Extensible Semantics for Lab Automation

Max Willsey and Jared Roesch
Paul G. Allen School for Computer Science and Engineering
University of Washington

Lab automation technology automatically manipulates chemical or biological samples at smaller scales than ever before, saving time and reagents. Scientists are now writing programs instead of informal, textbook-style protocols.

These programs mix computation with fluidic manipulation, introducing the difficulties of traditional programming *plus* liquid handling problems and domain-specific complexities.

A successful programming solution must manage these complexities and leverage scientists' domain knowledge. Importantly, the system must be accessible by users who are not programming language experts (or even programmers!).

We believe the programming language community is well-suited to address these challenges.

We picture a two-fold solution: a core fluidic semantics to manage the complexities of liquid handling, and an extensible layer that ensures programs respect user-provided domain-specific properties. This talk will present how some of the problems in this area line up with—and sometimes challenge—well understood PL techniques.

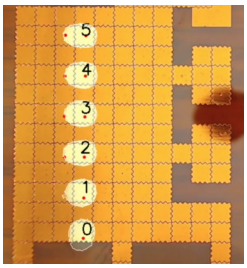
Programming Lab Automation

Lab automation is coming, with or without the input of programming language experts.

Droplet-based microfluidic (DMF) technology is especially promising because of its flexibility. DMF devices manipulate individual droplets of liquids on a grid of electrodes (Figure 1). Activating electrodes in certain patterns can move, mix, or split droplets anywhere on the chip. These DMF devices are akin to general-purpose CPUs when contrasted with other liquid handling technologies which are fixed function hardware, similar to GPUs or other accelerators. Unfortunately, they suffer from little programming abstraction and high failure rates [2].

Programming these devices is similar to other forms of heterogeneous programming in that it involves controlling and communicating with a specialized device. Figure 2 shows a short pseudocode snippet of a fluidic program. On top of conventional programming constructs, we have primitives like `mix` and `heat` that manipulate fluids. Embedded in a general purpose language, users can combine these primitives into complex procedures that mix computation and fluidic manipulation like `get_pH` and `acidify` in the example. `get_pH` also demonstrates how programs can depend on properties of the physical world like the pH of a sample.

Fluidic manipulation poses interesting compilation challenges related to placement and routing that we will not cover due to space. Instead, we will focus on two key questions raised by the combination of programming, liquid handling, and chemistry:




```
l3 = mix(l1, l2)
while get_pH(l3) > 7:
    heat(l3)
    acidify(l3)
    ...
# l4 = mix(l1, l3)
# error!
```

Figure 1: Our prototype DMF chip with droplet tracking.
Figure 2: A simple fluidic program. The commented-out line would fail because `l1` has already been consumed.

- Can we help the programmer deal with complexities and high failure rate of liquid handling?
- How can we reason about programs (like the loop in Figure 2) that rely on physical processes from chemistry and biology?

Core Fluidic Semantics

The first step toward an effective fluidic programming model is a core semantics. This semantics should capture notions inherent to liquid handling but generic across specific domains of chemistry or biology.

Fluids have physical properties such as volume or location on a DMF device. Liquid handling primitives like `mix` and `split` affect these properties and add complexity to the program. Programs that manipulate physical entities are fundamentally resource-oriented: you only have so much of a sample. Furthermore, these manipulations are error prone on DMF devices: a split may result in loss of precision, for example sample volumes that are slightly off, or failure to split altogether.

We will discuss a few programming language techniques we believe to be applicable in this setting to aid with resource management and error tolerance.

Substructual typing In programming languages, we use linear or affine logics to enforce *exactly once* or *at-most-once* use of certain variables. When a variable is *consumed*, the type system prevents the program from using it again. These kinds of semantics are a perfect fit for liquid handling, where operations like `mix` *physically*

consume their inputs. Such a type system could prevent errors like the one in Figure 2.

A more fine-grained notion of substructural typing that enriches variables with a notion of quantity could be useful. For example, a `split` operation on a DMF chip may not work if the sample is too small.

Garbage collection Garbage collection has made programming safer and more accessible by automating memory management. A similar technique would be useful in liquid handling, as many reactions generate waste byproducts that take up space and must be disposed.

A runtime system could determine which samples the programmer is no longer using and dispose of them. Some samples may be too volatile for automatically disposal. Linear semantics could require the user to invoke a destructor which safely disposes of the resource.

Dynamic analysis Many of the operations on a DMF device are error-prone, so prior work has used sensors and cameras to detect these errors at runtime [4]. Because of their frequency, treating these errors as user handled exceptions is a non-option; the execution must incorporate automatic error correction.

Simple errors can be fixed by retrying: if a split yields the wrong volume ratio, the runtime can mix them together and retry. More complex errors will require program analysis to correct. If a sample gets ruined (perhaps by an accidental collision with another sample), a dynamic analysis like slicing could figure out which parts of the program must be re-run to regenerate the sample.

Probabilistic Complications The ideas proposed above may solve some of the problems with fluidic programming, but we cannot just apply them off-the-shelf because of the domain's idiosyncrasies.

The error-prone nature of the hardware necessitates some kind of automatic error correction. The addition of a retry mechanism should improve reliability, but it complicates resource management. For example, sub-structural typing can no longer can promise safe resource usage if the runtime system can re-run any part of the program.

Further complicating things, parts of the program involving chemical reactions may have time constraints. For example, a heated sample should be used before it cools. But again, the system cannot guarantee that a program meets time constraints if the runtime can do retries.

In order to address these challenges we need to define probabilistic error models of the hardware and adapt these techniques to the stochastic environment. Recent work has recognized the need for a stochastic approach [1], but a system that provides high-level, safe liquid handling is still a ways out.

Domain-specific Extensions

Researchers have spent decades understanding how chemical and biological systems operate. Scientists rely on this vast body of knowledge when developing protocols. Ideally, the programming system would understand the relevant biochemical properties of the protocol and ensure that the user respects them. But attempting to encode a significant fraction of biology or chemistry into the core formal semantics is impossible. Instead, users should be able to *extend* the semantics with relevant biochemical properties.

Static assertions Many domain-specific properties could be phrased as safety properties: never let substance X mix with substance Y, or make sure I use at most 20mL of this solution. Such properties are easy to assert dynamically, but crashing the program at runtime is expensive when the experiment takes hours to run and consumes reagents.

Techniques from verification, testing, symbolic execution or abstract interpretation can be used to establish static guarantees but there are both scaling and precision problems applying these in large systems. Luckily, many protocols are simple in terms of computation and control flow. If the user can provide the right abstraction over the chemical or biological properties at play, these techniques could statically ensure many properties. This kind of assurance will be necessary as fluidic devices make their way into safety-critical fields like medicine.

Protocol checking Session types [3] describe communication protocols and enforce that programs adhere to the correct structure. We can view fluidic programs as protocols that communicate with the natural world. The correct session type would ensure that the user takes certain actions in the correct order.

For example, a synthetic biologist may want to ensure DNA amplification *always* occurs before sequencing it. The user could add domain knowledge to the system by writing down a session type describing the series of events that must occur. The system can then ensure regardless of what computation occurs, that the program respects the fluidic protocol.

Conclusion

Mixing conventional programming and liquid handling poses a rich set of challenges to the PL community. In the talk, we will give examples of these kinds of problems and discuss how PL techniques might be suited to solve them. We will also briefly present our prototype fluidic programming system.

[1] Alessandro Abate et al. *Experimental Biological Protocols with Formal Semantics*. 2017. eprint: `arXiv:1710.08016`.

[2] Kihwan Choi et al. "Digital Microfluidics". In: *Annual Review of Analytical Chemistry* 5.1 (2012), pp. 413–440. DOI: `10.1146/annurev-anchem-062011-143028`.

[3] Kohei Honda. "Types for dyadic interaction". In: *CONCUR'93*. 1993, pp. 509–523. ISBN: 978-3-540-47968-0. DOI: `10.1007/3-540-57208-2_35`.

[4] Philippe Q. N. Vo et al. "Image-based feedback and analysis system for digital microfluidics". In: (2017). DOI: `10.1039/C7LC00826K`.