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:

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 nanoliter 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...
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 use). 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 electrowetting principle (a) enables droplet transport (b).


(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
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 \(\phi\)- and \(\pi\)-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 \(\phi\)-function \(\phi_j\), which we denote using subscript-\(j\) to ensure notational consistency: \(u_i\) produces a droplet \(d_{i,j}\), which is read by \(\phi_j\), and \(\phi_j\) produces a droplet \(d_{j,k}\), which is read by \(u_k\); the exact statement of the \(\phi\)-function is therefore \(d_{j,k} \leftarrow \phi(\ldots, d_{i,j}, \ldots)\). SSA elimination replaces \(\phi_j\) 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 of 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.

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

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

Table 1. BioScript supported fluidic operations.

<table>
<thead>
<tr>
<th>Target</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Material Declaration</td>
</tr>
<tr>
<td></td>
<td>Mix</td>
</tr>
<tr>
<td></td>
<td>Output</td>
</tr>
<tr>
<td></td>
<td>Store</td>
</tr>
<tr>
<td></td>
<td>Repeat</td>
</tr>
<tr>
<td>Control Flow</td>
<td>Branch</td>
</tr>
<tr>
<td></td>
<td>Loop</td>
</tr>
<tr>
<td>Digital</td>
<td>Detect</td>
</tr>
<tr>
<td></td>
<td>Heat</td>
</tr>
<tr>
<td></td>
<td>Split</td>
</tr>
</tbody>
</table>

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

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

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

(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.
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 maybe
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.
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, \( \text{module}_1, \ldots, \text{module}_n \), each of which measures a property of a material. A detect term \( \text{detect module on } x \text{ for } t \) returns the property that \( \text{module} \) detects for the material represented by the variable \( x \) after a measurement for \( t \) time units. A value \( v \) is a material value \( \text{mat} \), a real number \( r \) or natural number \( n \). We use \( \mathcal{Mat} \), \( \mathcal{N} \) and \( \mathcal{R} \) 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 \( \text{skip} \) 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 \( "x := \text{mix } x_1 \text{ with } x_2 \text{ for } t" \) 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 \( x_1 \text{’s } \oplus \text{ group with that of } x_2 \). The split instruction \( "\langle x_1, \ldots, x_n \rangle := \text{split } x \text{ into } n" \) splits the material represented by the variable \( x \) into \( n \) parts and assigns them to variables \( x_0, \ldots, x_n \). The conditional and loop instructions are standard.

**Operational Semantics.** We model execution of BioScript assays on a DMFB as an operational semantics. A store \( \sigma \) maps the set of variables to a set of values. The state of the transition system is a pair \((\sigma, s)\), 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-\text{VAR} checks that the store \( \sigma \) holds the value of a variable; if so, the variable’s value is read from the store.

Rules E-MATHR1 and E-MATHR2 evaluate the first and second operands of a math operation in-order; then the rule E-MATH applies the operation to the arguments if both are numeric values.
The rule E-DetectR evaluates the time term \( t \) until it is reduced to a value. Then, the rule E-Detect reduces a detect term if the value of the variable in the store is a material. Property measurement by modules is modeled as the function detect that given the material, the module and the measurement time returns the property value. If the value of the variable in the store is not a material, the detect term is stuck.

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 \( \frac{1}{r} \) 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 \( x_2 \)
The then or else statement depending on the value of the condition. The operator \( \text{split} \) reduces the condition term; rules \( \text{E-IfR} \) and mapping the output variables to the splits. The function \( \text{split} \) already used and removed from the store, or the interaction of the two is unsafe.

\[ \text{E-Split} \]

\[ (\sigma, \langle x_1, \ldots, x_n \rangle) \Rightarrow \text{split } x \text{ into } n \] \[ \sigma \Rightarrow (\sigma \setminus \{x\})[x \mapsto \text{split}(\sigma(x), n)] \]

\[ \text{E-While} \]

\[ (\sigma, \text{if } n > 0 \text{ then } s_1 \text{ else } s_2; \ s) \Rightarrow (\sigma, \ s_1 \bullet s) \]

\[ \text{E-While} \]

\[ (\sigma, \text{if } 0 \text{ then } s_1 \text{ else } s_2; \ s) \Rightarrow (\sigma, \ s_2 \bullet s) \]

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 \( \text{E-Split} \) reduces a material mix 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 \( \text{mat} \) and the number of splits \( n \), it split \( \text{mat} \) to \( n \) parts of equal volume and returns a part. Rule \( \text{E-IfR} \) reduces the condition term; rules \( \text{E-IfTrue} \) and \( \text{E-IfFalse} \) 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 $\cup S$ 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 $Mat_1, \ldots, Mat_n$ 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 $Mat_1 \cup \ldots \cup Mat_n$. 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 $\overline{\mathbb{N}}$. 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 $\Gamma$ 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, $\Gamma$ 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 $\Gamma$ 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 $\Gamma$ 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 $\Gamma$ 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.
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.

Fig. 9. Type checking rules.
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 a set of 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 \( \sigma \) and the static type environment \( \Gamma \) and the set of variables \( X \). A store \( \sigma \) is consistent with the type environment \( \Gamma \) and the set of variables \( X \), if every variable that is in \( X \) and \( \Gamma \), has a value in \( \sigma \) whose type complies with \( \Gamma \).

**Definition 1.** For all \( \Gamma, X, \sigma, \) 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 \( \Gamma, X, s \) and \( X' \), if \( \Gamma, X \vdash s, X' \) then either \( s \) is skip or for all \( \sigma \) such that consistent\((\Gamma, X, \sigma) \), there exists \( \sigma' \) 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 \( \sigma \) steps to a statement \( s' \) and a store \( \sigma' \), then \( s' \) is typed and \( \sigma' \) is consistent as well.

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

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 \( \Gamma \) and available variables \( X \), the term \( t \) is typed as \( T \) if the constraints \( C \) are satisfied.
The constraints $C$ are quantifier-free set theory formulae. Since we have a finite set of scalar types, the constraints can be reduced to quantifier-free formula in the theory of equality. The type inference rules for terms are presented in 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 $\Gamma, x : T | C$, which states that under typing environment $\Gamma$ 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: $\Gamma, x : T | C$. 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; \( \Gamma_0 \) 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, \( \Gamma_0 \) 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 \( [x \mapsto m(V_x)] \) 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 \( \Gamma \) 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 \( \Gamma \) yields the concrete type environment \( m(\Gamma) \) under which the type checking system can type-check \( s \).

**Lemma 3 (Soundness).** For all \( \Gamma, X, s, C, X' \), 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 \( \Gamma \) 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 \( \Gamma \), derives the constraints \( C \) for \( s \), then \( C \) is satisfiable and \( m \) is a model for it.

**Lemma 4 (Completeness).** For all \( \Gamma, X, s, X', X'', C, \) 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 \( \phi \)- and \( \pi \)-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 \( \phi \)- and \( \pi \)-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 $\phi$-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 $\phi$-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 $\phi$- and $\pi$-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 $3 \times 3$ 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.
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 $\phi$-function $d_{j,k} \leftarrow \phi_j(\ldots, d_{i,j}, \ldots)$; or there exists a $\pi$-function $(\ldots, d_{j,k}, \ldots) \leftarrow \pi_i(d_{i,j})$.

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. 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
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 \( \text{Mat}_i \), that together with natural \( \mathbb{N} \), and real \( \mathbb{R} \) numbers, constitute the set of scalar types \( S \).

We calculate the abstract interaction function \( \text{interact-abs} \) (used in § 4) as a table that is indexed by two material types and stores union types. Each reactive group or type \( \text{Mat}_i \) comprises a non-empty set of chemicals \( C_i \). Abstract mixing of a pair of material types \( \text{Mat}_i \) and \( \text{Mat}_j \) 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 \((\text{Mat}_i, \text{Mat}_j)\) 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 \( \bigcup \text{Mat}_k \) to \( c_k \), which are added to the union type of the cell for \( \text{Mat}_i \) and \( \text{Mat}_j \). 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. To account for this fact, \( \text{Mat}_i \) and \( \text{Mat}_j \) 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.
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.

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)
/* 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();

/* 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)
rcn := MixInto(OutPlate, "", smpl...)
r1 := Incubate(rcn, mlTemp, InitDenatime, false)

/* Initialization Omitted */

input s1, ip1
input s2, ip2
move mixer1, s1;
move mixer1, s2;
mix mixer1, 10;
move heater1, mixer1;
incubate heater1, 100, 10;

/* Initialization Omitted */

mixture = mix 10uL of water with
10 uL of blood for 10s
heat mixture at 100C for 10s

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 [Ananthanarayan 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.

<table>
<thead>
<tr>
<th>Experiment Name</th>
<th>Outcome</th>
<th>Experiment Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIHA 1 [American Industrial Hygiene Association 2016]</td>
<td>FAIL - I</td>
<td>Mix Nitric Acid (2) and Tetrachloroethylene (17,28)</td>
</tr>
<tr>
<td>AIHA 2 [American Industrial Hygiene Association 2016]</td>
<td>FAIL - I</td>
<td>Mix Nitric Acid (2) and Methanol (4)</td>
</tr>
<tr>
<td>AIHA 3 [American Industrial Hygiene Association 2016]</td>
<td>FAIL - I</td>
<td>Mix Potassium Hydride (35, 21) and Diaminopropane (7)</td>
</tr>
<tr>
<td>Mustard Gas [Blog 2016]</td>
<td>FAIL - I</td>
<td>Mix Calcium Hypo (1) and Dichlor (17)</td>
</tr>
<tr>
<td>Safety Zone [Dobbs et al. 1998]</td>
<td>FAIL - I</td>
<td>Mix Hydrogen Peroxide (44) and Sulfuric Acid (2), then mix Acetone (19)</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Compilation Time (sec)</th>
<th>Constraint Solving Time (sec)</th>
<th>Gathered Constraints</th>
<th>Execution Engine Time (m:s:ms)</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIHA 1 [American Industrial Hygiene Association 2016]</td>
<td>0.012</td>
<td>0.936</td>
<td>70</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AIHA 2 [American Industrial Hygiene Association 2016]</td>
<td>0.012</td>
<td>1.648</td>
<td>68</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AIHA 3 [American Industrial Hygiene Association 2016]</td>
<td>0.014</td>
<td>1.214</td>
<td>17</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Broad Spectrum Opiate [Backer et al. 2005; Jiang et al. 2011; Mao et al. 2006]</td>
<td>0.011</td>
<td>0.887</td>
<td>11</td>
<td>0:18:55</td>
<td>0:23:21</td>
</tr>
<tr>
<td>Diazepam [Hornbeck 1991]</td>
<td>0.024</td>
<td>1.007</td>
<td>14</td>
<td>96:48:13</td>
<td>121:01:39</td>
</tr>
<tr>
<td>Dilution [Hornbeck 1997]</td>
<td>0.014</td>
<td>0.852</td>
<td>9</td>
<td>0:21:95</td>
<td>0:28:33</td>
</tr>
<tr>
<td>Fentanyl [Mao et al. 2006]</td>
<td>0.018</td>
<td>0.900</td>
<td>13</td>
<td>126:32:40</td>
<td>158:10:80</td>
</tr>
<tr>
<td>Full Morphine [Hornbeck 1991]</td>
<td>0.048</td>
<td>4.158</td>
<td>19</td>
<td>127:16:78</td>
<td>159:06:17</td>
</tr>
<tr>
<td>Glucose Detection [Amin et al. 2007]</td>
<td>0.012</td>
<td>1.633</td>
<td>14</td>
<td>0:23:77</td>
<td>0:29:73</td>
</tr>
<tr>
<td>Heroine [Hornbeck 1991]</td>
<td>0.020</td>
<td>1.553</td>
<td>13</td>
<td>126:32:40</td>
<td>158:10:80</td>
</tr>
<tr>
<td>Image Frie Synthesis [Amin et al. 2007]</td>
<td>0.015</td>
<td>2.181</td>
<td>13</td>
<td>8:38:96</td>
<td>10:47:30</td>
</tr>
<tr>
<td>Morphine [Hornbeck 1991]</td>
<td>0.018</td>
<td>1.026</td>
<td>13</td>
<td>126:32:40</td>
<td>158:10:80</td>
</tr>
<tr>
<td>Mustard Gas [Blog 2016]</td>
<td>0.015</td>
<td>1.433</td>
<td>83</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Oxycodone [Backer et al. 2005]</td>
<td>0.026</td>
<td>0.959</td>
<td>13</td>
<td>126:32:40</td>
<td>158:10:80</td>
</tr>
<tr>
<td>PCR [Amin et al. 2007]</td>
<td>0.032</td>
<td>3.534</td>
<td>8</td>
<td>11:16:12</td>
<td>14:36:29</td>
</tr>
<tr>
<td>Safety Zone [Dobbs et al. 1980]</td>
<td>0.013</td>
<td>1.341</td>
<td>76</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Cancer-detection via gene-editing [Sinha et al. 2018]</td>
<td>0.016</td>
<td>1.637</td>
<td>16</td>
<td>1920:08:01</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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].

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

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**


Mais J. Jehrail, 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...


