can then be stored easily on the shelves in Q:

$$\forall v_i \in \{v_1, v_2, ..., v_p\}, f(v_i), g(v_i) = (c_m, q) | q \in Q \tag{6.12}$$

If there are q shelves that do not interact negatively with v whose combined available capacity is greater than $\mathsf{maxVolume}(v)$, but whose individual available capacities are smaller than $\mathsf{maxVolume}(v)$, then we can split v into an equal number of p parts, where p is $\mathsf{maxVolume}(v)$ divided by the greatest common divisor of the available capacities of the q shelves. We can then de-coalesce as described above, and the resulting p instances of chemical v can be stored on the q shelves either directly or after further coalescing.

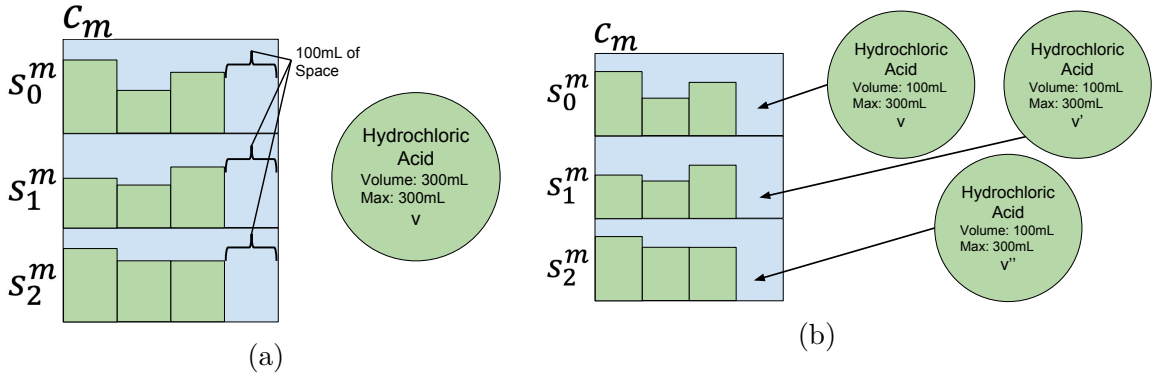


Figure 6.3: Demonstrating the de-coalescing strategy. To store a chemical whose volume exceeds the capacity of any one shelf **(A)**, *ChemStor* de-coalesces vertices and splits the chemical into p parts to derive a feasible storage configuration **(B)**.

6.3.10 No Solutions

There are instances when *ChemStor* might not converge to a legal solution. In some cases, this may be in part due to constraints imposed by the set of chemicals pre-assigned to storage locations. One possibility is to unassign all of these vertices and generate a new SMT problem instance. If this second instance is successfully solved, then a legal storage solution has been found, albeit one that may require a significant rearrangement of chemicals stored in the cabinets. If the second instance cannot be solved, then the user is informed that no legal storage solution is possible using the existing resources.

6.4 Results

We implemented *ChemStor* using the Python programming language and z3[20] as our SMT solver. All experiments were performed on a 64-bit Windows 10 Dell Laptop with an Intel(R) Core(TM) i5-7200U CPU @ 2.50 GHz with 8.00 GB of RAM. The code is available at `https://github.com/lilott8/BioScript`.

To test the efficacy of *ChemStor*, we used it to reproduce a number of real-world destructive chemical storage and disposal incidents. In some cases, details regarding the storage resources were sparse or non-existent; in these examples, we made reasonable assumptions about common storage cabinet sizes and quantities, chemical volumes, and chemical taxonomies. For each real-world incident in Table 6.2, we averaged run times across 100 runs. In each case, *ChemStor* is able to derive a safe and valid storage or disposal solution in a fraction of a second.

Table 6.3 reports the ability of *ChemStor* to handle storage problems which necessitate the use of our coalescing and de-coalescing strategies (i.e., valid solutions are unable to be found without combining or splitting containers of chemicals). *ChemStor* was able to find a *safe* storage configuration for all the synthetic storage problems where one exists, denoted by "Yes" in the *Valid Solution Found* column. In the case of the two synthetic failures, *ChemStor* was unable to find a *safe* storage configuration, as was expected. The two failing test cases demonstrate problem instances where a valid solution is impossible and the likelihood of an unsafe incident is significant. In these two cases, the coalescing and de-coalescing strategies would not prove helpful as the volume constraint on a shelf prevents a chemical from being safely stored. As in Table 6.2, we average run times over 100 runs, and note that every test returns in a few milliseconds.

Table 6.2: Results from using *ChemStor* to solve chemical storage and disposal problems from real-life incidents. In these incidents, faulty storage or disposal configurations caused lab fires, explosions, or human harm and incurred significant damages to lab spaces. All run times were averaged across 100 tests. *ChemStor* was able to find a safe chemical storage or disposal configuration for each incident in a few milliseconds.

Incident	**Avg. Run Time**(ms)	**No. of Chemicals**	**No. of Cabinets**	**No. of Shelves**	**Solution Found**
Tetracholorethylene+Nitric Acid[133]	5.523	2	1	4	Yes
Hexane Explosion[3]	2.105	7	1	7	Yes
Methanol+Nitric Acid[133]	2.386	2	1	2	Yes
Benzene+Urea+Benzotrichloride[133]	4.450	3	1	3	Yes
Lithium Aluminum Hydride Fire[2]	7.043	3	1	3	Yes
H_2O_2+Sulfuric Acid + Acetone[133]	10.086	3	1	4	Yes
Formaldehyde+Benzene[133]	11.338	3	1	4	Yes
Univ. of Tennessee, Knoxville Fire[13]	23.177	4	1	4	Yes
Broken Beaker of Barium Oxide[169]	1.272	11	3	3	Yes
Lab Fire at Ohio State University[148]	1.241	9	2	3	Yes

Table 6.3: Synthetic tests demonstrating the efficacy of *ChemStor*'s coalescing and de-coalescing strategies. All run times were averaged across 100 tests. We crafted tests for the edges case: restrictive placement, relaxed placement, coalescing success or failure, and decoalescing success or failure. If a solution could not be found, the corresponding column is marked with a "No".

Test	**Avg Run Time** (ms)	**No. of Chemicals**	**No. of Cabinets**	**No. of Shelves**	**Solution Exists**	**Solution Found**
Full Cabinet	0.64	5	1	4	Yes	Yes
Compatible Chemicals	39.54	5	3	4	Yes	Yes
Combine Two Tests	142.96	8	4	4	Yes	Yes
Pass Coalescing	4.94	1	1	1	Yes	Yes
Pass Decoalescing	10.60	1	1	3	Yes	Yes
Fail Coalescing	3.14	1	1	1	No	No
Fail Decoalesce	4.77	1	1	2	No	No

6.5 Conclusions

ChemStor automatically generates chemical disposal and storage configurations for laboratory, industrial, and domestic settings that guarantee safe storage and disposal. In all of our test cases, ranging from real-world to synthetic, *ChemStor* converged in less than one second, which indicates that realistic storage/disposal problem instances yield SMT problem instances that can be solved rapidly. This is important, as it enables *ChemStor* to provide real-time advice to users before dangerous storage and disposal mistakes are made.

In its current form, *ChemStor* has some significant limitations. For example, *ChemStor* doesn't account for chemical properties like concentrations or temperatures; obviously these properties heavily influence the reactivity of the chemicals. Also, the notion of a "cabinet" in *ChemStor* is abstract, and differentiating between, say, a refrigerator and a room-temperature shelf is an important distinction when storing a chemical with a flash point near room temperature. Finally, the *ChemStor* "container" is an abstract volume and does not capture the real-world container dimensions that dictate shelf or cabinet capacity. Future versions of *ChemStor* should address these shortcomings, and since *ChemStor* is an open-source project, we welcome others to add additional capabilities to the software and use it in their own projects.

In the near future, *ChemStor* can be incorporated into the various technological assistants that are gaining popularity in workplaces and homes. For example, by including *ChemStor* in an electronic laboratory notebook, the notebook software could automatically suggest chemical disposal strategies after each experiment. Cameras in augmented reality

systems could actively scan the workplace and use *ChemStor* to identify unsafe chemical storage situations before accidents occur, and microphones could listen for employees' questions about storage and disposal. These scenarios are not that far-fetched—software developers are already working on voice-based assistants for chemists [128], and integrating *ChemStor* into these tools seems relatively straightforward. Finally, including *ChemStor*'s recommendations in home voice assistants like Apple's Siri and Amazon's Alexa could significantly reduce the number of chemical-related injuries and accidents that occur in homes.

Chapter 7

Conclusion

LoC technology is poised to fundamentally transform biology and chemistry. However, the usability and necessary abstractions that are required were lacking. This thesis directly addressed the usability and abstraction problem by introducing *BioScript*, a fully-functional DSL targeting droplet-based pLoC technologies. *BioScript* includes a type system that can guarantee that incompatible chemicals are never mixed together and that chemicals are used only once. We then introduced the necessary analysis required for *BioScript* to support functions, including recursive functions. Further, we extended the *BioScript* compiler to support the automated design and fabrication of flow-based (p)LoC devices. Lastly, we introduce *ChemStor*, an extension to *BioScript*'s type system which guarantees the safe storage and disposal of chemicals. With these abstractions in place, (p)LoC devices are easier to program and/or design and guaranteed to operate safely.

Appendix A

Assay Execution Videos

A.1 *BioScript* on Physical Hardware

A.1.1 Image Probe Synthesis

Image probing synthesis is a technique using radioactivity or fluorescence to tag a DNA or RNA fragment to detect complimentary nucleotide substances. Image probing synthesis finds use in microbial ecology, allowing scientists to better identify species, genera, or microorganisms. Appendix E.1.12 details the corresponding *BioScript* assay, with the physical execution on a DropBot device is available on our video repository.

A.1.2 Titration

Titration is a common laboratory exercise that allows a scientist to determine the concentration of a reaction. It is a way to reverse-engineer the concentration of an unknown solvent. This experiment is carried out by taking the titrant, a chemical that has a known

concentration and volume, with a chemical whose concentration is unknown. A scientist mixes the two together until the reaction reaches "neutralization" — which usually results in a simple color change of the mixed chemicals. The assay is expressed in Appendix A.1.2, and the video is available on our video repository.

```
manifest hcl
manifest naoh
module ph

instructions:

acid[2] = dispense hcl
base[2] = dispense naoh
titration = mix acid[1] with base[1]
reagent = split titration into 2
acids = split acid[0] into 2
bases = split base[0] into 2
thirty_three = mix reagent[0] with bases[1]
sixty_six = mix reagent[1] with acids[1]

one = detect ph on acids[0]
two = detect ph on bases[0]
three = detect ph on thirty_three
four = detect ph on sixty_six

dispose acids[0]
dispose bases[0]
dispose thirty_three
dispose sixty_six
```

A.2 MFSim Simulated Videos

Below is a list of the videos generated while gathering data. We are not claiming these as novel or interesting contributions. They are merely provided to demonstrate how the simulator executes a given assay.

- Urine Opiate Panel with initial positive indication:

  `https://1drv.ms/v/s!AqwZF7jQGx_IgahNYojptiZr4RpjOg`

- Urine Opiate Panel with initial negative indication:

  https://1drv.ms/v/s!AqwZF7jQGx_IhJRtiOKGuSTm0qXX0g

- PCR Assay:

  https://1drv.ms/v/s!AqwZF7jQGx_IhNgWWxzt8GwtfICcFw

- Image Probe Synthesis:

  https://1drv.ms/v/s!AqwZF7jQGx_IhNgbKsUIzDGqfyOhbw

- Neurotransmitter Sensing:

  https://1drv.ms/v/s!AqwZF7jQGx_IhNgZjBF4vlsnDr1uCA

- PCR Droplet Replenishment Assay:

  https://1drv.ms/v/s!AqwZF7jQGx_IhJRqfq1dLhbBUIL9RQ

- Probabilistic PCR Full:

  https://1drv.ms/v/s!AqwZF7jQGx_IhJRo-xDbVT_Q6UsXPg

- Probabilistic PCR Early Exit:

  https://1drv.ms/v/s!AqwZF7jQGx_IhJRpLXveGI8Qr7BWrw

- Broad Spectrum Opiate Panel:

  https://1drv.ms/v/s!AqwZF7jQGx_IhJRs4f6laDxl6T_QJQ

- Fentanyl ELISA Assay:

  https://1drv.ms/v/s!AqwZF7jQGx_IgahJFK519Id1HCad4g

- Ciprofloxacin ELISA Assay:

  https://1drv.ms/v/s!AqwZF7jQGx_IgahKrD6EDWPbsy9G0g

- Heroin ELISA Assay:

  https://1drv.ms/v/s!AqwZF7jQGx_IgahIrCW0z08KR1111A

- Morphine ELISA Assay:

  https://1drv.ms/v/s!AqwZF7jQGx_IgahG3PHVTctW7u0rOw

- Oxycodone ELISA Assay:

  https://1drv.ms/v/s!AqwZF7jQGx_IgahHIgmGgR43A-l7EA

Appendix B

BioScript Proofs

B.1 Helper Lemmas

Lemma 5 (Name extension)

$\forall\ \Gamma, X, i, X', X''.$

$$\Gamma, X_1 \vdash i, X_2\ \wedge\ X_1 \subseteq X_1' \Rightarrow\ \exists X_2'.$$

$$\Gamma, X_1' \vdash i, X_2'\ \wedge\ X_2 \subseteq X_2'$$

and

$\forall\ \Gamma, X, s, X', X''.$

$$\Gamma, X_1 \vdash s, X_2\ \wedge\ X_1 \subseteq X_1'\ \Rightarrow$$

$\exists X_2'.$

$$\Gamma, X_1' \vdash s, X_2'\ \wedge\ X_2 \subseteq X_2'$$

Proof. Trivial by mutual induction on s and i. ■

Lemma 6

For every Γ, X, i, X', s, *and* X'', *if*

$\Gamma, X \vdash i, X', \;\; X'' \subseteq X'$ *and* $\Gamma, X'' \vdash s, X'''$

there exists X'''' *such that*

$\Gamma, X \vdash i; s, X''''$ *and* $X''' \subseteq X''''$

Proof. Direct from Lemma 5 and the rule T-Inst. ■

Lemma 7 (Typing $\bullet$)

For every Γ, X, s_1, X', s_2, *and* X''

$\Gamma, X \vdash s_1, X' \;\wedge\; \Gamma, X' \vdash s_2, X'' \;\Rightarrow\; \Gamma, X \vdash s_1 \bullet s_2, X''$

Proof. Trivial by induction on s_1. ■

Lemma 8

For every Γ, X, s_1, X', s_2, *and* X'', *if*

$\Gamma, X \vdash s_1, X', \;\; X'' \subseteq X'$ *and* $\Gamma, X'' \vdash s_2, X'''$

there exists X'''' *such that*

$\Gamma, X \vdash s_1 \bullet s_2, X''''$ *and* $X''' \subseteq X''''$

Proof. Direct from Lemma 5 and Lemma 7. ■

Lemma 9 (Canonical Forms)

For every Γ, X *and* v,

- *If* $\Gamma, X \vdash v : Mat_i$, *then* $v \in Mat_i$.
- *If* $\Gamma, X \vdash v : \mathbb{R}$, *then* $v \in \mathbb{R}$.

- *If* $\Gamma, X \vdash v : \mathbb{N}$*, then* $v \in \mathbb{N}$*.*

Proof. Immediate from case analysis on the structure of v and using the inversion lemma, Lemma 10. ■

Lemma 10 (Inversion on typing of terms)

1. *If* $\Gamma, X \vdash x : T$*,*

   then $x : T \in \Gamma$ *and* $x \in X$*.*

2. *If* $\Gamma, X \vdash t_1 \oplus t_2 : T$*,*

   then $\Gamma, X \vdash t_1 : T$ *and* $\Gamma, X \vdash t_1 : T$ *and* $T = \mathbb{N} \vee T = \mathbb{R}$

3. *If* $\Gamma, X \vdash$ *detect* $module$ *on* x *for* $t : T$*,*

   then there exists $\overline{Mat_i}$ *such that* $\Gamma, X \vdash x : \cup\overline{Mat_i}$ *and* $\Gamma, X \vdash t : \mathbb{R}$ *and* $T = \mathbb{R}$

4. *If* $_{\Gamma, X \vdash mat : T}$*, then there exists* i *such that* $T = Mat_i$

5. *If* $_{\Gamma, X \vdash r : T}$*, then* $T = \mathbb{R}$

6. *If* $_{\Gamma, X \vdash n : T}$*, then* $T = \mathbb{N}$

Proof. Immediate from case analysis on the type derivation rules. ■

Lemma 11 (Inversion on typing of statements and instructions)

1. *For all* Γ*,* X*,* i*,* s*,* X''*,*

   If

   $$\Gamma, X \vdash i;\ s, X'',$$

   then, there exists X' *such that*

$\Gamma, X \vdash i, X'$ *and*

$\Gamma, X' \vdash s, X''$.

2. *For all* Γ, X, x, t, X',

   If

   $\Gamma, X \vdash x := t, X'$

   then, there exists T, T' *such that*

   $x \colon T \in \Gamma$,

   $\Gamma, X \vdash t : T'$,

   $T' \subseteq T$,

   $T' = \mathbb{R} \vee T' = \mathbb{N} \vee t = mat$, *and*

   $X' = X \cup \{x\}$

   or

   $t = x'$ *and*

   $X' = X \setminus \{x'\} \cup \{x\}$.

3. *For all* Γ, X, x, x_1, x_2, t, X',

   If

   $\Gamma, X \vdash x :=$ *mix* x_1 *with* x_2 *for* t, X'

   then, there exist is *and* js *such that*

   $\Gamma, X \vdash x_1 : \cup\overline{Mat_i}$,

   $\Gamma, X \vdash x_2 : \cup\overline{Mat_j}$,

   $\Gamma, X \vdash t : \mathbb{R}$,

$\overline{\textsf{interact-abs}(Mat_i, Mat_j) \subseteq \Gamma(x)}$, *and*

$X' = X \setminus \{x_1, x_2\} \cup \{x\}$.

4. *For all* Γ, X, x, x_1, .., x_n, X',

   If

   $\Gamma, X \vdash \langle x_1, .., x_n \rangle := \textsf{split}\ x\ \textsf{into}\ n, X'$

   then, there exist is such that

   $\Gamma, X \vdash x : \cup\overline{Mat_i}$,

   $\Gamma(x) \subseteq \Gamma(x_1), .., \Gamma(x) \subseteq \Gamma(x_n)$, *and*

   $X' = X \setminus \{x\} \cup \{x_1, .., x_n\}$.

5. *For all* Γ, X, t, s_1, s_2, X''',

   If

   $\Gamma, X \vdash \textsf{if}\ t\ \textsf{then}\ s_1\ \textsf{else}\ s_2, X'''$

   then, there exists X' *and* X'' *such that*

   $\Gamma, X \vdash t : \mathbb{N}$,

   $\Gamma, X \vdash s_1, X'$,

   $\Gamma, X \vdash s_2, X''$, *and*

   $X''' = X' \cap X''$.

6. *For all* Γ, X, t, s, X',

   If

   $\Gamma, X \vdash \textsf{while}\ t\ s, X'$

   then,

   $\Gamma, X \vdash t : \mathbb{N}$

$\Gamma, X \vdash s, X''$,

$X \subseteq X''$ *and*

$X' = X$.

Proof. Immediate from case analysis on the type derivation rules. ■

Lemma 12

For every σ, t *and* t' *if* $(\sigma, t) \to t'$ *then* $t' \notin \mathcal{X}$

Proof. Immediate from the term transition rules. ■

Helper Definitions

The conservative property of the abstract interact-abs function:

$\forall mat_i, mat_j, r.$

$mat_i \in Mat_i \wedge mat_j \in Mat_j \Rightarrow$

$\mathsf{interact}(mat_i, mat_j, r) = \lightning \Rightarrow \mathsf{interact\text{-}abs}(Mat_i, Mat_j)$ undefined

$\mathsf{interact}(mat_i, mat_j, r) \neq \lightning \Rightarrow \mathsf{interact}(mat_i, mat_j, r) \in \mathsf{interact\text{-}abs}(Mat_i, Mat_j)$

The type of the functions used for evaluation:

$\mathsf{detect} : \ Mat_i \rightarrow Module \rightarrow \mathbb{R} \rightarrow \mathbb{R}$

$\mathsf{interact} : Mat_i \rightarrow Mat_j \rightarrow \mathsf{interact\text{-}abs}(Mat_i, Mat_j)$

$\mathsf{split} : \ Mat_i \rightarrow \mathbb{N} \rightarrow Mat_i$

We define the consistency condition between the static typing environment Γ and the run-time store σ as:

$consistent(\Gamma, \ X, \ \sigma) \ = \ \forall x, T.$

$(x : T) \in \Gamma \wedge x \in X \Rightarrow$

$\sigma(x) \in T$

B.2 Proof of Progress

Lemma 13 (Progress of Terms)

For every Γ, X, t *or* T,

if

$\Gamma, X \vdash t : T$

then

$\forall \sigma.\ consistent(\Gamma,\ X,\ \sigma) \Rightarrow \exists t'.\ (\sigma, t) \rightarrow t'$ *or*

t *is a value.*

Proof.

Proof by induction on t and case analysis thereafter.

We assume

(1) $\Gamma, X \vdash t : T$

Case for the rule T-VAR:

We have

(2) $t = x$

(3) $x : T \in \Gamma$

(4) $x \in X$

We assume

(5) $consistent(\Gamma, X, \sigma)$

From [3], [4] and [5]:

(5) $\sigma(x) \in T$

By the rule E-VAR on [5]:

(6) $(\sigma, x) \to \sigma(x)$

By the rule E-VAR on [6] and [2]:

(6) $(\sigma, t) \to \sigma(x)$

Case for the rule T-MATH:

(2) $t = t_1 \oplus t_2$

(3) $\Gamma \vdash t_1 : T$

(4) $\Gamma \vdash t_2 : T$

(5) $T = \mathbb{R} \vee T = \mathbb{N}$

By I.H. on t_1:

Case 1: t_1 is not a value:

(6) $\forall \sigma, X.\ consistent(\sigma, X, \Gamma) \Rightarrow \exists t_1'.\ (\sigma, t_1) \to t_1'$

We assume that

(7) $consistent(\sigma, X, \Gamma)$

and prove that

$\exists t', (\sigma,\ t) \to t'$

From [6] and [7], there exists t_1' such that

(8) $(\sigma, t_1) \to t_1'$

From the rule E-MATHR1 on [8]:

(9) $(\sigma,\ t_1 \oplus t_2) \to t_1' \oplus t_2$

From [9] on [2]

$(\sigma,\ t) \to t_1' \oplus t_2$

Case 2:

(10) t_1 is a value v_1:

By Lemma 9 on [3], [5], [10]

(11) $v_1 \in \mathbb{N} \vee v_1 \in \mathbb{R}$

By I.H. on t_2:

Case 2.1: t_2 is not a value:

(12) $\forall \sigma, X.\ \mathit{consistent}(\sigma, X, \Gamma) \Rightarrow \exists t_2', (\sigma,\ t_2) \to t_2'$

We assume that

(13) $\mathit{consistent}(\sigma, X, \Gamma)$

and prove that

$\exists t', (\sigma,\ t) \to t'$

From [12] and [13], there exists t_2' such that

(14) $(\sigma,\ t_2) \to t_2'$

From the rule E-MathR2 on [14]:

(15) $(\sigma,\ v_1 \oplus t_2) \to v_1 \oplus t_2'$

From [15], [2], [10]:

$(\sigma,\ t) \to v_1 \oplus t_2'$

Case 2.2:

(16) t_2 is a value, v_2:

By Lemma 9 on [4], [5], [16]

(17) $v_2 \in \mathbb{N} \vee v_2 \in \mathbb{R}$

From the rule E-Math on [11] and [17]:

(18) $(\sigma, v_1 \oplus v_2) \rightarrow v_1 \oplus v_2$

From on [17], [10], [16]

$(\sigma, t) \rightarrow v_1 \oplus v_2$

Case for the rule T-Detect:

$t = \mathsf{detect}\ module_i\ \mathsf{on}\ x\ \mathsf{for}\ t'$

Similar to the the rule T-Math rule, by induction hypothesis in t' and then using the rule E-Detect and the rule E-DetectR. The consistency condition is used to derive the first premise of the rule E-Detect.

Case for the rule T-Mat:

mat is a value.

Case for the rule T-Real:

r is a value.

Case for the rule T-Nat:

n is a value. ■ **Proof.**

Case analysis on the typing derivation:

Case for the rule T-Skip:

$s = \mathsf{skip}$

Case for the rule T-Inst:

$s = i;\ s'$

$\Gamma, X \vdash i, X''$

Immediate from Lemma 14. ■

Lemma 14 (Progress for Instructions)

For every Γ, X, i, s, X',

if

$\Gamma, X \vdash i, X'$

then

$\forall \sigma.\ consistent(\Gamma, X, \sigma) \Rightarrow$

$\exists \sigma', s'.\ (\sigma,\ i; s) \rightarrow (\sigma',\ s')$

Proof.

We have that

(1) $\Gamma, X \vdash i, X'$

Case analysis on the typing derivation:

Case for the rule T-Assign-1:

(2) $i = (x := v)$

(3) $x : T \in \Gamma$

(4) $\Gamma, X \vdash t : T'$

(5) $T' \subseteq T$

From the rule E-Assign

(6) $(\sigma, x := v;\ s) \rightarrow (\sigma[x \mapsto v];\ s)$

From [6], [2]

$(\sigma,\ i; s) \to (\sigma[x \mapsto v];\ s)$

Case for the rule T-Assign-2:

Similar to the previous case. The reduction uses the rule E-Assign$'$

Case for the rule T-Assign-3:

(2) $i = (x := t)$

(3) $x : T \in \Gamma$

(4) $\Gamma, X \vdash t : T'$

(5) $(T' = \mathbb{R} \vee T' = \mathbb{N}) \wedge t \notin \mathcal{V} \cup \mathcal{X}$

(6) $T' \subseteq T$

By Lemma 13 on [4]:

Case 1:

(7) $\forall \sigma.\ \mathit{consistent}(\Gamma,\ X,\ \sigma) \Rightarrow \exists t'.\ (\sigma, t) \to t'$

We assume that

(8) $\mathit{consistent}(\Gamma,\ X,\ \sigma)$

We prove that

$(\sigma,\ i; s) \to (\sigma',\ s')$

From [7] and [8], there exists t' such that

(9) $(\sigma, t) \to t'$

From the rule E-AssignR on [9] and [5]:

(10) $(\sigma, (x := t);\ s) \rightarrow (\sigma',\ (x := t');\ s)$

From [2] on [10]:

$(\sigma, i;\ s) \rightarrow (\sigma',\ (x := t');\ s)$

Case 2:

(11) t is a value, v.

Contradiction with [5].

Case for the rule T-Mix:

(2) $i = (x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t)$

The proof is similar to the case for the rule T-Assign. Lemma 13 is applied to the type derivation for t. There are two cases. Case 1: If t steps, the rule E-MixR is applicable. Case 2: If t is a value, the rule E-Mix is applicable. Lemma 10 and the consistency condition is used to show that x_1 and x_2 are both material values $\sigma(x_1)$ and $\sigma(x_2)$ in the store. In addition, from the case analysis on the typing derivation, we have that $\mathsf{interact\text{-}abs}(Mat_i, Mat_j)$ is defined; thus, by the conservative property of the abstract interaction $\mathsf{interact}(\sigma(x_1), \sigma(x_2)) \neq \lightning$.

Case for the rule T-Split:

(2) $i = (\langle x_1, .. x_n \rangle = \mathsf{split}\ x_1\ \mathsf{into}\ n)$

The consistency condition is used to show that x is a material value in the store. Then, the rule E-Split is applicable.

Case for the rule T-If:

(2) $i = \mathsf{if}\ t\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$

We apply Lemma 13 to the type derivation for t. There are two cases. Case 1: If t steps, the rule E-IfR is applicable. Case 2: If t is a value, by Lemma 9 on the typing judgement for t, we know that it is a natural number. If it is non-zero, the rule E-IfTrue is applicable; otherwise the rule E-IfFalse is applicable.

Case for the rule T-While:

(2) $i = \mathsf{while}\ t\ s$

The rule E-While is applied without any premise.

■

Lemma 15 (Progress for Statements)

For every Γ, X, s, X',

if

$\Gamma, X \vdash s, X'$

then either s *is* skip *or*

$\forall \sigma.\ \mathit{consistent}(\Gamma, X, \sigma) \Rightarrow$

$\exists \sigma', s'.\ (\sigma, s) \rightarrow (\sigma', s')$

Proof.

We have that

(1) $\Gamma, X \vdash s, X'$

Case analysis on s:

Case $s = \mathsf{skip}$

Conclusion is immediate.

Case

(2) $s = i; s'$

From [1] and [2],

(3) $\Gamma, X \vdash i; s', X'$

By Lemma 11 on [3], there exists X'' such that

(4) $\Gamma, X \vdash i, X''$

(5) $\Gamma, X'' \vdash s', X'$

By Lemma 14 on [4],

(6) $\forall \sigma.\ \mathit{consistent}(\Gamma, X, \sigma) \Rightarrow$

$\exists \sigma', s''.\ (\sigma, i; s') \rightarrow (\sigma', s'')$

The conclusion is immediate form [2] and [6]. ■

B.3 Proof of Preservation

Lemma 16 (Preservation of Terms)

For every Γ, X, t, T *and* σ,

if

$\Gamma, X \vdash t : T$ *and*

$(\sigma,\ t) \to t'$ *and*

$\mathit{consistent}(\Gamma, X, \sigma)$

then

$\Gamma, X \vdash t' : T$

Proof.

We have

(1) $\Gamma, X \vdash t : T$

(2) $(\sigma,\ t) \to t'$

(3) $\mathit{consistent}(\Gamma, X, \sigma)$

Straightforward induction on the derivation of $\Gamma, X \vdash t : T$ and then case analysis on the final rule in the derivation of $(\sigma, t) \to t'$

Case for the rule T-VAR:

From the rule T-VAR:

(4) $t = x$

(5) $x : T \in \Gamma$

(6) $x \in X$

From the rule E-VAR:

(7) $t' = \sigma(x)$

From [3], [5], and [6]

(7) $\sigma(x) \in T$

By case analysis on the value $\sigma(x)$ and the rule T-MAT,

the rule T-REAL and the rule T-NAT

$\Gamma, X \vdash \sigma(x) : T$

Case for the rule T-MATH:

(4) $t = t_1 \oplus t_2$

(5) $\Gamma, X \vdash t_1 : T$

(6) $\Gamma, X \vdash t_2 : T$

(7) $T = \mathbb{R} \vee T = \mathbb{N}$

Case analysis on [2]:

Case for the rule E-MATHR1:

(8) $t' = t'_1 \oplus t_2$

(9) $(\sigma, t_1) \to t'_1$

By I.H. on [5], [9], and [3]:

(10) $\Gamma, X \vdash t'_1 : T$

From the rule T-MATH on [8], [10], [6], [7]:

(11) $\Gamma, X \vdash t' : T$

Case for the rule E-MathR2:

(12) $t' = v_1 \oplus t'_2$

(13) $(\sigma, t_2) \to t'_2$

By I.H. on [6], [13], and [3]:

(14) $\Gamma, X \vdash t'_2 : T$

From the rule T-Math on [12], [5], [14]:

(15) $\Gamma, X \vdash t' : T$

Case for the rule E-Math:

(16) $t_1 = v_1$

(17) $t_2 = v_2$

(18) $(v_1 \in \mathbb{N} \land v_2 \in \mathbb{N}) \lor (v_1 \in \mathbb{R} \land v_2 \in \mathbb{R})$

(19) $v_1 \oplus v_2 = v$

(20) $t' = v$

From [18], we consider the case:

(21) $v_1 \in \mathbb{N} \land v_2 \in \mathbb{N}$

The other case is similar.

From [21], [19]:

(22) $v \in \mathbb{N}$

From [20], [22] and the rule T-Nat:

(23) $\Gamma, X \vdash t' : \mathbb{N}$

From Lemma 10 on [5], [16] and [21]

(24) $T = \mathbb{N}$

From [23] and [24]:

(23) $\Gamma, X \vdash t' : T$

Case for the rule T-DETECT:

(4) $t = \mathsf{detect}\ module_i\ \mathsf{on}\ x\ \mathsf{for}\ t'$

We consider the two cases for [2]: Case for the rule E-DETECTR: Induction hypothesis is applied to t' and then the rule T-DETECT is applied. Case for the rule E-DETECT: Immediate from the rule T-REAL.

Case for the rule T-MAT:

(4) $t = mat$

mat is a value and does not step.

Case for the rule T-REAL:

(4) $t = r$

r is a value and does not step.

Case for the rule T-NAT:

(4) $t = n$

n is a value and does not step.

■

Lemma 17 (Preservation of Statements)

For every Γ, X, σ, s, X'', σ', s',

if

$\Gamma, X \vdash s, X''$ *and*

$(\sigma,\ s) \to (\sigma',\ s')$ *and*

$consistent(\Gamma, X, \sigma)$

then there exists X' *such that*

$\Gamma, X' \vdash s', X''$ *and*

$consistent(\Gamma, X', \sigma')$

Proof.

We have

(1) $\Gamma, X \vdash s, X''$

(2) $(\sigma,\ s) \to (\sigma',\ s')$

(3) $consistent(\Gamma, X, \sigma)$

Case Analysis on [1]:

Case for the rule T-Skip:

Contradiction in [2]: The statement skip does not step.

Case for the rule T-Inst:

(4) $\Gamma, X \vdash i, X'$

(5) $\Gamma, X' \vdash s'', X''$

(6) $s = i;\ s''$

Case Analysis on [4]:

Case for the rule T-Assign-1:

(7) $i = (x := v)$

(8) $x : T \in \Gamma$

(9) $\Gamma, X \vdash t : T'$

(10) $X' = X \cup \{x\}$

(11) $T' \subseteq T$

From [6], [7]:

(12) $s = (x := v);\ s''$

Case analysis on [2]:

Case for the rule E-Assign:

(13) $t = v$

(14) $s' = s''$

Case analysis on v:

Case (15) $v = r$

By Lemma 10 on [9], [13] and [15]

(16) $T' = \mathbb{R}$

From [16], [11]

(17) $\{\mathbb{R}\} \subseteq T$

From [15], [17]

(18) $v \in T$

From [3], [10], [8], [18]

(19) $consistent(\Gamma, X', \sigma[x \mapsto v])$

From [5], [14]

(20) $\Gamma, X' \vdash s', X''$

The conclusion is [20] and [19].

Case $T' = \mathbb{N}$

Similar to the previous case.

Case $t = mat$

Similar to the previous case.

Case for the rule E-AssignR:

$(\sigma, v) \to t'$

There is no reduction rule for values. Contradiction.

Case for the rule T-Assign-2:

The reduction is with the the rule E-Assign$'$. Similar to the case for the rule T-Assign-2, the consistency of σ with Γ and X and that $T' \subseteq T$ implies the consistency of $(\sigma \setminus \{x'\})[x \mapsto \sigma(x')]$ with Γ and $X \setminus \{x'\} \cup \{x\}$.

Case for the rule T-Assign-3:

(7) $i = (x := t)$

(8) $x : T \in \Gamma$

(9) $\Gamma, X \vdash t : T'$

(10) $(T' = \mathbb{R} \vee T' = \mathbb{N}) \wedge t \notin \mathcal{V} \cup \mathcal{X}$

(11) $T' \subseteq T$

(12) $X' = X \cup \{x\}$

From [6], [7]:

(13) $s = (x := t);\ s''$

Case analysis on [2]:

Case for the rule E-AssignR:

(14) $s' = (x := t');\ s''$

(15) $(\sigma, t) \to t'$

By Lemma 16 on [9], [15], and [3]:

(16) $\Gamma, X \vdash t' : T$

By Lemma 12 on [15]

(17) $t' \notin \mathcal{X}$

From [17] and either the rule T-Assign-1 or the rule T-Assign-3 on [8], [16], [10], [11]:

(18) $\Gamma, X \vdash x := t', X \cup \{x\}$

From [18] and [12]:

(19) $\Gamma, X \vdash x := t', X'$

From the rule T-Inst on [19], [5], and then [14]:

(20) $\Gamma, X \vdash s', X''$

The conclusion is [20], [3].

Case for the rule E-Assign:

$t = v$

Contradiction with [10].

Case for the rule E-Assign′:

$t = x$

Contradiction with [10].

Case for the rule T-Mix:

(7) $i = (x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t)$

(8) $\Gamma, X \vdash x_1 : \cup\overline{Mat_i}$

(9) $\Gamma, X \vdash x_2 : \cup\overline{Mat_j}$

(10) $\Gamma, X \vdash t : \mathbb{R}$

(11) $\overline{\mathsf{interact\text{-}abs}(Mat_i, Mat_j) \subseteq \Gamma(x)}$

(12) $X \setminus \{x_1, x_2\} \cup \{x\}$

Case analysis on [2]: There are two cases.

Case of the rule E-MixR:

By Lemma 16, the type of t is preserved for t'. Thus, the rule T-Mix is applied to the new mix instruction. The assumed consistency condition is preserved since the store σ stays unchanged in the step.

Case of the rule E-Mix:

We already have the typing judgement for the remaining statement s'' in [5]. By Lemma 10 on [8] and the consistency condition, there exists i such that $\sigma(x_1) \in Mat_i$ where $\Gamma(x_1) = \cup\overline{Mat_i}$. Similarly, there exists j, $\sigma(x_2) \in Mat_j$ where $\Gamma(x_2) = \cup\overline{Mat_j}$. Since $\mathsf{interact\text{-}abs}(Mat_i, Mat_j)$ is defined, from the conservative property of the abstract interact-abs function, we have:

$$\mathsf{interact}(\sigma(x_1), \sigma(x_2), r) \in \mathsf{interact\text{-}abs}(Mat_i, Mat_j)$$

Thus, from [11], we have

$$\mathsf{interact}\ (\sigma(x_1), \sigma(x_2), r) \in \Gamma(x)$$

From this and the consistency assumption for σ, we have

$$\mathit{conistent}(\Gamma, X \cup \{x\}, \sigma[x \mapsto \mathsf{interact}(\sigma(x_1), \sigma(x_2), r)])$$

Therefore, we have

$$\mathit{conistent}(\Gamma, X \setminus \{x_1, x_2\} \cup \{x\}, \sigma \setminus \{x_1, x_2\}\ [x \mapsto \mathsf{interact}(\sigma(x_1), \sigma(x_2), r)])$$

Case for the rule T-Split:

(7) $i = (\langle x_1, .. x_n \rangle = \mathsf{split}\ x\ \mathsf{into}\ n)$

(8) $\Gamma, X \vdash x : \cup\overline{Mat_i}$

(9) $\Gamma(x) \subseteq \Gamma(x_1), .., \Gamma(x) \subseteq \Gamma(x_n)$

(10) $X' = X \setminus \{x\} \cup \{x_1, .., x_n\}$

Case analysis on [2]: There is only one case. Case of the rule E-Split: We already have the typing judgement for the remaining statement s'' in [5]. By Lemma 10 on [8], we have $\Gamma(x) = \cup\overline{Mat_i}$ and $x \in X$. From the consistency condition, we have $\sigma(x) \in \Gamma(x)$. From the type of the split function, we have $\mathsf{split}(\sigma(x), n) \in \Gamma(x)$. From this and [9], we have that for every $j \in \{1..n\}$, $\mathsf{split}(\sigma(x), n) \in \Gamma(x_j)$. From this and the consistency assumption for σ, we have $\mathit{conistent}(\Gamma, X \cup \{x_1, .., x_n\}, \sigma\overline{[x_i \mapsto \mathsf{split}(\sigma(x),\ n)]})$. Therefore, we have $\mathit{conistent}(\Gamma, X \setminus \{x\} \cup \{x_1, .., x_n\}, (\sigma \setminus \{x\})\overline{[x_i \mapsto \mathsf{split}(\sigma(x),\ n)]})$.

Case for the rule T-If

(7) $i = \mathsf{if}\ t\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$

(8) $\Gamma, X \vdash t : \mathbb{N}$

(9) $\Gamma, X \vdash s_1, X_1$

(10) $\Gamma, X \vdash s_2, X_2$

(11) $X' = X_1 \cap X_2$

Case analysis on [2]: There are three cases.

Case of the rule E-IfR:

By Lemma 16, the type of t is preserved for t'. Thus, the rule T-If is applied to the new if instruction. The assumed consistency condition is preserved since the store σ stays unchanged in the step.

Case of the rule E-IfTrue:

From [11], we have $X_1 \subseteq X'$. From this and [9], by Lemma 5, we have $\Gamma, X \vdash s_1, X'$. From this and [5], by Lemma 7, we have $\Gamma, X \vdash s_1 \bullet s'', X''$. The assumed consistency condition is preserved since the store σ stays unchanged in the step.

Case of the rule E-IfFalse:

Similar to the previous case.

Case for the rule T-While:

(7) $i = \mathsf{while}\ t\ s'''$

(8) $\Gamma, X \vdash \mathsf{while}\ t\ s''', X$

(9) $\Gamma, X \vdash t : \mathbb{N}$

(10) $\Gamma, X \vdash s''', X'''$

(11) $X \subseteq X'''$

(12) $X' = X$

Case analysis on [2]: There is only one case.

Case for the rule E-WHILE:

(13) $s' = \mathsf{if}\ t\ \mathsf{then}\ (s'''\bullet \mathsf{while}\ t\ s'''; \ s'')\ \mathsf{else}\ s''$.

By Lemma 8 on [10], [11] and [8], we have

(14) $\Gamma, X \vdash (s'''\bullet \mathsf{while}\ t\ s'''), X_1$

(15) $X \subseteq X_1$

From [5] and [12], we have

(16) $\Gamma, X \vdash s'', X''$

By Lemma 6 on [14], [15] and [16], we have

(17) $\Gamma, X \vdash (s'''\bullet \mathsf{while}\ t\ s'''; \ s''), X_2$

(18) $X'' \subseteq X_2$

By the rule T-IF on [9], [17], [16], we have

(19) $\Gamma, X \vdash \mathsf{if}\ t\ \mathsf{then}\ (s'''\bullet \mathsf{while}\ t\ s'''; \ s'')\ \mathsf{else}\ s'', X_2 \cap X''$

From [19], [13] and [18], we have

$\Gamma, X \vdash s', X''$

The assumed consistency condition [3] is preserved since the store σ stays unchanged in the step.

■

B.4 Proof of Soundness

Lemma 18 (Soundness of Type Inference for Terms)

For every Γ, X, t, T, C *and* m,

if

$\Gamma, X \vdash t : T \mid C$

m *is a model for* C

then

$m(\Gamma), X \vdash t : m(T)$

Proof.

Hypothesis:

(1) $\Gamma, X \vdash t : T \mid C$

(2) m is a model for C

Structural induction on [1]:

Case the rule CT-VAR:

Trivial

Case the rule CT-MATH:

(3) $\Gamma, X \vdash t_1 : T_1 \mid C_1$

(4) $\Gamma, X \vdash t_2 : T_2 \mid C_2$

(5) $C = C_1 \cup C_2 \cup \{T_1 = T_2 = \mathbb{N} \vee T_1 = T_2 = \mathbb{R}\}$

(6) $T = T_1$

(7) $t = t_1 \oplus t_2$

From [2] and [5]:

(8) m is a model for C_1.

(9) m is a model for C_2.

By I.H. on ([3], [8]), ([4], [9]):

(10) $m(\Gamma), X \vdash t_1 : m(T_1)$

(11) $m(\Gamma), X \vdash t_2 : m(T_2)$

From [5], [2]:

$m(T_1) = m(T_2) = \mathbb{N} \ \vee \ m(T_1) = m(T_2) = \mathbb{R}$

We consider the first disjunct. The case for the second one is similar.

(12) $m(T_1) = m(T_2) = \mathbb{N}$

From [12], [10] and [11]:

(13) $m(\Gamma), X \vdash t_1 : \mathbb{N} \ \wedge \ m(\Gamma), X \vdash t_2 : \mathbb{N}$

By the rule T-Math on [13], [14]:

(15) $m(\Gamma), X \vdash t_1 \oplus t_2 : \mathbb{N}$

From [15], [6], [7] and [12]:

(16) $m(\Gamma), X \vdash t : m(T)$

Case the rule CT-Detect:

Similar to the previous case. The only interesting step is that considering the syntax of types T, the equality $m(T_1) \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset$ implies $m(T_1) = \cup\overline{Mat_i}$.

Case the rule CT-MAT:

Trivial

Case the rule CT-REAL:

Trivial

Case the rule CT-NAT:

Trivial

■

Lemma 19 (Soundness of Type Inference for Statements)

For every Γ, X, s, C, X'', *and* m,

if

$\Gamma, X \vdash s, X'' \mid C$

m *is a model for* C

then

$m(\Gamma), X \vdash s, X''$

Proof.

Hypothesis:

(1) $\Gamma, X \vdash s \mid C$

(2) m is a model for C

Structural induction on [1]:

Case the rule CT-Skip:

Trivial.

Case the rule CT-Inst:

(3) $s = i; s'$

(4) $\Gamma, X \vdash i, X' \mid C_1$

(5) $\Gamma, X' \vdash s', X'' \mid C_2$

(6) $C = C_1 \cup C_2$

From [2] and [6]:

(7) m is a model for C_1

(8) m is a model for C_2

By I.H. on [5]:

(9) $m(\Gamma), X' \vdash s, X''$

We need to show

(10) $m(\Gamma), X \vdash i, X'$

and then by the rule T-Inst on [9], [10]:

(11) $m(\Gamma), X \vdash i; s', X''$

and then by from [10], [11]:

$m(\Gamma), X \vdash s, X''$

Now assuming [4] and [7] that is

(4) $\Gamma, X \vdash i, X' \mid C_1$

(7) m is a model for C_1

we show that

$m(\Gamma), X \vdash i, X'$

Case analysis on [4]:

Case the rule CT-Assign-1:

(12) $i = (x := v)$

(13) $x : T \in \Gamma$

(14) $\Gamma, X \vdash v : T' \mid C'$

(15) $X' = X \cup \{x\}$

(16) $C_1 = C' \cup \{T' \subseteq T\}$

From [7] and [16]:

(17) m is a model for C'.

(18) $m(T') \subseteq m(T)$

From [13]:

(19) $x : m(T) \in m(\Gamma)$

From Lemma 18 on [14] and [17]:

(20) $m(\Gamma), X \vdash mat : m(T')$

By the rule T-Assign on [19], [20], and [18]:

(21) $m(\Gamma), X \vdash x := mat, X \cup \{x\}$

From [21], [15] and [12]:

$m(\Gamma), X \vdash i, X'$

Case the rule CT-Assign-2:

Similar to the previous case. The the rule T-Assign-2 is used.

Case the rule CT-Assign-3:

Similar to the case the rule CT-Assign-1. To use the rule T-Assign, in contrast to the case the rule CT-Assign-1 that proved $t = v$, the disjunct $T' = \mathbb{R} \vee T' = \mathbb{N}$ is proved to use the rule T-Assign-3.

Case the rule CT-Mix:

(12) $\Gamma, X \vdash x_1 : T \mid C$

(13) $\Gamma, X \vdash x_2 : T' \mid C'$

(14) $\Gamma, X \vdash t : T'' \mid C''$

(15) $i = (x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t)$

(16) $X' = X \setminus \{x_1, x_2\} \cup \{x\}$

(17) $C_1 = C \cup C' \cup C'' \cup \{T \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset,\ T' \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset,\ T'' = \mathbb{R}$

$$\overline{Mat_i \in T \wedge Mat_j \in T' \Rightarrow \mathsf{interact\text{-}abs}(Mat_i, Mat_j) \subseteq \Gamma(x)}\}$$

From [7] and [17]:

(18) m is a model for C.

(19) m is a model for C'.

(20) m is a model for C''.

(21) $m(T) \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset$

(22) $m(T') \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset$

(23) $m(T'') = \mathbb{R}$

(24) $\overline{Mat_i \in m(T) \wedge Mat_j \in m(T') \Rightarrow \mathsf{interact\text{-}abs}(Mat_i, Mat_j) \subseteq m(\Gamma(x))}$

From Lemma 18 on [12] and [18]:

(25) $m(\Gamma), X \vdash x_1 : m(T)$

From Lemma 18 on [13] and [19]:

(26) $m(\Gamma), X \vdash x_2 : m(T')$

From Lemma 18 on [14] and [20]:

(27) $m(\Gamma), X \vdash t : m(T'')$

From [21], there exists is such that:

(28) $m(T) = \cup\overline{Mat_i}$

From [22], there exists js such that:

(29) $m(T') = \cup\overline{Mat_j}$

From [25] and [28]:

(30) $m(\Gamma), X \vdash x_1 : \cup\overline{Mat_i}$

From [26] and [29]:

(31) $m(\Gamma), X \vdash x_2 : \cup\overline{Mat_j}$

From [27] and [23]:

(32) $m(\Gamma), X \vdash t : \mathbb{R}$

From [24], [28], and [29]:

(33) $\overline{\mathsf{interact\text{-}abs}(Mat_i, Mat_j) \subseteq \Gamma(x)}$

By the rule T-Mix on [30], [31], [32], and [33]:

(34) $\Gamma, X \vdash x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t, X \setminus \{x_1, x_2\} \cup \{x\}$

From [34], [15] and [16]:

(34) $\Gamma, X \vdash i, X'$

Case the rule CT-Split:

Similar to the case for the rule CT-Mix. The Lemma 18 is applied to the constraint typing judgement for x.

Case the rule CT-If:

Immediate from applying the Lemma 18 on t and the induction hypothesis on s_1 and s_2.

Case the rule CT-While:

Immediate from applying the Lemma 18 on t and the induction hypothesis on s.

■

B.5 Proof of Completeness

Lemma 20 (Completeness of Type Inference for Terms)

For all Γ, X, t, T, *and* C,

if

$m(\Gamma), X \vdash t : T$

$\Gamma, X \vdash t : T' \mid C$

then

m *is a model for* C

$m(T') = T$

Proof.

Hypothesis

(1) $m(\Gamma), X \vdash t : T$

(2) $\Gamma, X \vdash t : T' \mid C$

Proof by induction on the given constraint typing derivation [2].

Case for the rule CT-Var:

(3) $t = x$

(4) $x : T \in \Gamma$

(5) $x \in X$

(6) $C = \emptyset$

By the inversion Lemma 10 on [1]:

(7) $x : T' \in m(\Gamma)$

(8) $x \in X$

From [4] and [7]:

(7) $T' = m(T)$

The conclusion is immediate from [6] and [7].

Case for the rule CT-MATH:

(3) $\Gamma, X \vdash t_1 : T_1 \mid C_1$

(4) $\Gamma, X \vdash t_2 : T_2 \mid C_2$

(5) $C = C_1 \cup C_2 \cup \{T_1 = T_2 = \mathbb{N} \vee T_1 = T_2 = \mathbb{R}\}$

(6) $T' = T_1$

By the inversion Lemma 10 on [1]:

(7) $m(\Gamma), X \vdash t_1 : T$

(8) $m(\Gamma), X \vdash t_2 : T$

$T = \mathbb{N} \vee T = \mathbb{R}$

We consider the case for the first disjunct:

(The case for the second one is similar.)

(9) $T = \mathbb{N}$

By induction hypothesis on [7] and [3]:

(10) m is a model for C_1

(11) $m(T_1) = T$

By induction hypothesis on on [8] and [4]:

(12) m is a model for C_2

(13) $m(T_2) = T$

From [11], [13] and [9]:

(14) $m(T_1) = m(T_2) = \mathbb{N}$

From [5], [10], [12] and [14]:

(15) m is a model for C.

From [6], [11], and [9]:

(16) $m(T') = T$

The conclusion is [15] and [16].

Case for the rule CT-DETECT:

Similar to the case for the rule T-MATH. By induction hypothesis on x and t.

Case for the rule CT-MAT

Trivial.

Case for the rule CT-REAL:

Trivial.

Case for the rule CT-Nat:

Trivial.

■

Lemma 21 (Completeness of Type Inference for Statements)

For all Γ, X, s, X', C, *and* m,

if

$m(\Gamma), X \vdash s, X'$

$\Gamma, X \vdash s, X'' \mid C$

then

m *is a model for* C.

Proof.

Hypothesis

(1) $m(\Gamma), X \vdash s, X_1$

(2) $\Gamma, X \vdash s, X_2 \mid C$

We show that

m is a model for C.

Induction on the derivation of [2]:

Case for the rule CT-Skip:

Trivial.

Case for the rule CT-INST:

(3) $s = i; s'$

(4) $\Gamma, X \vdash i, X' \mid C_1$

(5) $\Gamma, X' \vdash s', X_2 \mid C_2$

(6) $C = C_1 \cup C_2$

From [1] and [3]:

(7) $m(\Gamma), X \vdash i; s', X_1$

By the inversion Lemma 11 on [7]:

(8) $m(\Gamma), X \vdash i, X''$

(9) $m(\Gamma), X'' \vdash s', X_2$

By induction hypothesis on [9] and [5]:

(10) m is a model for C_2

We will show that from [8] and [4]:

(11) m is a model for C_1

From [6], [10] and [11]:

m is a model for C

Hypothesis:

(8) $m(\Gamma), X \vdash i, X''$

(4) $\Gamma, X \vdash i, X' \mid C_1$

Conclusion:

m is a model for C_1

Case analysis on derivation of [4]:

Case for the rule CT-Assign-1:

(12) $i = (x := v)$

(13) $x : T \in \Gamma$

(14) $\Gamma, X \vdash v : T' \mid C'$

(15) $X' = X \cup \{x\}$

(16) $C_1 = C' \cup \{T' \subseteq T\}$

From [8] and [12]:

(17) $m(\Gamma), X \vdash x := v, X''$

By the inversion Lemma 11 on [17]:

(18) $x : T'' \in m(\Gamma)$

(19) $m(\Gamma), X \vdash v : T'''$

(22) $T''' \subseteq T''$

(23) $X'' = X \cup \{x\}$

From [18], [13]:

(24) $m(T) = T''$

By Lemma 20 on [19], [14]:

(25) m is a model for C'

(26) $m(T') = T'''$

From [22], [24] and [26]

(27) $m(T') \subseteq m(T)$

From [16], [24], [25] and [27]

m is a model for C_1

Case for the rule CT-Assign-2:

Similar to the case for the rule CT-Assign-1.

Case for the rule CT-Assign-3:

Similar to the case for the rule CT-Assign-1.

Case for the rule CT-Mix:

(12) $i = (x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t)$

(12) $\Gamma, X \vdash x_1 : T \mid C$

(13) $\Gamma, X \vdash x_2 : T' \mid C'$

(14) $\Gamma, X \vdash t : T'' \mid C''$

(15) $C_1 = C \cup C' \cup C'' \cup$

$$\{T \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset,\ T' \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset,\ T'' = \mathbb{R}$$

$$\overline{Mat_k \in T \wedge Mat_l \in T' \Rightarrow \mathsf{interact\text{-}abs}(Mat_k, Mat_l) \subseteq \Gamma(x)}\}$$

From [8] and [12]:

(16) $m(\Gamma), X \vdash x := \mathsf{mix}\ x_1\ \mathsf{with}\ x_2\ \mathsf{for}\ t, X''$

By the inversion Lemma 11 on [16]:

(17) $m(\Gamma), X \vdash x_1 : \cup\overline{Mat_i}$

(18) $m(\Gamma), X \vdash x_2 : \cup\overline{Mat_j}$

(19) $m(\Gamma), X \vdash t : \mathbb{R}$

(20) $\overline{\mathsf{interact\text{-}abs}(Mat_i, Mat_j) \subseteq m(\Gamma(x))}$

By Lemma 20 on [17], [12]:

(21) m is a model for C

(22) $m(T) = \cup\overline{Mat_i}$

By Lemma 20 on [18], [13]:

(23) m is a model for C'

(24) $m(T') = \cup\overline{Mat_j}$

By Lemma 20 on [19], [14]:

(25) m is a model for C''

(26) $m(T'') = \mathbb{R}$

From [22], [24], and [20]:

(27) $\overline{Mat_k \in m(T) \wedge Mat_l \in m(T') \Rightarrow \mathsf{interact\text{-}abs}(Mat_k, Mat_l) \subseteq m(\Gamma(x))}\}$

From [22], [24], and [26]:

(28) $m(T) \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset,\ m(T') \cap \{\mathbb{R}, \mathbb{N}\} = \emptyset,\ m(T'') = \mathbb{R}$

From [15], [21], [23], [25], [28] and [27]:

m is a model for C_1

Case for the rule CT-Split:

Similar to the case for the rule CT-Mix.

Case for the rule CT-If:

Immediate from the induction hypothesis on s_1 and s_2 and Lemma 20 on t.

Case for the rule CT-While:

Immediate from the induction hypothesis on s and Lemma 20 on t.

■

Appendix C

Syntax Study

C.1 Syntax Study: ELISA Protocols

The *enzyme-linked immunosorbent assay (ELISA)* is a test that uses antibodies and color changes to identify a substance. ELISA assays are commonly performed by interacting a group of chemicals with an immobile antibody to detect the presence and concentration of particular opiates. A technique was introduced, to allow these assays to be performed on DMFB technology by baking the enzyme directly onto the top plate of the DMFB, requiring further syntactic extensions to declare stationary substances and to explicitly move mobile substances to interact with them.

Fig. C.1 shows an example ELISA assay, which is one of the steps of the larger hierarchical opiate-detection decision tree. The operator will need to swap out top plates with different antigens for each ELISA assay, or use a very large electrowetting array to support a top plate to which all necessary antigens for all of the ELISA assays are affixed.

Biologists use many equivalent terms that mean the same thing as Mix. *BioScript* supports several of these terms, including Tap and Vortex, as shown in Fig. C.1. The execution engine recognizes these terms as being equivalent and converts them all to its own internal mix operation.

The repeat operation is part of *BioScript*'s core instruction set. It allows the programmer to specify a sequence of instructions that will be repeated a constant number of times, which is far more common in biochemistry than in general computer programming. Thus, instead of the complex, unconstrained loop syntax employed by modern programming languages, we have opted for a simple syntax to make *BioScript* accessible to scientists with limited programming experience.

The Incubate and Heat operations leverage the external heaters that are integrated into or placed near the DMFB. A DMFB that lacks heating capabilities cannot perform this ELISA assay. Thus, the compiler writer must specialize a language syntax for each DMFB variant based on its integrated peripherals.

The final operation in any ELISA is the detection phase that compares the reading obtained from the sample to a reading obtained from a control. This provides a certain measurement of the presence and concentration of the opiate in the sample. In the hierarchical decision tree immunoassay, the result of this comparison determines which ELISA assay to execute next.

```
Stationary: Anti-Fentanyl

Move 20uL of Urine Sample to Anti-Fentanyl
Move 100uL of Fentanyl-Conjugate to Anti-Fentanyl
Tap Anti-Fentanyl for 60s
Incubate Anti-Fentanyl at 23  C  for 60min
Drain Anti-Fentanyl

Repeat 6 times {
  Move 350uL of Distilled Water to Anti-Fentanyl
  Vortex Anti-Fentanyl for 45s
  Drain Anti-Fentanyl
}

Move 100uL of TMB Substrate to Anti-Fentanyl
Incubate Anti-Fentanyl at 23C for 30min
Mix Anti-Fentanyl with 100uL of Stop Reagent
                                          for 60s
Negative Reading = Measure the fluorescence of
                          Anti-Fentanyl for 30min
Drain Anti-Fentanyl
```

Figure C.1: Fentanyl ELISA

Appendix D

Type System Tests

D.1 Real world Assays

listings D.1 to D.4 are *BioScript* representations of real-world assays resulting in hazardous incidents[7] that put scientists' safety in jeopardy. *BioScript* could have prevented all of these incidents; guaranteeing the safety of the scientists executing these assays. We include the typing annotations to denote which chemical reactive group(s) a chemical belongs — or in *BioScript*'s context, the concrete type of the chemical.

Listing D.1: An incident involving the mixing tetrachloroethylene and nitric acid, mixing these chemicals would result in a corrosive, flammable, explosive, toxic gas.

```
1 [Halogenated Organic Compounds; Hydrocarbons, Aliphatic Unsaturated]
     manifest tetrachloroethylene
2 [Acids, Strong Oxidizing] manifest nitric_acid
3
4 instructions:
5
6 a = dispense tetrachloroethylene
7 b = dispense nitric_acid
8
9 c = mix a with b for 40s
```

Listing D.2: An incident involving the mixing methanol and nitric acid, mixing these chemicals would result in an explosive, toxic, flammable gas.

```
[Alcohols and Polyols] manifest methanol
[Acids, Strong Oxidizing] manifest nitric_acid

instructions:

a = dispense methanol
b = dispense nitric_acid

c = mix a with b for 40s
```

Listing D.3: While mixing diaminopropane and potassium hydride together results in a benign reaction, mixing the resultant mixture with nitrogen produces an intense explosion.

```
[Not Chemically Reactive] manifest nitrogen
[Amines, Phosphines, and Pyridines] manifest diaminopropane
[Metal Hydrides, Metal Alkyls, Metal Aryls, and Silanes] manifest
    potassium_hydride

instructions:

a = dispense nitrogen
b = dispense diaminopropane
c = dispense potassium_hydride

d = mix b with c for 20s
e = mix a with d for 15s
```

Listing D.4: An incident involving the mixing of dichlore and calcium hypo. Mixing these chemicals together results in a flammable, toxic explosion.

```
[Acids, Carboxylic; Halogenated Organic Compounds ] manifest dichlore
[Salts, Basic; Oxidizing Agents, Strong] manifest calcium_hypo

instructions:

a = dispense dichlore
b = dispense calcium_hypo

c = mix a with b for 20s
```

D.2 Synthetic Preventions

The examples in listings D.5 to D.11 are representative of synthetic assays that *BioScript*'s type system would prevent from executing. These assays result in hazardous incidents. Again, the typing annotations are included for demonstrating reactive groups to which the respective chemicals belong.

Listing D.5: Mixing hydrochloric acid with nitric acid results in a toxic, flammable, explosion.

```
1 [Acids, Strong Oxidizing] manifest nitric_acid
2 [Acids, Strong Non-oxidizing; Water and Aqueous Solutions] manifest
     hydrochloric_acid
3
4 instructions:
5
6 a = dispense nitric_acid
7 b = dispense hydrochloric_acid
8
9 c = mix a with b for 40s
```

Listing D.6: Mixing hydrochloric acid with isoproponal results in a excessive heat, ultimately yielding an explosion.

```
1 [Alcohols and Polyols] manifest isoproponal
2 [Acids, Strong Non-oxidizing; Water and Aqueous Solutions] manifest
     hydrochloric_acid
3
4 instructions:
5
6 a = dispense isoproponal
7 b = dispense hydrochloric_acid
8
9 c = mix a with b for 40s
```

Listing D.7: Formaldehye is self-reactive, yielding in explosive, toxic gases.

```
1 [Aldehydes; Polymerizable Compounds] manifest formaldehyde
2
3 instructions:
4
```

```
a = dispense formaldehyde
b = dispense formaldehyde

c = mix a with b for 40s
```

Listing D.8: Mixing benzoin with urea is safe. However, adding formaldehyde will result in an intense explosion.

```
[Alcohols and Polyols; Hydrocarbons, Aromatic; Ketones] manifest benzoin
[Amides and Imides] manifest urea
[Aldehydes; Polymerizable Compounds] manifest formaldehyde

instructions:

a = dispense benzoin
b = dispense urea
c = dispense formaldehyde

d = mix a with b for 40s
e = mix c with d for 40s
```

Listing D.9: Mixing formaldehyde with benzene is benign. However, adding more formaldehyde to the solution could result in an intense explosion, as formaldehyde is self-reactive.

```
[Hydrocarbons, Aromatic] manifest benzene
[Aldehydes; Polymerizable Compounds] manifest formaldehyde

instructions:

a = dispense benzene
b = dispense formaldehyde
c = dispense formaldehyde

d = mix a with b for 40s
e = mix c with d for 20s
```

Listing D.10: Reacting benzene with urea results in a benign solution. However, introducing hydrochloric acid will generate a toxic gas.

```
[Hydrocarbons, Aromatic] manifest benzene
[Amides and Imides] manifest urea
[Acids, Strong Non-oxidizing; Water and Aqueous Solutions] manifest
    hydrochloric_acid

instructions:

a = dispense benzene
```

```
b = dispense urea
c = dispense hydrochloric_acid

d = mix a with b for 40s
e = mix c with d for 20s
```

Listing D.11: Mixing hydrogen peroxide and sulfuric acid will result in a corrosive, toxic, explosive gas that will explode. Adding acetone will only further exacerbate the issue.

```
[Oxidizing Agents, Strong; Water and Aqueous Solutions] manifest
    hydrogen_peroxide
[Acids, Strong Oxidizing] manifest sulfuric_acid
[Ketones] manifest acetone

instructions:

a = dispense hydrogen_peroxide
b = dispense sulfuric_acid
c = dispense acetone

d = mix a with b for 20s
e = mix c with d for 10s
```

D.3 Synthetic Successes

listings D.12 and D.13 are example assays where *BioScript*'s type system will allow the assay to execute. We include typing annotations to describe the reactive groups the respective chemicals belong.

Listing D.12: Reacting benzene with urea results in a benign solution.

```
[Hydrocarbons, Aromatic] manifest benzene
[Amides and Imides] manifest urea

instructions:

a = dispense benzene
b = dispense urea

c = mix a with b for 40s
```

Listing D.13: Reacting benzene with urea results in a benign solution. Introducing benzotrichloride still produces a safe reaction.

```
[Hydrocarbons, Aromatic] manifest benzene
[Amides and Imides] manifest urea
[Aryl Halides] manifest benzotrichloride

instructions:

a = dispense benzene
b = dispense urea
c = dispense benzotrichloride

d = mix a with b for 40s
e = mix d with c for 20s
```

Appendix E

Languages

We now present common assays expressed in Antha[164], Aquacore[9], BioCoder[44], and *BioScript* which are used in laboratories:

- Glucose Detection: Appendices E.1.13, E.2.1, E.3.4 and E.4.1
- Polymerase Chain Reaction (PCR):Appendices E.1.14, E.2.2, E.3.1 and E.4.3
- Imaging Probe Synthesis: Appendices E.1.12, E.2.3, E.3.5 and E.4.2
- Neurotransmitting Sensing: Appendices E.1.15, E.2.4, E.3.6 and E.4.4
- Probabilistic PCR: Appendices E.1.2 and E.3.2
- PCR with droplet replenishment: Appendices E.1.1 and E.3.3

E.1 BioScript Assays

E.1.1 PCR droplet Replenishment

- PCR droplet Replenishment Assay:

  https://1drv.ms/v/s!AqwZF7jQGx_IhJRqfq1dLhbBUIL9RQ

```
manifest PCRMasterMix
manifest Template

instructions:

PCRMix = mix 50 uL of PCRMasterMix with 50 uL of Template for 1s

repeat 50 times {
    heat PCRMix at 95c for 20s
    volumeWeight = detect weight on PCRMix

    if (volumeWeight <= 50) {
        replacement = mix 25uL of PCRMasterMix with 25uL of Template for
            5s
        heat replacement at 95c for 45s
        PCRMix = mix PCRMix with replacement for 5s
    }

    heat PCRMix at 68c for 30s
    heat PCRMix at 95c for 45s
}

heat PCRMix at 68c for 5m
```

E.1.2 Probabilistic PCR

- Probabilistic PCR Full:

  https://1drv.ms/v/s!AqwZF7jQGx_IhJRo-xdbVT_Q6UsXPg

- Probabilistic PCR early exit:

  https://1drv.ms/v/s!AqwZF7jQGx_IhJRpLXveGI8Qr7BWrw

```
module fluorescence
manifest PCR_Master_Mix
manifest Buffer
manifest PCRMix

instructions:

PCR_Master_Mix = mix 50uL of PCRMix with 50uL of Buffer

heat PCR_Master_Mix at 94c for 2m

repeat 20 times {
    heat PCR_Master_Mix at 94c for 20s
    heat PCR_Master_Mix at 50c for 40s
}
DNA_Sensor = detect fluorescence on PCR_Master_Mix for 30s
if (DNA_Sensor <= 85) {
    dispose PCR_Master_Mix
}

repeat 20 times {
    heat PCR_Master_Mix at 94c for 20s
    heat PCR_Master_Mix at 50c for 40s
}
heat PCR_Master_Mix at 70c for 5m
```

E.1.3 Broad Spectrum Opiate

```
module fluorescence

manifest Anti_Morphine
manifest Anti_Oxy
manifest Anti_Fentanyl
manifest Anti_Ciprofloxcin
manifest Anti_Heroin
manifest UrineSample

instructions:

us1 = dispense 10uL of UrineSample
us2 = dispense 10uL of UrineSample
us3 = dispense 10uL of UrineSample
us4 = dispense 10uL of UrineSample
us5 = dispense 10uL of UrineSample

a = mix us1 with Anti_Morphine
b = mix us2 with Anti_Oxy
cc = mix us3 with Anti_Fentanyl
d = mix us4 with Anti_Ciprofloxcin
e = mix us5 with Anti_Heroin

MorphineReading = detect fluorescence on a for 5s
```

```
OxyReading = detect fluorescence on b for 5s
FentanylReading = detect fluorescence on cc for 5s
CiproReading = detect fluorescence on d for 5s
HeroinReading = detect fluorescence on e for 5s

dispose Anti_Morphine
dispose Anti_Oxy
dispose Anti_Fentanyl
dispose Anti_Ciprofloxcin
dispose Anti_Heroin
```

E.1.4 Ciprofloxacin eLISA

```
module fluorescence

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

instructions:

us = dispense 20uL 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 250uL of distilled_water
    cfe = dispense ciprofloxacin_enzyme
    temp = mix 250uL of water with cfe for 45s
    dipose temp
}
tmb = dispense 50uL of tmb_substrate
cfe = dispense ciprofloxacin_enzyme

d = mix tmb_substrate with cfe
heat d at 25c for 30m

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

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

E.1.5 Diazepam eLISA

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

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

hrpc = dispense 150uL 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 250uL of distilled_water
    dpe = dispense diazepam_enzyme
    a = mix water with dpe for 45s
    dispose a
}

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

stop = dispense 100uL of stop_reagent

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

E.1.6 5-4-7 Dilution

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

instructions:

first_dilute = mix substance_a with dilutant1
x = split first_dilute into 2

dispose x2

second_dilute = mix x1 with dilutant2
y = split second_dilute into 2
dispose y2

third_dilute = mix y1 with dilutant3
z = split third_dilute into 2
dispose z2

fourth_dilute = mix substance_b with substance_c
a = split fourth_dilute into 2
dispose a2

final_dilute = mix third_dilute with fourth_dilute
b = split final_dilute into 2
dispose b2
```

E.1.7 Fentanyl eLISA

```
module fluorescence

manifest antigen

manifest urine_sample
manifest fentanyl_conjugate
manifest tmb_substrate
manifest distilled_water
manifest stop_reagent

instructions:

a = mix 20uL of urine_sample with antigen
b = mix 100uL of fentanyl_conjugate with a for 60s
heat b at 23c for 60m
dispose b

repeat 6 times {
```

```
    z = mix 350uL of distilled_water with a for 45s
    dispose z
}

a = mix 100uL of tmb_substrate with a
heat a at 23c for 30m

a = mix a with 100uL of stop_reagent for 60s
negative_reading = detect fluorescence on a for 30m
dispose a
```

E.1.8 Morphine eLISA

```
module fluorescence

manifest morphine_enzyme

manifest urine_sample
manifest morphine_conjugate
manifest distilled_water
manifest tmb_substrate
manifest morphine_enzyme
manifest stop_reagent

instructions:

a = mix 20uL of urine_sample with morphine_enzyme
b = mix 100uL of morphine_conjugate with morphine_enzyme for 60s
heat b at 23c for 60m

repeat 6 times {
    b = mix 350uL of distilled_water with morphine_enzyme for 45s
    dispose b
}

cc = mix 100uL of tmb_substrate with morphine_enzyme
heat cc at 23c for 30m
d = mix cc with 100uL of stop_reagent for 60s

urine_reading = detect fluorescence on d for 30m
dispose d
```

E.1.9 Morphine eLISA with Control Samples

```
module fluorescence

manifest Antigen1
manifest Antigen2
manifest Antigen3

manifest morphine_conjugate
```

```
manifest negative_standard
manifest diluted_sample
manifest positive_standard
manifest distilled_water
manifest tmb_substrate
manifest stop_reagent

instructions:

a = mix 20uL of negative_standard with Antigen1
b = mix 20uL of positive_standard with Antigen2
cc = mix 20uL of diluted_sample with Antigen3

a = mix 100uL of morphine_conjugate with a for 60s
b = mix 100uL of morphine_conjugate with b for 60s
cc = mix 100uL of morphine_conjugate with cc for 60s

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 {
    aa = mix 350uL of distilled_water with Antigen1 for 45s
    bb = mix 350uL of distilled_water with Antigen2 for 45s
    cc = mix 350uL of distilled_water with Antigen3 for 45s

    dispose aa
    dispose bb
    dispose cc
}

aa = mix 100uL of tmb_substrate with Antigen1
bb = mix 100uL of tmb_substrate with Antigen2
cc = mix 100uL of tmb_substrate with Antigen3

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

aa = mix stop_reagent with 100uL of aa for 60s
bb = mix stop_reagent with 100uL of bb for 60s
cc = mix stop_reagent with 100uL 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
```

E.1.10 Heroin eLISA

```
module fluorescence

manifest heroin_enzyme
manifest heroin_conjugate
manifest urine_sample
manifest tmb_substrate
manifest distilled_water
manifest stop_reagent

instructions:

a = mix 20uL of urine_sample with 1uL of heroin_enzyme for 10s
a = mix 100uL of heroin_conjugate with a for 60s
heat a at 23c for 60m
dispose a

repeat 6 times {
    b = mix 350uL of distilled_water with heroin_enzyme for 45s
    dispose heroin_enzyme
}

cc = mix 100uL of tmb_substrate with heroin_enzyme
heat cc at 23c for 30m
cc = mix stop_reagent with 100uL of cc for 60s

urine_reading = detect fluorescence on cc for 30m
dispose cc
```

E.1.11 Oxycodone eLISA

```
module fluorescence

manifest heroin_enzyme
manifest oxycodone_enzyme

manifest oxycodone_conjugate
manifest urine_sample
manifest tmb_substrate
manifest distilled_water
manifest stop_reagent

instructions:

a = mix 20uL of urine_sample with oxycodone_enzyme
a = mix 100uL of oxycodone_conjugate with a for 60s
heat a at 23c for 60m
dispose a

repeat 6 times {
    a = mix 350uL of distilled_water with oxycodone_enzyme for 45s
```

```
    dispose a
}

b = mix 100uL of tmb_substrate with oxycodone_enzyme
heat b at 23c for 30m
b = mix 100uL of stop_reagent with b for 60s

urine_reading = detect fluorescence on oxycodone_enzyme for 30m
dispose b
```

E.1.12 Image Probe Synthesis

```
manifest ion_exchange_beads
manifest fluoride_ions_f
manifest mecn_solution
manifest VEXZGXHMUGYJMC-UHFFFAOYSA-N

instructions:

aa = mix 10uL of ion_exchange_beads with 10uL of fluoride_ions_f for 30s

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

aa = mix aa with 10uL of mecn_solution for 30s

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

aa = mix aa with 10uL of VEXZGXHMUGYJMC-UHFFFAOYSA-N for 60s
heat aa at 60c for 60s
```

E.1.13 Glucose detection

```
module fluorescence

manifest reagent
manifest glucose
manifest distilled_water
manifest Sample

instructions:

result1 = mix 10uL of glucose with 10uL of reagent for 10s
reading1 = detect fluorescence on result1 for 30s
a = mix distilled_water with reagent for 30s
dispose a

result2 = mix 10uL of glucose with 20uL of reagent for 10s
reading2 = detect fluorescence on result2 for 30s
```

```
a = mix distilled_water with reagent for 30s
dispose a

result3 = mix 10uL of glucose with 40uL of reagent for 10s
reading3 = detect fluorescence on result3 for 30s
a = mix distilled_water with reagent for 30s
dispose a

result4 = mix 10uL of glucose with 80uL of reagent for 10s
reading4 = detect fluorescence on result4 for 30s
a = mix distilled_water with reagent for 30s
dispose a

result5 = mix 10uL of Sample with 10uL of reagent for 10s
reading5 = detect fluorescence on result5 for 30s
a = mix distilled_water with reagent for 30s
```

E.1.14 PCR

```
module fluorescence
manifest pcr_mixture

instructions:

a = dispense pcr_mixture

heat a at 95c for 5s

repeat 20 times {
    heat a at 53c for 15s
    heat a at 72c for 10s
}

x = detect fluorescence on a for 3m

dispose a
```

E.1.15 Neurotransmitter Sensing

```
module Capillary_Electrophoresis

manifest Sample
manifest Reagent

instructions:

mixture = mix Sample with Reagent for 50s
Perform Capillary_Electrophoresis ( 9 cm at 223 V/cm) on Mixture
    Seperate with electrophoresis buffer
Measure the fluorescence of Mixture for 10s
```

E.2 AquaCore Assays

E.2.1 Glucose Detection

```
Input port ip1 ;Standard glucose
Input port ip2 ;Reagent
Input port ip3 ;Sample
RESULT[1..5] ; dry array for final results
glucose-detection {
input s1, ip1
input s2, ip2
input s3, ip3
move mixer1, s1, 1;5s
move mixer1, s2, 1;5s
mix mixer1, 10;10s
move sensor1, mixer1;5s
sense.OD sensor1, RESULT[1] ;30s

move mixer1, s1, 1 ;5s
move mixer1, s2, 2;5s
mix mixer1, 10;10s
move sensor1, mixer1;5s
sense.OD sensor1, RESULT[2] ;30s

move mixer1, s1, 1 ;5s
move mixer1, s2, 4 ;5s
mix mixer1, 10 ;10s
move sensor1, mixer1 ;5s
sense.OD sensor1, RESULT[3] ;30s

move mixer1, s1, 1 ;5s
move mixer1, s2, 8 ;5s
mix mixer1, 10 ;10s
move sensor1, mixer1 ;5s
sense.OD sensor1, RESULT[4] ;30s

<dry routine to get best line fit for RESULT[1..4]> move mixer1, s3, 1
    ;5s
move mixer1, s2, 1 ;5s
mix mixer1, 10 ;10s
move sensor1, mixer1 ;5s
sense.OD sensor1, RESULT[5] ;30s

<dry routine to get concentration from line given RESULT[5]>
}
```

E.2.2 PCR

```
Input port ip1 ;PCR mixture
```

```
Input port ip2 ;CE separation medium

RESULT[] ;dry array for final results

PCR {
input s1, ip1
input s2, ip2

move heater1, s1 ;5s
dry-mov r1, 20
dry-label loop:
    incubate heater1, 95, 5 ;6s
    incubate heater1, 53, 15 ;17s
    incubate heater1, 72, 10 ;12s
dry-dec r1
dry-bgt loop

move separator1.buf, s2 ;5s
move separator1, heater1 ;5s

separate.CE separator1, 236, 5, 180
sense.FL sensor1, RESULT ;180s
}
Total time = 895s
```

E.2.3 Imaging Probe Synthesis

```
Input port ip1 ;Ion exchange beads (in buffer)
Input port ip2 ;Fluoride ions F-
Input port ip3 ;MeCN solution
Input port ip4 ;HCl

Imaging-Probe-Synthesis {
input s1, ip1
input s2, ip2
input s3, ip3
input s4, ip4

move mixer1, s1
move mixer1, s2
mix mixer1, 30

move heater1, mixer1
concentrate.EV heater1, 100, 30
concentrate.EV heater1, 120, 30
concentrate.EV heater1, 135, 180

move mixer1, s3
move mixer1, heater1
mix mixer1, 30

move heater1, mixer1
incubate heater1, 100, 30
```

```
incubate heater1, 120, 50
move mixer1, heater1

move mixer1, s4
mix mixer1, 60
move heater1, mixer1
concentrate.EV heater1, 60, 60
}
Total time = 548s #reservoirs = 4
ASLoC area = 13mm x 9mm = 117 mm2
```

E.2.4 Neurotransmitter Sensing

```
Input port ip1 ;Sample
Input port ip2 ;Reagent(OPA)
Input port ip3 ;Elecrtrophoresis buffer

RESULT[] ;dry array for final results

neurotransmitter-sensing {
input s1, ip1
input s2, ip2
input s3, ip3

move mixer1, s1 ;5s
move mixer1, s2 ;5s
mix mixer1, 50 ;50s

move separator1.buf, s3 ;5s
move separator1, mixer1 ;5s

separate.CE separator1, 223, 9, 22
sense.FL sensor1, RESULT ;22s

}
Total time = 92s #reservoirs = 3
ASLoC area = 2.5cm x 1.5cm = 3.75cm2 + CE column
```

E.3 BioCoder Assays

E.3.1 PCR

```
void PCR(){
    BioSystem bioCoder;

    Fluid *PCRMix = bioCoder.new_fluid("PCRMasterMix", Volume(
        MICRO_LITER,10));
    Fluid *SeperationMedium = bioCoder.new_fluid("Seperation Medium",
        Volume(MICRO_LITER,10));
```

```
    Container* tube = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML);

    bioCoder.first_step();
    bioCoder.measure_fluid(PCRMix,tube);

    bioCoder.next_step();
    bioCoder.incubate(tube,95,Time(5,SECS));

    bioCoder.next_step();
    bioCoder.LOOP(20);

    bioCoder.next_step();
    bioCoder.incubate(tube,53,Time(15,SECS));

    bioCoder.next_step();
    bioCoder.incubate(tube,72,Time(10,SECS));
    bioCoder.END_LOOP();

    bioCoder.next_step();
    bioCoder.ce_detect(tube,5,236,SeperationMedium);

    bioCoder.next_step();
    bioCoder.measure_fluorescence(tube,Time(3,MINS));

    bioCoder.next_step();
    bioCoder.end_protocol();
}
```

E.3.2 Probabilistic PCR

```
void ProbablisticPCR()
{
    BioSystem bioCoder;

    Fluid *PCRMix = bioCoder.new_fluid("PCRMasterMix", Volume(
        MICRO_LITER,10));

    Container* tube = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML);

    bioCoder.first_step();
    bioCoder.measure_fluid(PCRMix,tube);

    for(int i = 0 ; i < initial; ++i) {
        bioCoder.next_step();
        bioCoder.store_for(tube,94,Time(SECS,45));

        bioCoder.next_step();
        bioCoder.store_for(tube,65,Time(SECS,45));
    }

    for(int i = initial; i <= Threshold; ++i) {
        std::cout <<i<<std::endl;
        bioCoder.next_step();
```

```
23          bioCoder.store_for(tube,94,Time(SECS,45));
24
25          bioCoder.next_step();
26          bioCoder.store_for(tube,65,Time(SECS,45));
27
28          bioCoder.next_step();
29          bioCoder.measure_fluorescence(tube,Time(SECS,5),"DNASensor");
30
31          bioCoder.IF("DNASensor",GREATER_THAN, .85);
32          for(int j = i; j < Total+(Threshold-i); ++j) {
33              bioCoder.next_step();
34              bioCoder.store_for(tube,94,Time(SECS,45));
35
36              bioCoder.next_step();
37              bioCoder.store_for(tube,65,Time(SECS,45));
38          }
39
40          bioCoder.next_step();
41          bioCoder.drain(tube,"Amplified PCR");
42          bioCoder.END_IF();
43
44      }
45
46      bioCoder.drain(tube,"waste");
47      bioCoder.end_protocol();
48
49      bioCoder.PrintLeveledProtocol();
50      bioCoder.PrintTree();
51      bioCoder.PrintTreeVisualization("ProbablisticPCR");
52  }
```

E.3.3 PCR Droplet Replenish

```
1   void PCRDropletReplacement()
2   {
3       int TotalThermo = 9;
4       BioSystem bioCoder;
5
6       Fluid *PCRMix = bioCoder.new_fluid("PCRMasterMix", Volume(
            MICRO_LITER,10));
7       Fluid *Template = bioCoder.new_fluid("Template", Volume(MICRO_LITER
            ,10));
8
9       Container* tube = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML);
10      //Container* tube2 = bioCoder.new_container(
            STERILE_MICROFUGE_TUBE2ML);
11
12      bioCoder.first_step();
13      bioCoder.measure_fluid(PCRMix,tube);
14
15      bioCoder.next_step();
16      bioCoder.vortex(tube,Time(SECS,1));
17      bioCoder.measure_fluid(Template,tube);
```

```
18
19      bioCoder.next_step();
20      bioCoder.vortex(tube, Time(SECS,1));
21
22      bioCoder.next_step();
23      bioCoder.store_for(tube,95,Time(SECS,45));
24
25      bioCoder.next_step();
26      bioCoder.LOOP(TotalThermo);
27
28      std::cout<<"Debug statement2"<<std::endl;
29          bioCoder.next_step();
30          bioCoder.store_for(tube,95,Time(SECS,20));
31
32          bioCoder.next_step();
33          bioCoder.weigh(tube,"weightSensor");
34
35          bioCoder.next_step();
36          bioCoder.IF("WieghtSensor",LESS_THAN, 3.57);
37          bioCoder.next_step();
38          bioCoder.measure_fluid(PCRMix, tube);
39
40          bioCoder.next_step();
41          bioCoder.store_for(tube, 95,Time(SECS,45));
42
43          bioCoder.next_step();
44          bioCoder.vortex(tube, Time(SECS,1));
45          bioCoder.END_IF();
46
47          bioCoder.next_step();
48          bioCoder.store_for(tube,50,Time(SECS,30));
49
50          bioCoder.next_step();
51          bioCoder.store_for(tube,68,Time(SECS,45));
52          std::cout<<"Debug statement3"<<std::endl;
53          bioCoder.END_LOOP();
54          std::cout<<"Debug statement4"<<std::endl;
55
56
57      bioCoder.next_step();
58      bioCoder.store_for(tube,68,Time(MINS,5));
59      std::cout<<"Debug statement5"<<std::endl;
60      bioCoder.next_step();
61      bioCoder.drain(tube,"PCR");
62      std::cout<<"Debug statement6"<<std::endl;
63      bioCoder.end_protocol();
64
65
66      std::cout<<"Debug statemen7"<<std::endl;
67      bioCoder.PrintLeveledProtocol();
68      bioCoder.PrintTree();
69      bioCoder.PrintTreeVisualization("PCRReplacement");
70
71  }
```

E.3.4 Glucose Detection

```
1  void GlucoseDetection(){
2      BioSystem bioCoder;
3
4      Fluid *Glucose = bioCoder.new_fluid("Ion exchange beads", Volume(
           MICRO_LITER,160));
5      Fluid *Reagent = bioCoder.new_fluid("Fluoride ions",Volume(
           MICRO_LITER,50));
6      Fluid *Sample = bioCoder.new_fluid("HCL",Volume(MICRO_LITER,10));
7
8      Container* tube = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML);
9      Container* tube2 = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML)
           ;
10     Container* tube3 = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML)
           ;
11     Container* tube4 = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML)
           ;
12     Container* tube5 = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML)
           ;
13
14     bioCoder.first_step();
15     bioCoder.measure_fluid(Glucose,Volume(MICRO_LITER,10),tube);
16     bioCoder.measure_fluid(Glucose,Volume(MICRO_LITER,10),tube2);
17     bioCoder.measure_fluid(Glucose,Volume(MICRO_LITER,10),tube3);
18     bioCoder.measure_fluid(Glucose,Volume(MICRO_LITER,10),tube4);
19     bioCoder.measure_fluid(Glucose,Volume(MICRO_LITER,10),tube5);
20
21     bioCoder.measure_fluid(Reagent,Volume(MICRO_LITER,10),tube);
22     bioCoder.measure_fluid(Reagent,Volume(MICRO_LITER,20),tube2);
23     bioCoder.measure_fluid(Reagent,Volume(MICRO_LITER,40),tube3);
24     bioCoder.measure_fluid(Reagent,Volume(MICRO_LITER,80),tube4);
25
26     bioCoder.measure_fluid(Sample,Volume(MICRO_LITER,80),tube5);
27
28     bioCoder.next_step();
29     bioCoder.measure_fluorescence(tube,Time(5,SECS));
30     bioCoder.measure_fluorescence(tube2,Time(5,SECS));
31     bioCoder.measure_fluorescence(tube3,Time(5,SECS));
32     bioCoder.measure_fluorescence(tube4,Time(5,SECS));
33     bioCoder.measure_fluorescence(tube5,Time(5,SECS));
34
35     bioCoder.next_step();
36     bioCoder.end_protocol();
37 }
```

E.3.5 Image Probe Synthesis

```
1  void ImageProbSynthesis(){
2      BioSystem bioCoder;
3
```

```
4       Fluid *IonBeads = bioCoder.new_fluid("Ion exchange beads", Volume(
            MICRO_LITER,10));
5       Fluid *Fluoride = bioCoder.new_fluid("Fluoride ions",Volume(
            MICRO_LITER,10));
6       Fluid *HCL = bioCoder.new_fluid("HCL",Volume(MICRO_LITER,10));
7       Fluid *MeCNSolution = bioCoder.new_fluid("MeCN solution",Volume(
            MICRO_LITER,10));
8
9       Container* tube = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML);
10
11      bioCoder.first_step();
12      bioCoder.measure_fluid(IonBeads,tube);
13      bioCoder.measure_fluid(Fluoride,tube);
14
15      bioCoder.next_step();
16      bioCoder.vortex(tube,Time(30,SECS));
17
18      bioCoder.next_step();
19      bioCoder.incubate(tube,100,Time(30,SECS));
20
21      bioCoder.next_step();
22      bioCoder.incubate(tube,120,Time(30,SECS));
23
24      bioCoder.next_step();
25      bioCoder.incubate(tube,135, Time(3,MINS));
26
27      bioCoder.next_step();
28      bioCoder.measure_fluid(MeCNSolution,tube);
29
30      bioCoder.next_step();
31      bioCoder.vortex(tube,Time(30,SECS));
32
33      bioCoder.next_step();
34      bioCoder.incubate(tube,100,Time(30,SECS));
35
36      bioCoder.next_step();
37      bioCoder.incubate(tube,120,Time(50,SECS));
38
39      bioCoder.next_step();
40      bioCoder.measure_fluid(HCL,tube);
41
42      bioCoder.next_step();
43      bioCoder.vortex(tube,Time(60,SECS));
44
45      bioCoder.next_step();
46      bioCoder.incubate(tube,60,Time(60,SECS));
47
48      bioCoder.next_step();
49      bioCoder.end_protocol();
50  }
```

E.3.6 Neurotransmitter Sensing

```
void neurotransmitterSensing(){
    BioSystem bioCoder;

    Fluid *Sample = bioCoder.new_fluid("Sample", Volume(MICRO_LITER,10))
        ;
    Fluid *Reagent = bioCoder.new_fluid("Reagent",Volume(MICRO_LITER,10)
        );
    Fluid *SeperationMedium = bioCoder.new_fluid("Seperation Medium",
        Volume(MICRO_LITER,10));

    Container* tube = bioCoder.new_container(STERILE_MICROFUGE_TUBE2ML);

    bioCoder.first_step();
    bioCoder.measure_fluid(Sample,tube);
    bioCoder.measure_fluid(Reagent,tube);

    bioCoder.next_step();
    bioCoder.vortex(tube,Time(50,SECS));

    bioCoder.next_step();
    bioCoder.ce_detect(tube,9,223,SeperationMedium);

    bioCoder.next_step();
    bioCoder.measure_fluorescence(tube,Time(10,SECS));

    bioCoder.next_step();
    bioCoder.end_protocol();
}
```

E.4 Antha Assays

E.4.1 Glucose Detection

```
protocol Glucose_Detection
import (
    "github.com/antha-lang/antha/antha/anthalib/wtype"
    // LHComponent type
    "github.com/antha-lang/antha/antha/anthalib/wutil"
    "github.com/antha-lang/antha/antha/anthalib/mixer"
    //sample function is imported from mixed
// Input parameters
Parameters (
    Reagent_volume1 Volume  // 10ul
    Reagent_volume2 Volume  // 20ul
    Reagent_volume3 Volume  // 40ul
    Reagent_volume4 Volume  // 80ul
    Glucose_volume Volume   // 10ul
    Sample_volume Volume    // 10ul
)
// Data which is returned from this protocol, and data types
Data ()
```

```
// Physical Inputs to this protocol with types
Inputs (
    Glucose *wtype.LHComponent
    Reagent *wtype.LHComponent
    Sample *wtype.LHComponent
)
// Physical outputs from this protocol with types
Outputs (
    Result1 *wtype.LHComponent
    Result2 *wtype.LHComponent
    Result3 *wtype.LHComponent
    Result4 *wtype.LHComponent
    Result5 *wtype.LHComponent
)

Requirements {}
// Conditions to run on startup
Setup {}
// The core process for this protocol, with the steps to be performed
// for every input
Steps {

    glucose := mixer.Sample(Glucose, Glucose_volume)
    reagent := mixer.Sample(Reagent, Reagent_volume1)

    Result1 = mixer.Mix(glucose, reagent) // cannot specify duration of
        mixture
    //cannot measure fluorescence to get a reading

    glucose := mixer.Sample(Glucose, Glucose_volume)  //get new sample
        of Glucose
    reagent := mixer.Sample(Reagent, Reagent_volume2) //get new sample
        of Reagent

    Result2 = mixer.Mix(glucose, reagent) // mix new samples
    // would measure fluorescence here

    glucose := mixer.Sample(Glucose, Glucose_volume)  //get new sample
        of Glucose
    reagent := mixer.Sample(Reagent, Reagent_volume3) //get new sample
        of Reagent

    Result3 = mixer.Mix(glucose, reagent) // mix new samples
    // would measure fluorescence here

    glucose := mixer.Sample(Glucose, Glucose_volume)  //get new sample
        of Glucose
    reagent := mixer.Sample(Reagent, Reagent_volume4) //get new sample
        of Reagent

    Result4 = mixer.Mix(glucose, reagent) // mix new samples
    // would measure fluorescence here
    sample := mixer.Sample(Sample, Sample_volume)  //get new sample of
        Sample
```

```
    reagent := mixer.Sample(Reagent, Reagent_volume1) //get new sample
        of Reagent
    Result5 = mixer.Mix(sample, reagent) // mix new samples
    // would measure fluorescence here
}
Analysis {}
Validation {}
```

E.4.2 Imaging Probe Synthesis

```
protocol Imaging_Probe_Synthesis
import (
    "github.com/antha-lang/antha/antha/anthalib/wtype" // LHComponent
        type
    "github.com/antha-lang/antha/antha/anthalib/wutil"
    "github.com/antha-lang/antha/antha/anthalib/mixer"
    //sample function is imported from mixed

// Input parameters
Parameters (
    Ion_volume Volume
    Flouride_volume Volume
    MeCN_volume Volume
    HCl_volume Volume
)

// Data which is returned from this protocol, and data types
Data ()
// Physical Inputs to this protocol with types
Inputs (
    Ion_exchange_beads *wtype.LHComponent
    Fluoride_ions_F- *wtype.LHComponent
    MeCN *wtype.LHComponent
    HCl *wtype.LHComponent
)
// Physical outputs from this protocol with types
Outputs (
    Mixture *wtype.LHComponent
)
Requirements {}
// Conditions to run on startup
Setup {}
// The core process for this protocol, with the steps to be performed
// for every input
Steps {
    ibv := mixer.Sample(Ion_exchange_beads, Ion_volume)
    fif := mixer.Sample(Flourid_ions_F-, Flouride_volume)

    mixture := mixer.Mix(ibv, fif) //cannot specify time for mixture

    mixture = Incubate(mixture, 100, 30, false) //heat 100 for 30s
    mixture = Incubate(mixture, 120, 30, false) //heat 120 for 30s
```

```
    mixture = Incubate(mixture, 135, 180, false) //heat 135 for 3
        minutes

    mecn := mixer.Sample(MeCN, MeCN_volume)

    mixture = mixer.Mix(mixture, mecn)

    mixture = Incubate(mixture, 100, 30, false) //incubate 100 for 30s
    mixture = Incubate(mixture, 120, 50, false) //incubuate 120 for 50s

    hcl := mixer.Sample(HCl, HCl_volume)

    mixture = mixer.Mix(mixture, hcl)

    mixture = Incubate(mixture, 60, 60, false) // heat 60 for 60s

    Mixture = mixture
}

Analysis {}

Validation {}
```

E.4.3 PCR

```
protocol PCR

import (
    "github.com/antha-lang/antha/antha/anthalib/wtype"
    "github.com/antha-lang/antha/antha/anthalib/mixer"
    "github.com/antha-lang/antha/antha/AnthaStandardLibrary/Packages/
        enzymes"
    "fmt"
    "github.com/antha-lang/antha/antha/AnthaStandardLibrary/Packages/
        text"
    "github.com/antha-lang/antha/antha/AnthaStandardLibrary/Packages/
        search"
    "github.com/antha-lang/antha/antha/AnthaStandardLibrary/Packages/
        sequences"
)

/*type Polymerase struct {
    wtype.LHComponent
    Rate_BPpers float64
    Fidelity_errorrate float64 // could dictate how many colonies are
        checked in validation!
    Extensiontemp Temperature
    Hotstart bool
    StockConcentration Concentration // this is normally in U?
    TargetConcentration Concentration
    // this is also a glycerol solution rather than a watersolution!
}
*/
```

```
// Input parameters for this protocol (data)
Parameters (
    // PCRprep parameters:
    ReactionVolume Volume
    FwdPrimerConc Concentration
    RevPrimerConc Concentration
    Additiveconc Concentration
    TargetpolymeraseConcentration Concentration
    Templatevolume Volume
    DNTPconc Concentration

    // Reaction parameters: (could be a entered as thermocycle
        parameters type possibly?)
    Numberofcycles int
    InitDenaturationtime Time
    Denaturationtime Time
    //Denaturationtemp Temperature
    Annealingtime Time
    AnnealingTemp Temperature // Should be calculated from primer and
        template binding
    Extensiontime Time // should be calculated from template length and
        polymerase rate
    Extensiontemp Temperature
    Finalextensiontime Time
    Targetsequence string
    FwdPrimerSeq string
    RevPrimerSeq string
)
// Data which is returned from this protocol, and data types
Data (
    FwdPrimerSites []search.Thingfound
    RevPrimerSites []search.Thingfound
)
// Physical Inputs to this protocol with types
Inputs (
    FwdPrimer *wtype.LHComponent
    RevPrimer *wtype.LHComponent
    DNTPS *wtype.LHComponent
    PCRPolymerase *wtype.LHComponent
    Buffer *wtype.LHComponent
    Template *wtype.LHComponent
    Additives []*wtype.LHComponent // e.g. DMSO
    OutPlate *wtype.LHPlate
)

// Physical outputs from this protocol with types
Outputs (
    Reaction *wtype.LHComponent
)
Requirements {}
// Conditions to run on startup
Setup {}
// The core process for this protocol, with the steps to be performed
// for every input
Steps {
```

```
76      FwdPrimerSites = sequences.FindSeqsinSeqs(Targetsequence, []string{
            FwdPrimerSeq})
77      RevPrimerSites = sequences.FindSeqsinSeqs(Targetsequence, []string{
            RevPrimerSeq})
78
79      if len(FwdPrimerSites)==0 || len(RevPrimerSites)==0{
80          errordescription := fmt.Sprint(
81              text.Print("FwdPrimerSitesfound:", fmt.Sprint(FwdPrimerSites
                    )),
82              text.Print("RevPrimerSitesfound:", fmt.Sprint(RevPrimerSites
                    )),
83          )
84          Errorf(errordescription)
85      }
86      // Mix components
87      samples := make([]*wtype.LHComponent, 0)
88      bufferSample := mixer.SampleForTotalVolume(Buffer, ReactionVolume)
89      samples = append(samples, bufferSample)
90      templateSample := mixer.Sample(Template, Templatevolume)
91      samples = append(samples, templateSample)
92      dntpSample := mixer.SampleForConcentration(DNTPS, DNTPconc)
93      samples = append(samples, dntpSample)
94      FwdPrimerSample := mixer.SampleForConcentration(FwdPrimer,
            FwdPrimerConc)
95      samples = append(samples, FwdPrimerSample)
96      RevPrimerSample := mixer.SampleForConcentration(RevPrimer,
            RevPrimerConc)
97      samples = append(samples, RevPrimerSample)
98
99      for _, additive := range Additives {
100         additiveSample := mixer.SampleForConcentration(additive,
                Additiveconc)
101         samples = append(samples, additiveSample)
102     }
103     polySample := mixer.SampleForConcentration(PCRPolymerase,
            TargetpolymeraseConcentration)
104     samples = append(samples, polySample)
105     reaction := MixInto(OutPlate, "", samples...)
106     // thermocycle parameters called from enzyme lookup:
107     polymerase := PCRPolymerase.CName
108     extensionTemp := enzymes.DNApolymerasetemps[polymerase]["
            extensiontemp"]
109     meltingTemp := enzymes.DNApolymerasetemps[polymerase]["meltingtemp"]
110     // initial Denaturation
111     r1 := Incubate(reaction, meltingTemp, InitDenaturationtime, false)
112
113     for i:=0; i < Numberofcycles; i++ {
114         // Denature
115         r1 = Incubate(r1, meltingTemp, Denaturationtime, false)
116         // Anneal
117         r1 = Incubate(r1, AnnealingTemp, Annealingtime, false)
118         //extensiontime := TargetTemplatelengthinBP/PCRPolymerase.
                RateBPpers
119         // we'll get type issues here so leave it out for now
120         // Extend
```

```
        r1 = Incubate(r1, extensionTemp, Extensiontime, false)
        }
        // Final Extension
        r1 = Incubate(r1, extensionTemp, Finalextensiontime, false)
        // all done
        Reaction = r1
}

// Run after controls and a steps block are completed to
// post process any data and provide downstream results
Analysis {}
// A block of tests to perform to validate that the sample was processed correctly
// Optionally, destructive tests can be performed to validate results on a
// dipstick basis
Validation {}
```

E.4.4 Neurotransmitter Sensing

```
protocol Neurotransmitter_Sensing

import (
    "github.com/antha-lang/antha/antha/anthalib/wtype" // LHComponent type
    "github.com/antha-lang/antha/antha/anthalib/wutil"
    "github.com/antha-lang/antha/antha/anthalib/mixer" //sample function is imported from mixed

// Input parameters
Parameters (
    Sample_volume //no val specified
    Reagent_volume //no val specified
    electrophoresis_buffer_volume //no val specified
)

// Data which is returned from this protocol, and data types
Data ()

// Physical Inputs to this protocol with types
Inputs (
    Sample *wtype.LHComponent
    Reagent *wtype.LHComponent
    electrophoresis_buffer *wtype.LHComponent
)
// Physical outputs from this protocol with types
Outputs (
    Mixture *wtype.LHComponent
)

Requirements {}

// Conditions to run on startup
```

```
32 Setup {}
33
34 // The core process for this protocol, with the steps to be performed
35 // for every input
36 Steps {
37
38     sample := mixer.Sample(Sample, Sample_volume)
39     reagent := mixer.Sample(Reagent, Reagent_volume)
40
41     Mixture = mixer.Mix(sample, reagent) // cannot specify duration of
           mixture
42     //cannot perform capillary electrophoresis
43     //cannot measure fluorescence to get a reading
44 }
45 Analysis {}
46 Validation {}
```

Bibliography

[1] AIHA. internet.

[2] AIHA. internet.

[3] AIHA. internet.

[4] Mirela Alistar and Urs Gaudenz. Opendrop: An integrated do-it-yourself platform for personal use of biochips. *Bioengineering*, 4(2), 2017.

[5] Mirela Alistar and Paul Pop. Synthesis of biochemical applications on digital microfluidic biochips with operation execution time variability. *Integration*, 51:158–168, 2015.

[6] 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.

[7] American Industrial Hygiene Association. `http://bit.ly/2eZtf1m`, 2016. Accessed: 2016-11-08.

[8] 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.

[9] 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. ACM, 2007.

[10] Scott C. Ananian and Arthur C. Smith. *The Static Single Information Form*. PhD thesis, Massachusetts Institue of Technology, 1999.

[11] Vaishnavi Ananthanarayanan and William Thies. Biocoder: A programming language for standardizing and automating biology protocols. *Journal of Biological Engineering*, 4, NOV 2010.

[12] Kenneth Appel and Wolfgang Haken. The solution of the four-color-map problem. *Scientific American*, 237(4):108–121, 1977.

[13] Chemical & Engineering News Archive. Letters. *Chemical & Engineering News Archive*, 76(24):4, 1998.

[14] 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.

[15] Evgenij. Barsoukov and J. Ross Macdonald. Impedence spectroscopy: Theory, experiments, and applications. *Wiley-Interscience*, 2005.

[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] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE design & Test of Computers*, pages 68–83, 2000.

[18] 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.

[19] Nirveek Bhattacharjee, Arturo Urrios, Shawn Kang, and Albert Folch. The upcoming 3d-printing revolution in microfluidics. *Lab on a Chip*, 16(10):1720–1742, 2016.

[20] Nikolaj Bjørner and Leonardo de Moura. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems,(TACAS'08)*, 2008.

[21] Swimming Pool Help Blog. Swimming pool chemical incident. `http://bit.ly/2gghGZI`, 2016. accessed: 2016-11-01.

[22] 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.

[23] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. SSI properties revisited. *ACM Trans. Embedded Comput. Syst.*, 11(S1):21, 2012.

[24] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. Ssi properties revisited. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(1):21, 2012.

[25] Max Born and Emil Wolf. *Principles of optics: electromagnetic theory of propagation, interference and diffraction of light*. Elsevier, 2013.

[26] 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.

[27] Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of ssa form programs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(5):772–779, 2006.

[28] Philip Brisk, Ajay K Verma, and Paolo Ienne. An optimal linear-time algorithm for interprocedural register allocation in high level synthesis using ssa form. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(7):1096–1109, 2010.

[29] Rowland Leonard Brooks. On colouring the nodes of a network. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 37, pages 194–197. Cambridge University Press, 1941.

[30] Arnold Cahn and Paul Hepp. Das antifebrin, ein neues fiebermittel. *Centralblatt für Klinische Medizin*, 7:561–564, 1886.

[31] CDC. Pool chemical injuries lead to over 4,500 emergency department visits each year. *Pool Chemical Injuries Lead to Over 4,500 Emergency Department Visits Each Year*, May 2019.

[32] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Massachusetts, USA, June 23-25, 1982*, pages 98–105, 1982.

[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] ChemAxon. `http://www.chemaxon.com`, 2016. Marvin was used for characterizing chemical structures, substructures and reactions, Marvin 16.10.3.

[35] 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.

[36] 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.

[37] Marek Chrobak and Thomas H Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters*, 54(4):241–246, 1995.

[38] 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.*, 20:21512156, 2005.

[39] Michael E Cournoyer. A risk determining model for hazardous material operations: Part ii. In *Probabilistic Safety Assessment and Management*, pages 1534–1540. Springer, 2004.

[40] Michael E Cournoyer and Marvin M Maestas. Addressing safety requirements through management walkarounds. *Chemical Health and Safety*, 11(6):12–16, 2004.

[41] Michael E Cournoyer, Marvin M Maestas, Donivan R Porterfield, and Patrick Spink. Chemical inventory management: The key to controlling hazardous materials. *Chemical Health and Safety*, 12(5):15–20, 2005.

[42] Brian Crites, Karen Kong, and Philip Brisk. Diagonal component expansion for flow-layer placement of flow-based microfluidic biochips. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):126, 2017.

[43] Brian Crites, Radhakrishna Sanka, Joshua Lippai, Jeffrey McDaniel, Philip Brisk, and Douglas Densmore. Parchmint: A microfluidics benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 78–79. IEEE, 2018.

[44] 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.

[45] Christopher Curtis, Daniel Grissom, and Philip Brisk. A compiler for cyber-physical digital microfluidic biochips. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 365–377. ACM, 2018.

[46] Christopher Curtis, Daniel T. Grissom, and Philip Brisk. A compiler for cyber-physical digital microfluidic biochips. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pages 365–377, 2018.

[47] 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.

[48] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[49] 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.

[50] Daniel A. Dobbs, Robert G. Bergman, and Klaus H. Theopold. Piranha solution explosion, 1990.

[51] Christopher M. Dobson. Chemical space and biology. *Nature*, 432(7019):824, 2004.

[52] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[53] Jamil El-Ali, Peter K Sorger, and Klavs F Jensen. Cells on chips. *Nature*, 442(7101):403–411, 2006.

[54] Jamil El-Ali, Peter K Sorger, and Klavs F Jensen. Cells on chips. *Nature*, 442(7101):403, 2006.

[55] Elsevier. Reaxys, 2009.

[56] Environmental Protection Agency & National Oceanic and Atmospheric Administration. `https://cameochemicals.noaa.gov/`, 2016.

[57] D. Falconnet, A. Niemistö, R. J. Taylor, M. Ricicova, T. Galitski, I. Shmulevich, and C. L. Hansen. High-throughput tracking of single yeast cells in a microfluidic imaging matrix. *Lab on a Chip*, 11(3):466–473, 2011.

[58] "F.G.". Tinting maps. *The Athenæum*, 1389:726, 1854.

[59] Luis M. Fidalgo and Sebastian J. Maerkl. A software-programmable microfluidic device for automated biology. *Lab Chip*, 11(9):1612–1619, May 2011.

[60] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[61] 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.

[62] Barbara L Foster. The chemical inventory management system in academia. *Chemical Health and Safety*, 12(5):21–25, 2005.

[63] 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.

[64] Bruce Gale, Alexander Jafek, Christopher Lambert, Brady Goenner, Hossein Moghimifam, Ugochukwu Nze, and Suraj Kamarapu. A review of current methods in microfluidic device fabrication and future commercialization prospects. *Inventions*, 3(3):60, 2018.

[65] 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 on a Chip*, 13:443–451, 2013.

[66] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[67] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[68] LM Gibbs. Chemtracker consortium–the higher education collaboration for chemical inventory management and regulatory reporting. *Chemical Health and Safety*, 12(5):9–14, 2005.

[69] 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.

[70] 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.

[71] Daniel 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. ACM, 2012.

[72] 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.

[73] Daniel Grissom, Christopher Curtis, and Philip Brisk. Interpreting assays with control flow on digital microfluidic biochips. *ACM Journal on Emerging Technologies (JETC) in Computing Systems*, 10, April 2014.

[74] Daniel Grissom, Christopher Curtis, Skyler Windh, Calvin Phung, Navin Kumar, Zachary Zimmerman, ONeal 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.

[75] Daniel Grissom, Kenneth O'Neal, Benjamin Preciado, Hiral Patel, Robert Doherty, Nick Liao, and Philip Brisk. A digital microfluidic biochip synthesis framework. In *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on*, pages 177–182. IEEE, 2012.

[76] 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.

[77] Carl L Hansen, Morten O A Sommer, and Stephen R. Quake. Systematic investigation of protein phase behavior with a microfluidic formulator. *Proceedings of the National Academy of Sciences of the United States of America*, 101(40):14431–6, 2004.

[78] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *ACM SIGPLAN Notices*, volume 33, pages 97–105. ACM, 1998.

[79] Peter Hornbeck. Enzyme-linked immunosorbent assays. *Current protocols in immunology*, pages 2–1, 1991.

[80] 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.

[81] 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.

[82] 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. IEEE Computer Society, 2009.

[83] 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.

[84] Paul J. Hung, Philip J. Lee, Poorya Sabounchi, Robert Lin, and Luke P. Lee. Continuous perfusion microfluidic cell culture array for high-throughput cell-based assays. *Biotechnology and Bioengineering*, 89(1):1–8, 2005.

[85] Paul J Hung, Philip J Lee, Poorya Sabounchi, Robert Lin, and Luke P Lee. Continuous perfusion microfluidic cell culture array for high-throughput cell-based assays. *Biotechnology and bioengineering*, 89(1):1–8, 2005.

[86] 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.

[87] Mohamed Ibrahim and Krishnendu Chakrabarty. Error recovery in digital microfluidics for personalized medicine. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 247–252, 2015.

[88] Mohamed Ibrahim, Krishnendu Chakrabarty, and Kristin Scott. Synthesis of cyber-physical digital-microfluidic biochips for real-time quantitative analysis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 36(5):733–746, 2017.

[89] Takashi Ito and Shinji Okazaki. Pushing the limits of lithography. *Nature*, 406(6799):1027, 2000.

[90] Michael J. Schertzer, Ridha Ben Mrad, and Pierre Sullivan. Automated detection of particle concentration and chemical reactions in ewod devices. *Sensors and Actuators B: Chemical*, 164:1–6, 03 2012.

[91] Christopher Jaress, Philip Brisk, and Daniel 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, 2015.

[92] 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*, 15:151158, 2015.

[93] 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.

[94] 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.

[95] Roxan Joncour, Nicolas Duguet, Estelle Métay, Amadéo Ferreira, and Marc Lemaire. Amidation of phenol derivatives: a direct synthesis of paracetamol (acetaminophen) from hydroquinone. *Green Chemistry*, 16(6):2997–3002, 2014.

[96] Oliver Keszocze, 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. IEEE, 2015.

[97] Oliver Keszocze, 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. IEEE, 2014.

[98] Faisal I Khan and Paul R Amyotte. How to make inherent safety practice a reality. *The Canadian Journal of Chemical Engineering*, 81(1):2–16, 2003.

[99] Sunghwan Kim, Paul A Thiessen, Evan E Bolton, Jie Chen, Gang Fu, Asta Gindulyte, Lianyi Han, Jane He, Siqian He, Benjamin A Shoemaker, et al. Pubchem substance and compound databases. *Nucleic acids research*, 44(D1):D1202–D1213, 2015.

[100] Eric Klavins. Aquarium, your protocols will be assimilated. `http://klavinslab.org/aquarium.html`, 2014. Accessed: 2017-11-13.

[101] Ali Lashkaripour, Christopher Rodriguez, Luis Ortiz, and Douglas Densmore. Performance tuning of microfluidic flow-focusing droplet generators. *Lab on a Chip*, 2019.

[102] Ali Lashkaripour, Ryan Silva, and Douglas Densmore. Desktop micromilled microfluidics. *Microfluidics and Nanofluidics*, 22(3):31, 2018.

[103] 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.

[104] Yiyan Li, Hongzhong Li, and R. Jacob Baker. Volume and concentration identification by using an electrowetting on dielectric device. *IEEE DCAS*, pages 1–4, 2014.

[105] 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.

[106] 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*, PP(99), 2017.

[107] 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.

[108] Gabriel Lippmann. *Relations entre les phénomènes électriques et capillaires*. Gauthier-Villars, 1875.

[109] 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. IEEE, 2013.

[110] 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.

[111] Lin Luan, Matthew White 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. *Sensors Journal, IEEE*, 12:1794–1800, June 2012.

[112] 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.

[113] Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. Error recovery in cyberphysical digital microfluidic biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(1):59–72, 2013.

[114] 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.

[115] 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.

[116] Elena Maftei, Paul Pop, and Jan Madsen. Module-based synthesis of digital microfluidic biochips with droplet-aware operation execution. *JETC*, 9(1):2, 2013.

[117] 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.

[118] Joshua S. Marcus, W. French Anderson, and Stephen R. Quake. Microfluidic single-cell mRNA isolation and analysis. *Analytical Chemistry*, 78(9):3084–3089, 2006.

[119] J McDaniel, B Crites, C Curtis, and PL Brisk. Design automation for flow-based microfluidic biochips. *Mini-Symposium: Continuous-Flow Biochips: Technology, Testing, and Design for Fault-Tolerance and Reliability*, page 1, 2018.

[120] Jeffrey McDaniel, Auralila Baez, Brian Crites, Aditya Tammewar, and Philip Brisk. Design and verification tools for continuous fluid flow-based microfluidic devices. In *18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, pages 219–224, 2013.

[121] Jeffrey McDaniel, Brian Crites, Philip Brisk, and William H. Grover. Flow-layer physical design for microchips based on monolithic membrane valves. *IEEE Design & Test*, 32(6):51–59, 2015.

[122] Jeffrey McDaniel, Brian Crites, Philip Brisk, and William H Grover. Flow-layer physical design for microchips based on monolithic membrane valves. *IEEE Design & Test*, 32(6):51–59, 2015.

[123] 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.

[124] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[125] Wajid Hassan Minhass, Jeffrey McDaniel, Michael Raagaard, Philip Brisk, Paul Pop, and Jan Madsen. Scheduling and fluid routing for flow-based microfluidic laboratories-on-a-chip. *IEEE Trans. on CAD of Integrated Circuits and Systems*, PP(99), 2017.

[126] Hyejin Moon, Sung Kwon. Cho, Robin L. Garrell, and Chang-Jin Kim. Low voltage electrowetting-on-dielectric. *J. Appl. Phys.*, 92:40804087, 2002.

[127] Frieder Mugele and Jeanchristophe Baret. Electrowetting: from basics to applications. *Journal of Physics: Condensed Matter*, 17:R705–R774, 2005.

[128] Rick Mullin. Alexa and your phone are getting schooled in chemistry. *Chemical & Engineering News*, 97, 2019.

[129] Kary B. Mullis, Henry A. Erlich, Norman Arnheim, Glenn T. Horn, Randall K. Saiki, and Stephen J. Scharf. Process for amplifying, detecting, and/or-cloning nucleic acid sequences, July 28 1987. US Patent 4,683,195.

[130] Miguel Angel Murran and Homayoun Najjaran. Capacitance-based droplet position estimator for digital microfluidic devices. *Lab Chip*, 12:2053–2059, 2012.

[131] 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.

[132] Kenneth O'Neal, Daniel 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. IEEE, 2012.

[133] 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.

[134] Jens Palsberg and Christina Pavlopoulou. From Polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, 2001.

[135] Nicole Pamme. Continuous flow separations in microfluidic devices. *Lab on a chip*, 7(12):1644–59, 2007.

[136] Nicole Pamme. Continuous flow separations in microfluidic devices. *Lab on a Chip*, 7(12):1644–1659, 2007.

[137] 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.

[138] 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.

[139] Jay Rappaport and James Lichtman. Ongoing development of a chemical/biological inventory and safety management solution for temple university. *Chemical Health and Safety*, 5(12):4–8, 2005.

[140] Hong Ren, Richard Fair, and Michael G. Pollack. Automated on-chip droplet dispensing with volume control by electro-wetting actuation and capacitance metering. *Sensors and Actuators B*, 98:319–327, 03 2004.

[141] Andrew J. Ricketts, Kevin M. Irick, Narayanan Vijaykrishnan, and Mary Jane Irwin. Priority scheduling in digital microfluidics-based biochips. In Gielen [69], pages 329–334.

[142] 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. ACM, 2010.

[143] 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.

[144] 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.

[145] 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.

[146] Radhakrishna Sanka, Haiyao Huang, Ryan Silva, and Douglas Densmore. Mint-microfluidic netlist. 2016.

[147] Jaclyn Elizabeth R Santos, Franz Nicolas N Alfonso, Fernando C Mendizabal Jr, and Fabian M Dayrit. Developing a chemical and hazardous waste inventory system. *Journal of Chemical Health and Safety*, 18(6):15–18, 2011.

[148] William G. Schulz. internet, 2005.

[149] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences , 1978.

[150] 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.

[151] 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.

[152] Yong Jun Shin and Jeong Bong Lee. Machine vision for digital microfluidics. *Review of Scientific Instruments*, 81(1), 2 2010.

[153] Sılvia Silva, Maria Luısa Lima, and Conceicao Baptista. Osci: an organisational and safety climate inventory. *Safety science*, 42(3):205–220, 2004.

[154] Jeremy Singer. *Static Program Analysis based on Virtual Register Renaming*. PhD thesis, University of Cambridge, UK, 2005.

[155] 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.

[156] Aaron Smith, Jim Burrill, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, and Kathryn S McKinley. Compiling for edge architectures. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 11–pp. IEEE, 2006.

[157] Larisa N. Soldatova, Wayne Aubrey, Ross D. King, and Amanda Clare. The exact description of biomedical protocols. *Bioinformatics*, 24, JUL 2008.

[158] 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.

[159] 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.

[160] Fei Su and Krishnendu Chakrabarty. Module placement for fault-tolerant microfluidics-based biochips. *ACM Trans. Design Autom. Electr. Syst.*, 11(3):682–710, 2006.

[161] Fei Su and Krishnendu Chakrabarty. High-level synthesis of digital microfluidic biochips. *JETC*, 3(4), 2008.

[162] Fei Su, William L. Hwang, and Krishnendu Chakrabarty. Droplet routing in the synthesis of digital microfluidic biochips. In Gielen [69], pages 323–328.

[163] Ian I. Suni. Impedance methods for electrochemical sensors using nanomaterials. *TrAC Trends in Analytical Chemistry*, 27(7):604 – 611, 2008. Electroanalysis Based on Nanomaterials.

[164] Synthace. Antha-lang, coding biology. `https://www.antha-lang.org`, 2016. accessed: 2016-11-01.

[165] Environmental of Health Texas Tech University and Safety. internet.

[166] William Thies, John Paul Urbanski, Todd Thorsen, and Saman Amarasinghe. Abstraction layers for scalable microfluidic biocomputing. *Natural Computing*, 7(2):255–275, 5 2007.

[167] Darci J Trader and Erin E Carlson. Taming of a superbase for selective phenol desilylation and natural product isolation. *The Journal of organic chemistry*, 78(14):7349–7355, 2013.

[168] Marc A. Unger, Hou Pu Chou, Todd Thorsen, Axel Scherer, and Stephen R. Quake. Monolithic microfabricated valves and pumps by multilayer soft lithography. *Science*, 288(5463):113–116, 2000.

[169] Center for Laboratory Safety University of California. internet, 2018.

[170] Division of Research Safety University of Illinois. internet.

[171] John Paul Urbanski, William Thies, Christopher Rhodes, Saman Amarasinghe, and Todd Thorsen. Digital microfluidics using soft lithography. *Lab Chip*, 6:96–104, 2006.

[172] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[173] 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.

[174] Mitchell Wand. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–43. ACM, 1986.

[175] Environmental Health Washington University in St. Louis and Safety. internet.

[176] Matthew White Royal, Nan M. Jokerst, and Richard Fair. Droplet-based sensing: Optical microresonator sensors embedded in digital electrowetting microfluidics systems. *Sensors Journal, IEEE*, 13:4733–4742, 12 2013.

[177] Max Willsey, Ashley P Stephenson, Chris Takahashi, Pranav Vaid, Bichlien H Nguyen, Michal Piszczek, Christine Betts, Sharon Newman, Sarang Joshi, Karin Strauss, et al. 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*, volume 19, 2019.

[178] Angela R. Wu, Tiara L.A. Kawahara, Nicole A. Rapicavoli, Jan van Riggelen, Emelyn H. Shroff, Liwen Xu, Dean W. Felsher, Howard Y. Chang, and Stephen R. Quake. High throughput automated chromatin immunoprecipitation as a platform for drug screening and antibody validation. *Lab on a Chip*, 12(12):2190, 2012.

[179] Y Xia and G M Whitesides. Soft Lithography. *Annual Reviews in Materials Science*, 28(1):153–184, 1998.

[180] Tao Xu and Krishnendu Chakrabarty. Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips. *JETC*, 4(3), 2008.

[181] 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.

[182] Paul Yager, Thayne Edwards, Elain Fu, Kristen Helton, Kjell Nelson, Milton R. Tam, and Bernhard H. Weigl. Microfluidic diagnostic technologies for global public health. *Nature*, 442(7101):412–418, 2006.

[183] 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.

[184] Alireza Ahmadian Yazdi, Adam Popma, William Wong, Tammy Nguyen, Yayue Pan, and Jie Xu. 3d printing: an emerging tool for novel microfluidics and lab-on-a-chip applications. *Microfluidics and Nanofluidics*, 20(3):50, 2016.

[185] 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), 2007.

[186] 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.

[187] 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.

[188] 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.