BioScript: Programming Safe Chemistry on Laboratories-on-a-Chip

JASON OTT, University of California, Riverside, USA
TYSON LOVELESS, University of California, Riverside, USA
CHRIS CURTIS, University of California, Riverside, USA
MOHSEN LESANI, University of California, Riverside, USA
PHILIP BRISK, University of California, Riverside, USA

This paper introduces *BioScript*, a domain-specific language (DSL) for programmable biochemistry which executes on emerging microfluidic platforms. The goal of this research is to provide a simple, intuitive, and type-safe DSL that is accessible to life science practitioners. The novel feature of the language is its syntax, which aims to optimize human readability; the technical contributions of the paper include the *BioScript* type system and relevant portions of its compiler. The type system ensures that certain types of errors, specific to biochemistry, do not occur, including the interaction of chemicals that may be unsafe. The compiler includes novel optimizations that place biochemical operations to execute concurrently on a spatial 2D array platform on the granularity of a control flow graph, as opposed to individual basic blocks. Results are obtained using both a cycle-accurate microfluidic simulator and a software interface to a real-world platform.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **General programming languages**; • **Theory of computation** → *Logic*; Type theory; • **Social and professional topics** → *History of programming languages*;

Additional Key Words and Phrases: Microfluidics, Laboratory-on-a-Chip (LoC), Domain-Specific Language, Type System, Compiler

ACM Reference Format:
Jason Ott, Tyson Loveless, Chris Curtis, Mohsen Lesani, and Philip Brisk. 2018. BioScript: Programming Safe Chemistry on Laboratories-on-a-Chip. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 128 (November 2018), 31 pages. https://doi.org/10.1145/3276498

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

Authors' addresses: Jason Ott, University of California, Riverside, USA, jason.ott@ucr.edu; Tyson Loveless, University of California, Riverside, USA, tlove004@ucr.edu; Chris Curtis, University of California, Riverside, USA, ccurt002@ucr.edu; Mohsen Lesani, University of California, Riverside, USA, lesani@cs.ucr.edu; Philip Brisk, University of California, Riverside, USA, philip@cs.ucr.edu.

2475-1421/2018/11-ART128
https://doi.org/10.1145/3276498

[Amin et al. 2013, 2007; Fidalgo and Maerkl 2011; Jensen et al. 2010; Pollack et al. 2002; Urbanski et al. 2006].

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 [Alistar and Gaudenz 2017; Gong and Kim 2008; Hadwen et al. 2012; Moon et al. 2002; Noh et al. 2012; Pollack et al. 2002]. 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.

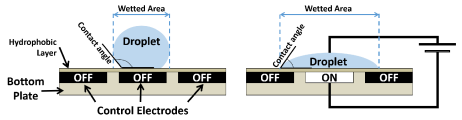
The *BioScript* language, type system, and compiler are evaluated using a set of benchmark applications obtained from scientific literature. We use a microfluidic simulator to assess performance under ideal operating conditions, and also execute them on a real device, which is much smaller and supports 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.

2 BACKGROUND & RELATED WORK

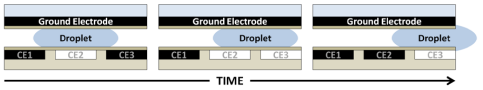
2.1 Digital Microfluidic Biochips (DMFBs).

This paper targets a specific class of programmable LoCs that manipulate discrete droplets of fluid via electrostatic actuation. Fig. 1a illustrates the electrowetting principle [Lippmann 1875; Mugele and Baret 2005]: applying an electrostatic potential to a droplet modifies the shape of the droplet and its contact angle with the surface. As shown in Fig. 1b, droplet transport can be induced by activating and deactivating a sequence of electrodes adjacent to the droplet [Pollack et al. 2002]; the ground electrode, on top of the array, improves the fidelity of droplet motion and reduces the voltage required to induce droplet transport.

Fig. 2a depicts a programmable 2D electrowetting array, called a "Digital Microfluidic Biochip (DMFB)." A DMFB can support five basic operations, shown in Fig. 2b: 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, as these operations can be performed anywhere on the array, and any given electrode can be used to perform different operations at different times.

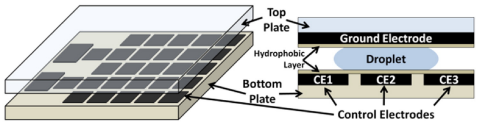


(a) The electrowetting principle [Lippmann 1875; Mugele and Baret 2005]: applying an electrostatic potential to a droplet at rest reduces the contact angle with the surface, thereby increasing the surface area in contact with the droplet

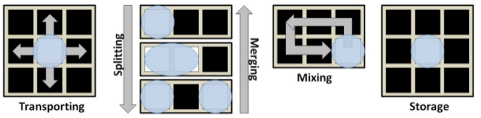


(b) A droplet is transported from control electrode CE2 to neighboring electrode CE3 by activating CE3, and then deactivating CE2 (white: activated electrode; black: deactivated electrode).

Fig. 1. The elecrowetting principle (a) enables droplet transport (b).



(a) Left: A DMFB is a planar array of electrodes [Alistar and Gaudenz 2017; Gong and Kim 2008; Hadwen et al. 2012; Moon et al. 2002; Noh et al. 2012; Pollack et al. 2002]. Right: Cross-sectional view.



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

Fig. 2. A DMFB (a) and it's reconfigurable instruction set (b).

Droplet I/O is performed using reservoirs on the perimeter of the chip, which are not depicted in Fig. 2.

The DMFB ISA can be extended by integrating sensors [Alistar and Gaudenz 2017; Barsoukov and Macdonald 2005; Bhattacharjee and Najjaran 2012; Cooreman et al. 2005; Gao et al. 2013; J. Schertzer et al. 2012; Lederer et al. 2012; Li et al. 2014, 2015; Murran and Najjaran 2012; Ren et al. 2004; Sadeghi et al. 2012; Shih et al. 2013, 2011; Suni 2008], optical detectors [Luan et al. 2008, 2012; Srinivasan et al. 2004; White Royal et al. 2013], heaters [Luo et al. 2015], or online video monitoring capabilities [Basu 2013; Fobel et al. 2013; Hu et al. 2013; Li et al. 2017; Shin and Lee 2010; Vo et al. 2017]. 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 [Alistar and Gaudenz 2017; Basu 2013; Bhattacharjee and Najjaran 2012; Fobel et al. 2013; Hu et al. 2013; Li et al. 2015, 2017; Murran and Najjaran 2012; Shih et al. 2011; Vo et al. 2017] and online error detection and recovery [Alistar and Pop 2015; Alistar et al. 2016; Hsieh et al. 2014; Ibrahim and Chakrabarty 2015a,b; Ibrahim et al. 2017; Jaress et al. 2015; Li et al. 2017; Luo et al. 2013a,b; Poddar et al. 2016; Zhao et al. 2010]; efforts to leverage these capabilities to provide control flow constructs at the language syntax level have been far more limited [Curtis and Brisk 2015; Curtis et al. 2018; Grissom et al. 2014].

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 Fig. 3, a compiler for a fluidic data

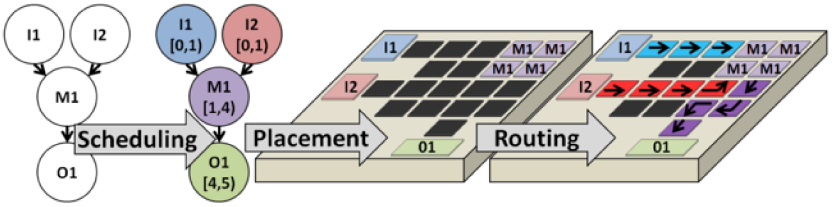


Fig. 3. A DMFB compiler for biochemical programs without control flow.

dependence graph must solve three interdependent NP-complete problems: operation scheduling [Ding et al. 2001; Grissom and Brisk 2012; Liu et al. 2013; O'Neal et al. 2012; Ricketts et al. 2006; Su and Chakrabarty 2008], reconfigurable module placement [Chen et al. 2013; Grissom and Brisk 2014; Liao and Hu 2011; Maftei et al. 2010, 2013; Su and Chakrabarty 2006; Xu and Chakrabarty 2008; Xu et al. 2008; Yuh et al. 2007], and droplet routing [Böhringer 2006; Cho and Pan 2008; Huang and Ho 2009; Keszocze et al. 2015, 2014; Roy et al. 2010, 2012; Su et al. 2006; Yuh et al. 2008].

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 [Huang et al. 2010; Yao et al. 2016; Zhao and Chakrabarty 2012].

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-time (JIT) compiling each basic block on-the-fly as the program executes [Grissom et al. 2014]: 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[Curtis et al. 2018]; 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 [Briggs et al. 1994; Chaitin 1982; Chaitin et al. 1981; George and Appel 1996] 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 [Curtis et al. 2018] 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) [Cytron et al. 1991] and Static Single Information (SSI) [Ananian and Smith 1999; Boissinot et al. 2012; Singer 2005] 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 [Sreedhar et al. 1999].

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 [Curtis et al. 2018] emphasized correctness (i.e., the ability to statically compile a CFG), but did not attempt to reduce the number or length of droplet transport operations that were inserted.

Prior work on microfluidic placement has taken inspiration from spatial computing: [Ding et al. 2001; Grissom and Brisk 2014; Xu et al. 2008] and have adapted placement algorithms originally introduced for dynamically reconfigurable FPGAs [Bazargan et al. 2000] 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 [Smith et al. 2006]. 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.

2.3 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 [Sadowski et al. 2016].

Ontologies. Ontologies in synthetic biology, such as *Synthetic Biology Open Language (SBOL)* [Galdzicki et al. 2014] or *EXACT* [Soldatova et al. 2008] aim to standardize how bio-chemical scientists discuss and disseminate information in a standardized form. They describe experiments and models in a common language, but cannot directly execute the experiment.

Laboratory Automation. *Aquarium* [Klavins 2014] 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 *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.

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 [Thies et al. 2007; Urbanski et al. 2006]. *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 [Amin et al. 2013, 2007] 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 [Ananthanarayanan and Thies 2010] and was later extended to target programmable LoCs [Curtis and Brisk 2015; Curtis et al. 2018; Grissom et al. 2014; McDaniel et al. 2013]. 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.

3 OVERVIEW

***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 § 4.

[1] http://autoprotocol.org
[2] https://docs.antha.com

Table 1. *BioScript* supported fluidic operations.

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

We divide *BioScript*'s fluidic operations into three categories, as shown in Table 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*.

Example: PCR with Droplet Replenishment. Fig. 4 presents a *BioScript* specification for a DMFB-compatible implementation of the *Polymerase Chain Reaction (PCR)*, used to amplify DNA [Mullis et al. 1987]. This specific example was obtained from the scientific literature [Jebrail et al. 2015], 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 a template solution [line 12] to ensure that replenishment does not affect the temperature of the DNA.

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}[Dobson 2004]. Cheminformatics is a field where chemists rely on computers to manage drug and compound discovery, a process whereby 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[Cahn and Hepp 1886], is a common pain medication used today, and its synthesis has been extensively studied. Reaxys[Elsevier 2009], a leading chemical reaction database, details 275 different ways to synthesize acetaminophen, but ignores factors such as safety and efficiency. As an example, Fig. 5a and 5b report *BioScript* specifications of two of the documented 275 paths: the one shown in Fig. 5a is safe, while its counterpart in Fig. 5b is unsafe

```
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
11                     with 25uL of Template for 5s
12     heat replacement at 95C for 45s
13     PCRMix = mix PCRMix with replacement for 5s
14   }
15   heat PCRMix at 68C for 30s
16   heat PCRMix at 95C for 45s
17 }
18 heat PCRMix at 68C for 5min
19 save PCRMix
```

Fig. 4. PCR with droplet replenishment [Jebrail et al. 2015]. It uses the target-specific save instruction.

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

(a) Safe assay for synthesizing acetaminophen [Joncour et al. 2014]

```
1  step_1 = mix 10uL of acetic_acid
2       with 10uL of tetrahydrofuran
3  step_2 = mix step_1 with
4               10uL of water
5  step_3 = mix step_2 with
6               10uL of acetonitrile
7  heat step_3 at 20C for 12h
```

(b) Unsafe assay synthesizing acetaminophen [Trader and Carlson 2013]. Mixing water with acetonitrile creates hydrogen cyanide, an extremely poisonous and flammable gas. Hence, this reaction may be unsafe.

Fig. 5. Example *BioScript* Examples of safe and unsafe ways to produce acetaminophen.

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.

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 [Environmental Protection Agency & National Oceanic and Atmospheric Administration 2016], 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 Fig. 4 and 5 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.

Software & Hardware Architecture. The *BioScript* compiler and runtime system is comprised of three discrete modules, as shown in Fig. 6: 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 [Ananian and Smith 1999; Boissinot et al. 2012; Singer 2005], 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 be performed either statically [Curtis et al. 2018] or dynamically [Grissom et al. 2014]. 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 [Alistar and Pop 2015; Alistar et al. 2016; Hsieh et al. 2014; Ibrahim and Chakrabarty 2015a,b; Ibrahim et al. 2017; Jaress et al. 2015; Li et al. 2017; Luo et al. 2013a,b; Poddar et al. 2016; Zhao et al. 2010]. 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.

4 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 operations 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.

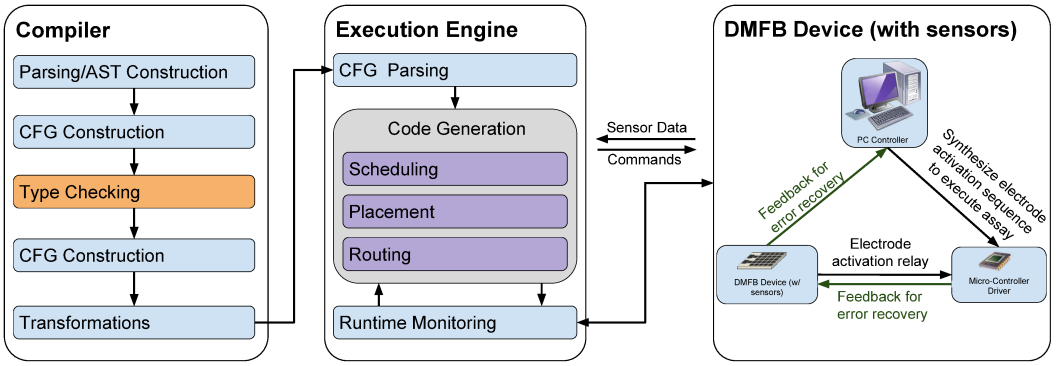


Fig. 6. The *BioScript* compiler, execution engine and DMFB device.

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.

Syntax. Fig. 7 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 "detect module on x 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.

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.

Fig. 8.(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.

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 ::=		
|	$\text{module}_1 \mid .. \mid \text{module}_n$	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
Γ ::=		Context:
|	$\emptyset$	Empty context
|	$\Gamma, x : T$	Variable type binding
X		Set of variables x
C		Constraints

Fig. 7. Syntax of *BioScript*'s type system.

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.

Fig. 8.(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

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

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

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

(a) Evaluation rules for terms

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

$$\textsc{E-MixR}\ \frac{(\sigma, t) \to t'}{(\sigma, x := \text{mix } x_1 \text{ with } x_2 \text{ for } t;\ s) \to (\sigma, x := \text{mix } x_1 \text{ with } x_2 \text{ for } t';\ s)} \qquad \textsc{E-Mix}\ \frac{\sigma(x_1) \in Mat \qquad \sigma(x_2) \in Mat \quad \mathrm{interact}(\sigma(x_1),\ \sigma(x_2),\ r) \neq \lightning \quad \sigma' = (\sigma \setminus \{x_1, x_2\})[x \mapsto \mathrm{interact}(\sigma(x_1),\ \sigma(x_2),\ r)]}{(\sigma, x := \text{mix } x_1 \text{ with } x_2 \text{ for } r;\ s) \to (\sigma',\ s)}$$

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

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

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

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

(b) Evaluation rules for statements

Fig. 8. Evaluation rules

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.

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.

Fig. 7 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 § 6.

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. Fig. 9.(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. Fig. 9.(b) lists type checking rules for statements.

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

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

(a) Typing rules for terms

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

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

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

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

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

(b) Typing rules for statements

Fig. 9. Type checking rules.

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[Palsberg and Pavlopoulou 2001]. 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.

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(Γ, X, σ), there exists σ' and s' such that $(\sigma,\ s) \rightarrow (\sigma',\ s')$.*

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').*

The proofs are available in the supplemental material.

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.

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

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

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

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

(a) Type inference rules for terms

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

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

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

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

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

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

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

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

(b) Type inference rules for statements

Fig. 10. Type inference rules

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 Fig. 10.(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: $\Gamma, X \vdash i, X' \mid C$. Fig. 10.(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 the supplemental material.

5 CODE GENERATION

Code generation begins with a typeable program, whose CFG, including fluidic variables, has been converted to SSI Form. To conserve space, we assume that the reader is familiar with SSI's ϕ- and π-functions, which respectively split variable live ranges at branch convergence and divergence points [Ananian and Smith 1999; Boissinot et al. 2012; Cytron et al. 1991; Singer 2005].

The first step is to schedule the operations in each basic block in isolation; in principle, any of the scheduling algorithms described in § 2.2 can be used.

The second step, which is a novel contribution of this work, is to perform "global" placement in a manner that is cognizant of the CFG, as opposed to prior work [Curtis et al. 2018], which limited the scope of placement to individual basic blocks. As noted in § 2.2, a fluidic dependency (u_i, u_j) represents a droplet $d_{i,j}$ that is produced by u_i and consumed by u_j, which necessitates fluid transport; however, if u_i and u_j are placed at the same location, or at least nearby, the transport operation can be eliminated or shortened. When compiling a CFG, this observation generalizes to dependencies that cross basic block boundaries.

In SSI Form, droplets that are stored across basic block boundaries are represented explicitly by ϕ- and π-functions; prior work has introduced techniques to translate out of fluidic SSI Form [Curtis et al. 2018], but did not attempt to minimize droplet transport latencies while doing so;

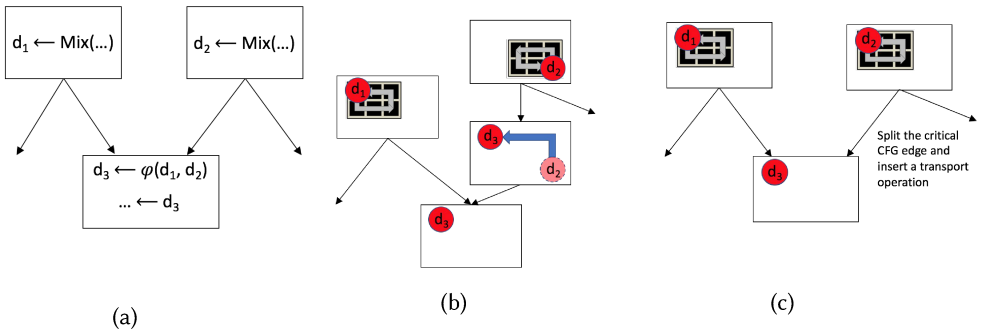


Fig. 11. An SSA/SSI program fragment (a), shown with a single ϕ-function: $d_3 \leftarrow \phi(d_1, d_2)$. An example of placement that has not been globally optimized from the perspective of the CFG (b): the operation that produces droplet d_1 is placed at the same location as the operation that consumes droplet d_3 (after renaming via the ϕ-function), eliminating the droplet transport operation if the path on the left is taken; however, the operation that produces droplet d_2 on the right is *not* placed at the same location as the operation that consumes d_3, necessitating the introduction of a new basic block (via critical edge splitting [Sreedhar et al. 1999]) that contains the transport operation. Globally optimized placement (c): the operations that produce droplets d_1 and d_2 are placed at the same location as the operation that consumes droplet d_3, thereby eliminating the need to insert additional droplet transport operations.

Fig. 11 illustrates the different results that may occur depending on whether or not the placer accounts for transport operations that occur due to ϕ- and π-functions.

The global placement problem shares many principle similarities to graph coloring register allocation [Briggs et al. 1994; Chaitin 1982; Chaitin et al. 1981; George and Appel 1996], however there are significant and subtle differences. The first is that global placement must account for both scheduled operations and fluidic variables that are stored on-chip, as both compete for space on the surface of the DMFB. The second is that operations and variables must be placed on a 2D surface, as opposed to being allocated to registers; this requires an extension to the graph coloring model, as simply assigning distinct integer "colors" to two concurrent mixing operations is insufficient to describe where they are placed. The third and final difference is that there is no off-chip fluidic memory, which means that "spilling" is not allowed; if a legal placement cannot be found, then code generation fails, and a larger DMFB is needed.

Let D be the set of fluidic variables; liveness information for each variable in D is already available as a byproduct of SSI construction. Let U be the set of assay operations, which have already been scheduled; thus, the lifetime of each operation in U, which is contained wholly within each basic block, can be derived from the schedule. Let $V = D \cup U$.

Let E be the set of interferences edges; in other words, for each pair of fluidic variables or operations $v_i, v_j \in V$, interference edge $(v_i, v_j) \in E$ exists if and only if the lifetimes of v_i and v_j overlap; in other words, v_i and v_j must be placed at different locations on the surface of the DMFB. Let $q_i = (x_i, y_i)$ denote the location at which operation or fluidic variable v_i is placed (for non-unit-size operations, such as 3×3 mixers, the location can be defined anywhere, e.g., upper-left corner, center, etc., as long as the definition is applied consistently). Two operations (and/or stored variables) are placed "legally" if they do not overlap and there is at least one row of unused electrodes between them. For a more precise definition, refer to Refs. [Curtis et al. 2018; Su and Chakrabarty 2006]; details are omitted to conserve space.

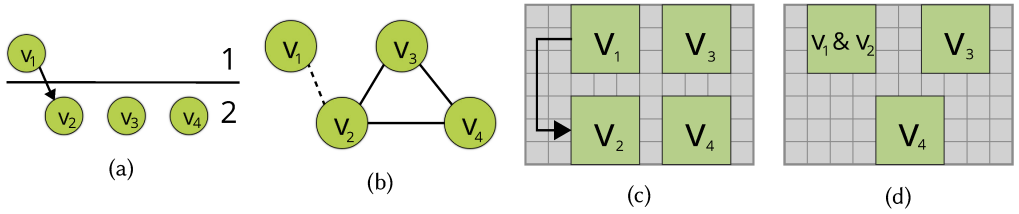


Fig. 12. A scheduled basic block (a); the resulting interference graph corresponding to the schedule, with one affinity edge (b); an unoptimized placement (c): since there is an affinity edge between v_1 an v_2, placing them at different locations necessitates droplet transport; and an optimized placement (d): since there is an affinity edge between v_1 an v_2, placing them at the same location eliminates the need to transport a droplet.

Let A be the set of affinity edges; conceptually, for each pair of non-interfering fluidic variables or operations $v_i, v_k \in V$, an affinity edge $(v_i, v_k) \in A$ exists if it is possible to eliminate a droplet transport operation by placing v_i and v_k are placed at the same location. This occurs in three situations: operation v_i produces a fluid that is used by operation v_k (or vice-versa) as discussed earlier; there exists a ϕ-function $d_{j,k} \leftarrow \phi_j(\ldots, d_{i,j}, \ldots)$; or there exists a π-function $(\ldots, d_{j,k}, \ldots) \leftarrow \pi_j(d_{i,j})$.

Given a placement solution, the droplet $d_{i,j}$ associated with affinity edge $(v_i, v_j) \in A$ must travel a distance $D(v_i, v_j)$, which depends on q_i and q_j. The exact distance that $d_{i,j}$ will travel is not known until after routing, as there may be additional placed operations and variables that obstruct the shortest possible paths between q_i and q_j. Under the assumption of rectilinear droplet transport, the distance $D'(v_i, v_j) = |x_i - x_j| + |y_i - y_j|$ is a lower bound; and if $q_i = q_j$, then $D(v_i, v_j) = D'(v_i, v_j) = 0$ is exact.

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

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

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

As an example, Fig. 12a shows a scheduled basic block. Fig. 12b shows the resulting interference graph, which features three interference edges and one affinity edge (v_1, v_2). Fig. 12c shows a legal, but unoptimized placement, the places v_1 and v_2 at different locations, thereby incurring the overhead of droplet transport. Fig. 12d shows a legal and optimized placement, where v_1 and v_2 are placed at the same location, thereby eliminating the need to transport the droplet.

Device Execution. To solve the global placement problem in practice, we used NSGA-II [Deb et al. 2002], a publicly available genetic algorithm[3]. As an iterative improvement heuristic, NSGA-II produces locally optimal solutions, although the running time and overall solution quality depend on a number of user-specified parameters. An initial feasible solution is obtained using an efficient heuristic [Grissom and Brisk 2014]. The solution is encoded as binary variables of the form (x_i, y_i, o_i, s_i), where o_i and s_i are the orientation and size of operation or stored variable v_i; the number of bits required for x_i and y_i vary, based on the maximum dimension of the architecture.

We configured NSGA-II to use a population size of 100 and to run for 250 iterations. The initial feasible solution is implanted in the initial population, and additional solutions are generated via mutation and crossover operations. After each generation, encoded solutions that NSGA-II

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

discovers via evolution are extracted and examined for legality; the objective is computed for legal solutions. The constraints and objectives are returned to NSGA-II for the subsequent evolution. We maintain a copy of the best solution(s) found thus far. After the final generation, we return the best legal placement solution that was discovered.

Once global placement has been solved, the final step is to translate from SSI form [Curtis et al. 2018], and to insert droplet routes as needed; any of the droplet routing methods described in § 2.2 can be used.

6 IMPLEMENTATION

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

BioScript. The *BioScript* language was implemented as described in § 3. 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. 14d. 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.

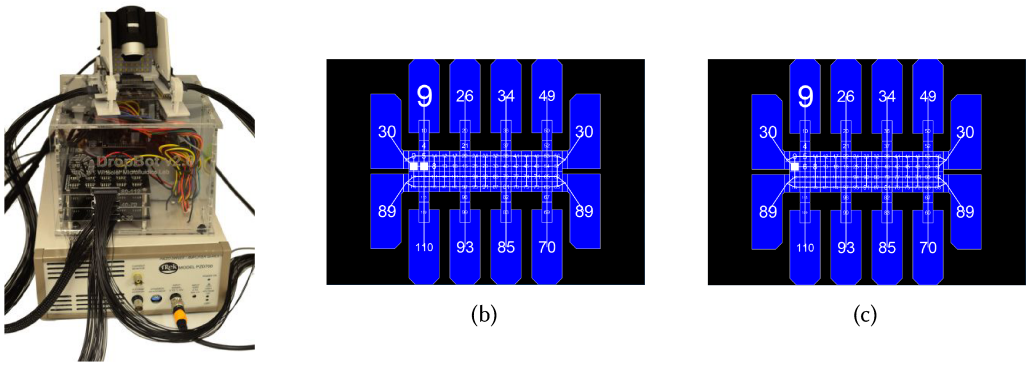
The Type System. *BioScript*'s type system utilizes static type checking, which runs during compilation. The type system automatically infers types using an abstract interaction function that is a conservative over-approximation of the resulting chemical types of each interaction. The type system uses 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 § 4) 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 § 4). Otherwise, if the mix operation yields a new chemical c_k, we use the industry-standard *ChemAxon* [ChemAxon 2016] 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 [4]. 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 [Milner 1978]. Constraints are gathered from the CFG according to the type inference rules. Constraints are encoded in the SMTLib2 format and passed to Z3 [De Moura and Bjørner 2008] for satisfiability. In cases that no type can be inferred, deciding which part of the program is to blame is a classical problem[Wand 1986]. Life scientists using *BioScript* would benefit from localized typing errors; including the necessary heuristics is left for future work.

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

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



(a) (b) (c)

Fig. 13. DropBot device (a) and example execution, where (b) & (c) depict mixing two droplets.

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.

Code Generation. The code generator presently targets three DMFB back-ends. The first two are simulators that can statically compile [Curtis et al. 2018] or dynamically interpret [Grissom et al. 2014] 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 [Fobel et al. 2013], shown in Fig. 13a. 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. 13 shows DropBot's graphical interface while manipulating droplets on a real-world device.

7 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*.

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 [Amin et al. 2007]; *Antha*, a language for cloud-based laboratory automation [Synthace 2016]; and BioCoder, a C++ library that has been previously specialized for DMFBs [Ananthanarayanan and Thies 2010; Curtis and Brisk 2015; Grissom et al. 2014]. We review these three languages in greater detail in § 2. Our comparison uses a set of compact, yet representative, bioassays taken from published literature. As an illustrative example, Fig. 14 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. 14a) 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. 14b), is imperative, but involves unintuitive notation such as []* and . operators (e.g., []*wtype.LHComponent).
- The *AIS* specification (Fig. 14c) 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. 14a) requires 3 lines of code. The specification is compact and declares variables implicitly.[5]

Fig. 15 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 [Amin et al. 2007]) 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. 14d, 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, [Curtis and Brisk 2015; Grissom et al. 2014], unlike *AIS* and *Antha*. Although these results do not account for subjective experience, we believe that they convey the same basic sentiments as Fig. 14: *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 Fig. 15 are included in our supplementary materials.

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 [American Industrial Hygiene Association 2016; Blog 2016], 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 2 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)*

[5]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.

```
/*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)

Fig. 14. Example assay specified using Biocoder(Fig. 14a)[Curtis and Brisk 2015; Grissom and Brisk 2014], Antha(Fig. 14b)[Synthace 2016], AIS(Fig. 14c)[Amin et al. 2007], and *BioScript*(Fig. 14d).

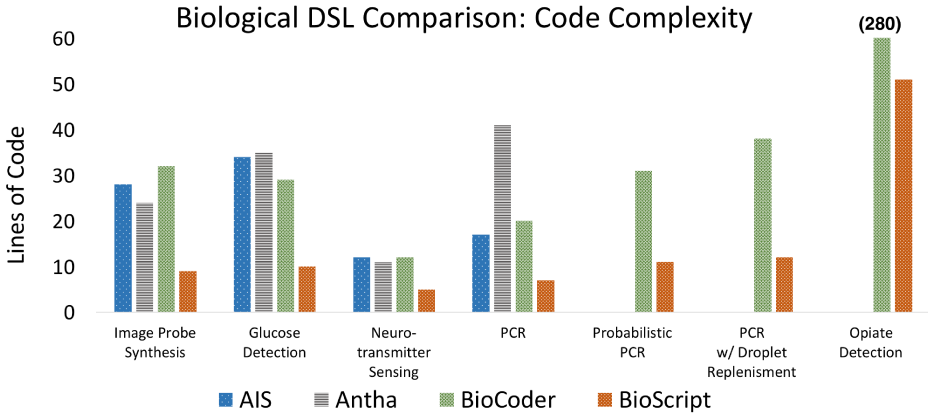


Fig. 15. The number of lines of code to specify Image Probe Synthesis, Glucose Detection, Neurotransmitter Sensing, PCR[Amin et al. 2007], Probabilistic PCR[Luo et al. 2015], PCR w/ Droplet Replacement[Jebrail et al. 2015], and Opiate Detection[Backer et al. 2005; Jiang et al. 2011; Mao et al. 2006] in AIS [Amin et al. 2007], BioCoder [Ananthanarayanan and Thies 2010; Curtis and Brisk 2015; Grissom et al. 2014], Antha [Synthace 2016], and *BioScript*. We were unable to specify the latter three assays in AIS and Antha.

[American Industrial Hygiene Association 2016]. *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 [Dobbs et al. 1990] (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.

Table 2. 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.

Experiment Name	Outcome	Experiment Description
AIHA 1 [American Industrial Hygiene Association 2016]	FAIL - I	Mix Nitric Acid (2) and Tetrachloroethylene (17,28)
AIHA 2 [American Industrial Hygiene Association 2016]	FAIL - I	Mix Nitric Acid (2) and Methanol (4)
AIHA 3 [American Industrial Hygiene Association 2016]	FAIL - I	Mix Potassium Hydride (35, 21) and Diaminopropane (7)
Mustard Gas [Blog 2016]	FAIL - I	Mix Calcium Hypo (1) and Dichlor (17)
Safety Zone [Dobbs et al. 1990]	FAIL - I	Mix Hydrogen Peroxide (44) and Sulfuric Acid (2), then mix Acetone (19)

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, 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.

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™.

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 [ChemAxon 2016]. Constructing the *abstraction interaction table* using a multithreaded implementation of *ChemAxon* would significantly reduce construction time.

Table 3 reports the compilation time, constraint solving time, number of constraints gather, 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.

BioScript assays, along with several additional synthetic benchmarks, are made available in the supplemental materials.

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 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 [Curtis et al. 2018]. 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

Table 3. Compile time, the number of constraints gathered, and simulated execution times for the safe and unsafe assays.

Benchmark	Compilation Time (sec)	Constraint Solving Time (sec)	Gathered Constraints	Execution Engine Time (m:s:ms)	Execution Time
AIHA 1 [American Industrial Hygiene Association 2016]	0.012	0.936	70	N/A	N/A
AIHA 2 [American Industrial Hygiene Association 2016]	0.012	1.648	68	N/A	N/A
AIHA 3 [American Industrial Hygiene Association 2016]	0.014	1.214	17	N/A	N/A
Broad Spectrum Opiate [Backer et al. 2005; Jiang et al. 2011; Mao et al. 2006]	0.011	0.887	11	0:18:55	0:23:21
Ciprofloxacin [Jiang et al. 2011]	0.023	1.722	14	101:31:80	128:54:32
Diazepam [Hornbeck 1991]	0.024	1.007	14	96:48:13	121:01:39
Dilution [Hornbeck 1991]	0.014	0.892	9	0:21:05	0:26:33
Fentanyl [Mao et al. 2006]	0.018	0.900	13	126:32:40	158:10:80
Full Morphine [Hornbeck 1991]	0.048	4.188	19	127:16:78	159:06:17
Glucose Detection [Amin et al. 2007]	0.012	1.633	14	0:23:77	0:29:73
Heroine [Hornbeck 1991]	0.020	1.553	13	126:32:40	158:10:80
Image Probe Synthesis [Amin et al. 2007]	0.015	2.181	13	8:38:96	10:47:50
Morphine [Hornbeck 1991]	0.018	1.026	13	126:32:40	158:10:80
Mustard Gas [Blog 2016]	0.015	1.433	83	N/A	N/A
Oxycodone [Backer et al. 2005]	0.026	0.959	13	126:32:40	158:10:80
PCR [Amin et al. 2007]	0.032	3.534	8	11:16:12	14:36:29
Safety Zone [Dobbs et al. 1990]	0.013	1.341	76	N/A	N/A
Cancer-detection via gene-editing [Sinha et al. 2018]	0.016	1.637	16	1920:08:01	N/A

Table 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 [Curtis et al. 2018].

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

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.

8 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.

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 [Kim et al. 2015] 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.

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.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1740052, Grant No. 1640757, Grant No. 1545907, Grant No. 1536026, and Grant No. 1351115. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Mirela Alistar and Urs Gaudenz. 2017. OpenDrop: An Integrated Do-It-Yourself Platform for Personal Use of Biochips. *Bioengineering* 4, 2 (2017). https://doi.org/10.3390/bioengineering4020045

Mirela Alistar and Paul Pop. 2015. Synthesis of biochemical applications on digital microfluidic biochips with operation execution time variability. *Integration* 51 (2015), 158–168. https://doi.org/10.1016/j.vlsi.2015.02.004

Mirela Alistar, Paul Pop, and Jan Madsen. 2016. Synthesis of Application-Specific Fault-Tolerant Digital Microfluidic Biochip Architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems* 35, 5 (2016), 764–777. https://doi.org/10.1109/TCAD.2016.2528498

American Industrial Hygiene Association. 2016. http://bit.ly/2eZtf1m. Accessed: 2016-11-08.

Ahmed M. Amin, Raviraj Thakur, Seth Madren, Han-Sheng Chuang, Mithuna Thottethodi, T. N. Vijaykumar, Steven T. Wereley, and Stephen C. Jacobson. 2013. Software-programmable continuous-flow multi-purpose lab-on-a-chip. *Microfluid Nanofluidics* 15, 5 (Nov 2013), 647–659.

Ahmed M. Amin, Mithuna Thottethodi, T. N. Vijaykumar, Steven Wereley, and Stephen C. Jacobson. 2007. Aquacore: a programmable architecture for microfluidics. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, Dean M. Tullsen and Brad Calder (Eds.). ACM, 254–265. https://doi.org/10.

1145/1250662.1250694

Scott C. Ananian and Arthur C. Smith. 1999. *The Static Single Information Form.* Ph.D. Dissertation. Massachusetts Institue of Technology.

Vaishnavi Ananthanarayanan and William Thies. 2010. Biocoder: A programming language for standardizing and automating biology protocols. *Journal of Biological Engineering* 4 (NOV 2010).

Ronald C Backer, Joseph R Monforte, and Alphonse Poklis. 2005. Evaluation of the DRI® oxycodone immunoassay for the detection of oxycodone in urine. *Journal of analytical toxicology* 29, 7 (2005), 675–677.

Evgenij. Barsoukov and J. Ross Macdonald. 2005. Impedence Spectroscopy: Theory, Experiments, and Applications. *Wiley-Interscience* (2005).

Amar S. Basu. 2013. Droplet morphometry and velocimetry (DMV): a video processing software for time-resolved, label-free tracking of droplet parameters. *Lab Chip* 13 (2013), 1892–1901. Issue 10. https://doi.org/10.1039/C3LC50074H

Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. 2000. Fast template placement for reconfigurable computing systems. *IEEE design & Test of Computers* 1 (2000), 68–83.

Biddut Bhattacharjee and Homayoun Najjaran. 2012. Droplet sensing by measuring the capacitance between coplanar electrodes in a digital microfluidic system. *Lab Chip* 12 (2012), 4416–4423. Issue 21. https://doi.org/10.1039/C2LC40647K

Swimming Pool Help Blog. 2016. Swimming pool chemical incident. http://bit.ly/2gghGZI. accessed: 2016-11-01.

Karl-Friedrich Böhringer. 2006. Modeling and Controlling Parallel Tasks in Droplet-Based Microfluidic Systems. *IEEE Trans. on CAD of Integrated Circuits and Systems* 25, 2 (2006), 334–344. https://doi.org/10.1109/TCAD.2005.855958

Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. 2012. SSI Properties Revisited. *ACM Trans. Embedded Comput. Syst.* 11, S1 (2012), 21. https://doi.org/10.1145/2180887.2180898

Preston Briggs, Keith D Cooper, and Linda Torczon. 1994. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 428–455.

Arnold Cahn and Paul Hepp. 1886. Das antifebrin, ein neues fiebermittel. *Centralblatt für Klinische Medizin* 7 (1886), 561–564.

Gregory J. Chaitin. 1982. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Massachusetts, USA, June 23-25, 1982.* 98–105. https://doi.org/10.1145/800230.806984

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation Via Coloring. *Comput. Lang.* 6, 1 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

ChemAxon. 2016. http://www.chemaxon.com. Marvin was used for characterizing chemical structures, substructures and reactions, Marvin 16.10.3.

Ying-Han Chen, Chung-Lun Hsu, Li-Chen Tsai, Tsung-Wei Huang, and Tsung-Yi Ho. 2013. 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 (2013), 1151–1162. https://doi.org/10.1109/TCAD.2013.2249558

Minsik Cho and David Z. Pan. 2008. A High-Performance Droplet Routing Algorithm for Digital Microfluidic Biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems* 27, 10 (2008), 1714–1724. https://doi.org/10.1109/TCAD.2008.2003282

Peter Cooreman, Ronald Thoelen, Jean Manca, M. vandeVen, V. Vermeeren, L. Michiels, M. Ameloot, and P. Wagner. 2005. Impedimetric immunosensors based on the conjugated polymer PPV. *Biosens. Bioelectron.* 20 (2005), 2151–2156. Issue 10.

Christopher Curtis and Philip Brisk. 2015. Simulation of feedback-driven PCR assays on a 2D electrowetting array using a domain-specific high-level biological programming language. *Microelectronic Engineering* 148 (2015), 110–116.

Christopher Curtis, Daniel Grissom, and Philip Brisk. 2018. A compiler for cyber-physical digital microfluidic biochips. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization.* ACM, 365–377.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (Apr 2002), 182–197. https://doi.org/10.1109/4235.996017

Jie Ding, Krishnendu Chakrabarty, and Richard B. Fair. 2001. Scheduling of microfluidic operations for reconfigurabletwo-dimensional electrowetting arrays. *IEEE Trans. on CAD of Integrated Circuits and Systems* 20, 12 (2001), 1463–1468. https://doi.org/10.1109/43.969439

Daniel A. Dobbs, Robert G. Bergman, and Klaus H. Theopold. 1990. Piranha solution explosion. , 2–2 pages.

Christopher M. Dobson. 2004. Chemical space and biology. *Nature* 432, 7019 (2004), 824.

Elsevier. 2009. Reaxys. https://new.reaxys.com/

Environmental Protection Agency & National Oceanic and Atmospheric Administration. 2016. https://cameochemicals.noaa.gov/. https://cameochemicals.noaa.gov/

Luis M. Fidalgo and Sebastian J. Maerkl. 2011. A software-programmable microfluidic device for automated biology. *Lab Chip* 11, 9 (May 2011), 1612–1619.

Ryan Fobel, Christian Fobel, and Aaron R. Wheeler. 2013. 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, Article 193513 (2013). https://doi.org/10.1063/1.4807118

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. 2014. The Synthetic Biology Open Language (SBOL) provides a community standard for communicating designs in synthetic biology. *Nature biotechnology* 32, 6 (2014), 545–550.

Jie Gao, Xianming Liu, Tianlan Chen, Pui-In Mak, Yuguang Du, Mang-I Vai, Bingcheng Lin, and Rui P. Martins. 2013. An intelligent digital microfluidic system with fuzzy-enhanced feedback for multi-droplet manipulation. *Lab Chip* 13 (2013), 443–451. Issue 3. https://doi.org/10.1039/C2LC41156C

Lal George and Andrew W. Appel. 1996. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.* 18, 3 (1996), 300–324. https://doi.org/10.1145/229542.229546

Georges G. E. Gielen (Ed.). 2006. *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. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=11014

Jian Gong and Chang-Jin Kim. 2008. Direct-referencing two-dimensional-array digital microfluidics using multilayer printed circuit board. *J. Microelectromech. Syst.* 17 (2008), 257–264. Issue 2.

Daniel Grissom and Philip Brisk. 2012. Path scheduling on digital microfluidic biochips. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun (Eds.). ACM, 26–35. https://doi.org/10.1145/2228360.2228367

Daniel Grissom and Philip Brisk. 2014. Fast Online Synthesis of Digital Microfluidic Biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems* 33, 3 (2014), 356–369. https://doi.org/10.1109/TCAD.2013.2290582

Daniel Grissom, Christopher Curtis, and Philip Brisk. 2014. Interpreting Assays with Control Flow on Digital Microfluidic Biochips. *ACM Journal on Emerging Technologies (JETC) in Computing Systems* 10 (April 2014). Issue 3.

Ben. Hadwen, G. R. Broder, D. Morganti, A. Jacobs, C. Brown, J. R. Hector, Y. Kubota, and H. Morgan. 2012. Programmable large area digital microfluidic array with integrated droplet sensing for bioassays. *Lab Chip* 12, 18 (Sep 2012), 3305–3313.

Peter Hornbeck. 1991. Enzyme-linked immunosorbent assays. *Current protocols in immunology* (1991), 2–1.

Yi-Ling Hsieh, Tsung-Yi Ho, and Krishnendu Chakrabarty. 2014. Biochip Synthesis and Dynamic Error Recovery for Sample Preparation Using Digital Microfluidics. *IEEE Trans. on CAD of Integrated Circuits and Systems* 33, 2 (2014), 183–196. https://doi.org/10.1109/TCAD.2013.2284010

Kai Hu, Bang-Ning Hsu, Andrew Madison, Krishnendu Chakrabarty, and Richard B. Fair. 2013. Fault detection, real-time error recovery, and experimental demonstration for digital microfluidic biochips. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, Enrico Macii (Ed.). EDA Consortium San Jose, CA, USA / ACM DL, 559–564. https://doi.org/10.7873/DATE.2013.124

Tsung-Wei Huang and Tsung-Yi Ho. 2009. 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*. IEEE Computer Society, 445–450. https://doi.org/10.1109/ICCD.2009.5413119

Tsung-Wei Huang, Chun-Hsien Lin, and Tsung-Yi Ho. 2010. A Contamination Aware Droplet Routing Algorithm for the Synthesis of Digital Microfluidic Biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems* 29, 11 (2010), 1682–1695. https://doi.org/10.1109/TCAD.2010.2062770

Mohamed Ibrahim and Krishnendu Chakrabarty. 2015a. Efficient Error Recovery in Cyberphysical Digital-Microfluidic Biochips. *IEEE Trans. Multi-Scale Computing Systems* 1, 1 (2015), 46–58. https://doi.org/10.1109/TMSCS.2015.2478457

Mohamed Ibrahim and Krishnendu Chakrabarty. 2015b. 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*. 247–252. http://dl.acm.org/citation.cfm?id=2755807

Mohamed Ibrahim, Krishnendu Chakrabarty, and Kristin Scott. 2017. Synthesis of Cyberphysical Digital-Microfluidic Biochips for Real-Time Quantitative Analysis. *IEEE Trans. on CAD of Integrated Circuits and Systems* 36, 5 (2017), 733–746. https://doi.org/10.1109/TCAD.2016.2600626

Michael J. Schertzer, Ridha Ben Mrad, and Pierre Sullivan. 2012. Automated detection of particle concentration and chemical reactions in EWOD devices. 164 (03 2012), 1–6.

Christopher Jaress, Philip Brisk, and Daniel Grissom. 2015. 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*. IEEE Computer Society, 1–6. https://doi.org/10.1109/VTS.2015.7116246

Mais J. Jebrail, Ronald F. Renzi, Anupama Sinha, Jim Van De Vreugde, Carmen Gondhalekar, Cesar Ambriz, Robert J. Meagher, and Steven S. Branda. 2015. A solvent replenishment solution for managing evaporation of biochemical reactions in

air-matrix digital microfluidics devices. *Lab Chip* 15 (2015), 151–158.

Erik C. Jensen, Bharath P. Bhat, and Richard A. Mathies. 2010. A digital microfluidic platform for the automation of quantitative biomolecular assays. *Lab Chip* 10, 6 (Mar 2010), 685–691.

Yousheng Jiang, Xuanyun Huang, Kun Hu, Wenjuan Yu, Xianle Yang, and Liqun Lv. 2011. Production and characterization of monoclonal antibodies against small hapten-ciprofloxacin. *African Journal of Biotechnology* 10, 65 (2011), 14342–14347.

Roxan Joncour, Nicolas Duguet, Estelle Métay, Amadéo Ferreira, and Marc Lemaire. 2014. Amidation of phenol derivatives: a direct synthesis of paracetamol (acetaminophen) from hydroquinone. *Green Chemistry* 16, 6 (2014), 2997–3002.

Oliver Keszocze, Robert Wille, Krishnendu Chakrabarty, and Rolf Drechsler. 2015. A General and Exact Routing Methodology for Digital Microfluidic Biochips. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015*, Diana Marculescu and Frank Liu (Eds.). IEEE, 874–881. https://doi.org/10.1109/ICCAD.2015.7372663

Oliver Keszocze, Robert Wille, and Rolf Drechsler. 2014. Exact routing for digital microfluidic biochips with temporary blockages. In *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*, Yao-Wen Chang (Ed.). IEEE, 405–410. https://doi.org/10.1109/ICCAD.2014.7001383

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. 2015. PubChem substance and compound databases. *Nucleic acids research* 44, D1 (2015), D1202–D1213.

Eric Klavins. 2014. Aquarium, Your protocols will be assimilated. http://klavinslab.org/aquarium.html. Accessed: 2017-11-13.

Thomas Lederer, Stefan Clara, Bernhard Jakoby, and Wolfgang Hilber. 2012. Integration of impedance spectroscopy sensors in a digital microfluidic platform. *Microsystem Technologies* 18, 7 (01 Aug 2012), 1163–1180. https://doi.org/10.1007/s00542-012-1464-6

Yiyan Li, Hongzhong Li, and R. Jacob Baker. 2014. Volume and concentration identification by using an electrowetting on dielectric device. *IEEE DCAS* (2014), 1–4.

Yiyan Li, Hongzhong Li, and R. Jacob Baker. 2015. A Low-Cost and High-Resolution Droplet Position Detector for an Intelligent Electrowetting on Dielectric Device. *Journal of Laboratory Automation* 20, 6 (2015), 663–669. https://doi.org/10.1177/2211068214566940 arXiv:https://doi.org/10.1177/2211068214566940 PMID: 25609255.

Zipeng Li, Kelvin Yi-Tse Lai, John McCrone, Po-Hsien Yu, Krishnendu Chakrabarty, Miroslav Pajic, Tsung-Yi Ho, and Chen-Yi Lee. 2017. 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). https://doi.org/10.1109/TCAD.2017.2729347

Chen Liao and Shiyan Hu. 2011. Multiscale variation-aware techniques for high-performance digital microfluidic lab-on-a-chip component placement. *IEEE Trans Nanobioscience* 10, 1 (Mar 2011), 51–58.

Gabriel Lippmann. 1875. *Relations entre les phénomènes électriques et capillaires.* Gauthier-Villars. https://books.google.ch/books?id=IkqVNwAACAAJ

Chia-Hung Liu, Kuang-Cheng Liu, and Juinn-Dar Huang. 2013. Latency-optimization synthesis with module selection for digital microfluidic biochips. In *2013 IEEE International SOC Conference, Erlangen, Germany, September 4-6, 2013*, Norbert Schuhmann, Kaijian Shi, and Nagi Naganathan (Eds.). IEEE, 159–164. https://doi.org/10.1109/SOCC.2013.6749681

L. Luan, R.D. Evans, N.M. Jokerst, and R.B. Fair. 2008. Integrated optical sensor in a digital microfluidic platform. *IEEE Sensors* 8 (2008), 628–635. Issue 5.

Lin Luan, Matthew White Royal, Randall Evans, Richard B. Fair, and Nan M. Jokerst. 2012. Chip Scale Optical Microresonator Sensors Integrated With Embedded Thin Film Photodetectors on Electrowetting Digital Microfluidics Platforms. 12 (June 2012), 1794–1800. Issue 6.

Yan Luo, Bhargab B. Bhattacharya, Tsung-Yi Ho, and Krishnendu Chakrabarty. 2015. 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 (2015), 29–42. https://doi.org/10.1109/TCAD.2014.2363396

Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. 2013a. Error Recovery in Cyberphysical Digital Microfluidic Biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems* 32, 1 (2013), 59–72. https://doi.org/10.1109/TCAD.2012.2211104

Yan Luo, Krishnendu Chakrabarty, and Tsung-Yi Ho. 2013b. Real-Time Error Recovery in Cyberphysical Digital-Microfluidic Biochips Using a Compact Dictionary. *IEEE Trans. on CAD of Integrated Circuits and Systems* 32, 12 (2013), 1839–1852. https://doi.org/10.1109/TCAD.2013.2277980

Elena Maftei, Paul Pop, and Jan Madsen. 2010. Tabu search-based synthesis of digital microfluidic biochips with dynamically reconfigurable non-rectangular devices. *Design Autom. for Emb. Sys.* 14, 3 (2010), 287–307. https://doi.org/10.1007/s10617-010-9059-x

Elena Maftei, Paul Pop, and Jan Madsen. 2013. Module-Based Synthesis of Digital Microfluidic Biochips with Droplet-Aware Operation Execution. *JETC* 9, 1 (2013), 2. https://doi.org/10.1145/2422094.2422096

Chi-Liang Mao, Keith D Zientek, Patrick T Colahan, Mei-Yueh Kuo, Chi-Ho Liu, Kuo-Ming Lee, and Chi-Chung Chou. 2006. 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 (2006), 1332–1341.

Jeffrey McDaniel, Christopher Curtis, and Philip Brisk. 2013. Automatic synthesis of microfluidic large scale integration chips from a domain-specific language. *2013 IEEE Biomedical Circuits and Systems Conference, BioCAS 2013* 1 (2013), 101–104. https://doi.org/10.1109/BioCAS.2013.6679649

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Hyejin Moon, Sung Kwon. Cho, Robin L. Garrell, and Chang-Jin Kim. 2002. Low voltage electrowetting-on-dielectric. *J. Appl. Phys.* 92 (2002), 4080–4087. Issue 7.

Frieder Mugele and Jeanchristophe Baret. 2005. Electrowetting: from basics to applications. *Journal of Physics: Condensed Matter* 17 (2005), R705–R774.

Kary B. Mullis, Henry A. Erlich, Norman Arnheim, Glenn T. Horn, Randall K. Saiki, and Stephen J. Scharf. 1987. Process for amplifying, detecting, and/or-cloning nucleic acid sequences. https://www.google.com/patents/US4683195 US Patent 4,683,195.

Miguel Angel Murran and Homayoun Najjaran. 2012. Capacitance-based droplet position estimator for digital microfluidic devices. *Lab Chip* 12 (2012), 2053–2059. Issue 11. https://doi.org/10.1039/C2LC21241B

Joo Hyon Noh, Jiyong Noh, Eric Kreit, Jason Heikenfeld, and Philip D. Rack. 2012. Toward active-matrix lab-on-a-chip: programmable electrofluidic control enabled by arrayed oxide thin film transistors. *Lab Chip* 12, 2 (Jan 2012), 353–360.

Kenneth O'Neal, Daniel Grissom, and Philip Brisk. 2012. Force-Directed List Scheduling for Digital Microfluidic Biochips. In *20th IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC 2012, Santa Cruz, CA, USA, October 7-10, 2012*, Srinivas Katkoori, Matthew R. Guthaus, Ayse Kivilcim Coskun, Andreas Burg, and Ricardo Reis (Eds.). IEEE, 7–11. https://doi.org/10.1109/VLSI-SoC.2012.6378997

Jens Palsberg and Christina Pavlopoulou. 2001. From Polyvariant flow information to intersection and union types. *Journal of Functional Programming* 11, 3 (2001), 263–317. https://doi.org/10.1017/S095679680100394X

Sudip Poddar, Sarmishtha Ghoshal, Krishnendu Chakrabarty, and Bhargab B. Bhattacharya. 2016. Error-Correcting Sample Preparation with Cyberphysical Digital Microfluidic Lab-on-Chip. *ACM Trans. Design Autom. Electr. Syst.* 22, 1 (2016), 2:1–2:29. https://doi.org/10.1145/2898999

Michael G. Pollack, Alexander D. Shenderov, and Richard B. Fair. 2002. Electrowetting-based actuation of droplets for integrated microfluidics. *Lab on a Chip* 2, 2 (2002), 96–101.

Hong Ren, Richard Fair, and Michael G. Pollack. 2004. Automated on-chip droplet dispensing with volume control by electro-wetting actuation and capacitance metering. 98 (03 2004), 319–327.

Andrew J. Ricketts, Kevin M. Irick, Narayanan Vijaykrishnan, and Mary Jane Irwin. 2006. Priority scheduling in digital microfluidics-based biochips, See [Gielen 2006], 329–334. https://doi.org/10.1109/DATE.2006.244178

Pranab Roy, Hafizur Rahaman, and Parthasarathi Dasgupta. 2010. A novel droplet routing algorithm for digital microfluidic biochips. In *Proceedings of the 20th ACM Great Lakes Symposium on VLSI 2009, Providence, Rhode Island, USA, May 16-18 2010*, R. Iris Bahar, Fabrizio Lombardi, David Atienza, and Erik Brunvand (Eds.). ACM, 441–446. https://doi.org/10.1145/1785481.1785583

Pranab Roy, Hafizur Rahaman, and Parthasarathi Dasgupta. 2012. Two-level clustering-based techniques for intelligent droplet routing in digital microfluidic biochips. *Integration* 45, 3 (2012), 316–330. https://doi.org/10.1016/j.vlsi.2011.11.006

Saman Sadeghi, Huijiang Ding, Gaurav J. Shah, Supin Chen, Pei Yuin Keng, Chang-Jin "CJ" Kim, and R. Michael van Dam. 2012. On Chip Droplet Characterization: A Practical, High-Sensitivity Measurement of Droplet Impedance in Digital Microfluidics. *Analytical Chemistry* 84, 4 (2012), 1915–1923. https://doi.org/10.1021/ac202715f arXiv:http://dx.doi.org/10.1021/ac202715f PMID: 22248060.

Michael I. Sadowski, Chris Grant, and Tim S. Fell. 2016. Harnessing QbD, Programming Languages, and Automation for Reproducible Biology. *Trends in Biotechnology* 34, 3 (2016), 214 – 227. https://doi.org/10.1016/j.tibtech.2015.11.006 Special Issue: Industrial Biotechnology.

Steve C. Shih, Irena Barbulovic-Nad, Xuning Yang, Ryan Fobel, and Aaron R. Wheeler. 2013. Digital microfluidics with impedance sensing for integrated cell culture and analysis. *Biosens Bioelectron* 42 (Apr 2013), 314–320.

Steve C. Shih, Ryan Fobel, Paresh Kumar, and Aaron R. Wheeler. 2011. A feedback control system for high-fidelity digital microfluidics. *Lab Chip* 11 (2011), 535–540. Issue 3. https://doi.org/10.1039/C0LC00223B

Yong Jun Shin and Jeong Bong Lee. 2010. Machine vision for digital microfluidics. *Review of Scientific Instruments* 81, 1 (2 2010). https://doi.org/10.1063/1.3274673

Jeremy Singer. 2005. *Static Program Analysis based on Virtual Register Renaming*. Ph.D. Dissertation. University of Cambridge, UK.

Hugo Sinha, Angela B. V. Quach, Philippe Q. N. Vo, and Steve C. Shih. 2018. An automated microfluidic gene-editing platform for deciphering cancer genes. *Lab Chip* (2018), 11–12. https://doi.org/10.1039/C8LC00470F

Aaron Smith, Jim Burrill, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, and Kathryn S McKinley. 2006. Compiling for EDGE architectures. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*. IEEE, 11–pp.

Larisa N. Soldatova, Wayne Aubrey, Ross D. King, and Amanda Clare. 2008. The EXACT description of biomedical protocols. *Bioinformatics* 24 (JUL 2008).

Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. 1999. Translating Out of Static Single Assignment Form. In *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*. 194–210. https://doi.org/10.1007/3-540-48294-6_13

Vijay Srinivasan, Vamsee Pamula, and Richard Fair. 2004. Droplet-based microfluidic lab-on-a-chip for glucose detection. 507 (04 2004), 145–150.

Fei Su and Krishnendu Chakrabarty. 2006. Module placement for fault-tolerant microfluidics-based biochips. *ACM Trans. Design Autom. Electr. Syst.* 11, 3 (2006), 682–710. https://doi.org/10.1145/1142980.1142987

Fei Su and Krishnendu Chakrabarty. 2008. High-level synthesis of digital microfluidic biochips. *JETC* 3, 4 (2008). https://doi.org/10.1145/1324177.1324178

Fei Su, William L. Hwang, and Krishnendu Chakrabarty. 2006. Droplet routing in the synthesis of digital microfluidic biochips, See [Gielen 2006], 323–328. https://doi.org/10.1109/DATE.2006.244177

Ian I. Suni. 2008. Impedance methods for electrochemical sensors using nanomaterials. *TrAC Trends in Analytical Chemistry* 27, 7 (2008), 604 – 611. https://doi.org/10.1016/j.trac.2008.03.012 Electroanalysis Based on Nanomaterials.

Synthace. 2016. Antha-lang, Coding Biology. https://www.antha-lang.org. accessed: 2016-11-01.

William Thies, John Paul Urbanski, Todd Thorsen, and Saman Amarasinghe. 2007. Abstraction layers for scalable microfluidic biocomputing. *Natural Computing* 7, 2 (5 2007), 255–275.

Darci J Trader and Erin E Carlson. 2013. Taming of a superbase for selective phenol desilylation and natural product isolation. *The Journal of organic chemistry* 78, 14 (2013), 7349–7355.

John Paul Urbanski, William Thies, Christopher Rhodes, Saman Amarasinghe, and Todd Thorsen. 2006. Digital microfluidics using soft lithography. *Lab Chip* 6 (2006), 96–104. Issue 1. https://doi.org/10.1039/B510127A

Philippe Q. N. Vo, Mathieu C. Husser, Fatemeh Ahmadi, Hugo Sinha, and Steve C. Shih. 2017. Image-based feedback and analysis system for digital microfluidics. *Lab Chip* 17 (2017), 3437–3446. Issue 20. https://doi.org/10.1039/C7LC00826K

Mitchell Wand. 1986. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 38–43.

Matthew White Royal, Nan M. Jokerst, and Richard Fair. 2013. Droplet-Based Sensing: Optical Microresonator Sensors Embedded in Digital Electrowetting Microfluidics Systems. 13 (12 2013), 4733–4742.

Tao Xu and Krishnendu Chakrabarty. 2008. Integrated droplet routing and defect tolerance in the synthesis of digital microfluidic biochips. *JETC* 4, 3 (2008). https://doi.org/10.1145/1389089.1389091

Tao Xu, Krishnendu Chakrabarty, and Fei Su. 2008. Defect-Aware High-Level Synthesis and Module Placement for Microfluidic Biochips. *IEEE Trans. Biomed. Circuits and Systems* 2, 1 (2008), 50–62. https://doi.org/10.1109/TBCAS.2008.918283

Hailong Yao, Qin Wang, Yiren Shen, Tsung-Yi Ho, and Yici Cai. 2016. 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 (2016), 1283–1296. https://doi.org/10.1109/TCAD.2015.2504397

Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. 2007. Placement of defect-tolerant digital microfluidic biochips using the T-tree formulation. *JETC* 3, 3 (2007). https://doi.org/10.1145/1295231.1295234

Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. 2008. 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 (2008), 1928–1941. https://doi.org/10.1109/TCAD.2008.2006140

Yang Zhao and Krishnendu Chakrabarty. 2012. Cross-Contamination Avoidance for Droplet Routing in Digital Microfluidic Biochips. *IEEE Trans. on CAD of Integrated Circuits and Systems* 31, 6 (2012), 817–830. https://doi.org/10.1109/TCAD.2012.2183369

Yang Zhao, Tao Xu, and Krishnendu Chakrabarty. 2010. Integrated control-path design and error recovery in the synthesis of digital microfluidic lab-on-chip. *JETC* 6, 3 (2010), 11:1–11:28. https://doi.org/10.1145/1777401.1777404