Modern data management systems are increasingly distributed across large collections of machines, both to store and process the rapidly-increasing data volumes now produced by even modest-sized enterprises and to satisfy the growing hunger of analysts for "big data." Distributed systems are difficult to program and reason about because of fundamental uncertainty about their executions and outcomes. Due to asynchronous communication, nondeterminism in the ordering and timing of messages delivery can "leak" into program outcomes, leading to data inconsistencies. Due to partial failure—a failure mode unique to distributed systems—some components and communication attempts may fail during execution, resulting in incomplete or corrupt outcomes. Traditional solutions (e.g., distributed transactions) that hide these complexities from the programmer are considered by many to be inviable at scale, and are increasingly replaced with ad-hoc solutions that trade correctness guarantees for acceptable and predictable performance. The challenges of programming distributed systems are exacerbated by the fact that they are no longer the sole domain of experts. The relatively recent accessibility of large-scale computing resources (e.g., the public cloud), and proliferation of reusable data management components (e.g., “NoSQL” stores, data processing frameworks, caches and message queues) have created a crisis: all programmers must learn to be distributed programmers. Few tools exist to assist application programmers, data analysts and mobile developers to struggle with these tradeoffs.

My research focuses on developing new languages, analyses and tools to simplify the task of implementing and reasoning about large-scale distributed data-management systems. This agenda draws from a variety of disciplines, including distributed systems, databases, formal logic, programming languages and software engineering. It has required collaboration with experts in these areas, as well as outreach both external (engaging with practitioners who build and maintain large-scale distributed systems) and internal (teaching Berkeley undergraduates distributed systems concepts using my languages and tools as vehicles). I am strongly influenced by the data-centric philosophy espoused by many data management systems, focusing on data (diversely represented as distributed state, messages and events) as a first-class concern and on computation as a generalization of queries.

I believe that quality database research requires agility and perspective: my cross-disciplinary approach reflects this breadth of disciplines and influences. I have found that a useful and repeatable pattern for impactful systems research involves 1.) observing the common practice of domain experts programming large-scale systems, 2.) studying (in a simplified model such as a small language) the design patterns and best practices of the domain, 3.) generalizing these patterns into formal theories, and finally 4.) converting these theories into practical analyses and tools that can provide strong guarantees about system behavior and ultimately improve the state of the art. Concretely, I found that my system design work (e.g. building and studying BOOM Analytics) deeply informed the programming language design process (DEDALUS and BLOOM); theoretical results that emerged from the study of these languages drove the design of programmer tools such as BLAZES and LDFI, which will facilitate the implementation of a new generation of systems—a virtuous circle.

In the remainder of this statement I present my thesis work in the context of two distinct but deeply interdependent themes: disorderly programming and confluence analysis.

Disorderly Programming

Many of the challenges of programming distributed systems arise from the mismatch between the sequential model of computation adopted by most programming languages and the inherently disorderly nature of distributed
systems, in which no total order of events exists. Nondeterminism in message delivery order can cause distributed programs to produce nondeterministic results, which complicates testing and debugging. For programs that replicate state, this nondeterminism can lead to replica divergence and other consistency anomalies. Ensuring that sequential programs—which are specified as an ordered list of operations to perform on an ordered array of memory cells—are executed in lock-step in a distributed execution can incur unacceptable performance penalties.

At the other end of the spectrum, purely “declarative” languages like SQL—which shift the programmer’s focus from computation to data—are set-based and lack the ability to even express ordered operations. In exchange for this limited expressivity, statements in such languages can be safely evaluated in a data-parallel, coordination-free manner.

Disorderly programming—a key theme in my thesis work—extends the declarative programming paradigm with a minimal set of ordering constructs. Instead of overspecifying order and then exploring ways to relax it (e.g., by using threads and synchronization primitives), a disorderly programming language encourages programmers to underspecify order, to make it easy (and natural) to express safe and scalable computations. When required, ordering constructs can be expressed and inserted at an appropriately coarse grain to achieve the needs of core tasks like mutable state and distributed coordination. This programming style necessarily requires powerful analysis capabilities and tool support, to ensure that programs which require additional ordering constraints to produce correct results are recognized and repaired.

**BOOM: Data-centric system design**

The BOOM (Berkeley Order Of Magnitude) project explored the hypothesis that using declarative languages and data-centric programming techniques could dramatically simplify the implementation, maintenance and evolution of large-scale distributed systems. To validate this hypothesis, my colleagues and I used a distributed logic programming language based on Datalog to implement BOOM Analytics, an API-compliant Hadoop/HDFS clone [10], which performed competitively with the reference implementation despite having been written in orders of magnitude less code. We then extended this simple core with features previously unavailable in Hadoop/HDFS, including a Paxos-backed replicated master [11] for high availability, a partitioned namespace for scalability and state-of-the-art tracing and monitoring facilities. BOOM Analytics served as a proof of concept for the state-as-data, systems-as-queries approach to programming distributed systems.

**Dedalus: A formal disorderly language**

Despite our successes applying a declarative networking language to protocol and application implementation, the early generation of our distributed logic languages were fraught with semantic difficulties. While ideally suited to expressing relationships among data elements (and hence, according to our hypothesis, the majority of interesting distributed computation), query languages cannot unambiguously express relationships between states in time. Such relationships (e.g., atomicity, sequentiality, mutual exclusion and state mutation) are required to unambiguously specify or implement critical consistency protocols and algorithms, including locking, atomic commit and consensus. Worse still, the semantics of existing query languages failed to capture the fundamental uncertainty in distributed executions, in which the consequences of certain deductions can be lost or arbitrarily delayed.

The disorderly programming language Dedalus [9, 7] extends Datalog with a small set of temporal operators meant to intuitively capture state change and uncertainty—the signature features of distributed systems—within a relational logic paradigm. Dedalus preserves many of the key benefits of declarative networking languages: by uniformly treating state, events, and messages as data, it encourages high-level, disorderly implementations of distributed protocols and applications, delivering the benefits of the declarative programming paradigm. More importantly, by capturing asynchronous communication and the possibility of failure in its formal semantics, Dedalus lays a foundations that enables the formal study of the relationship between logical semantics and consistency in distributed systems.

**Bloom: A disorderly language for distributed programmers**

Dedalus programs are executable, despite the fact that they read like specifications and are amenable (as we shall see) to powerful analysis techniques. Nevertheless Dedalus lacks many desirable features for a practical
programming language, including mechanisms for encapsulation and reuse and a syntax familiar to programmers. **Bloom** [8, 6, 3]—a variant of **Dedalus** embedded in Ruby as an internal domain-specific language—provides these and other usability features while preserving the semantic core of **Dedalus**.

Since its release in 2011, Bloom has grown a wide base of users. It has been utilized for protocol implementation (e.g. Raft at Stanford), as an orchestration language for cooperating solvers at UCLA/Viewpoints Research Institute, as a host language for WebDAMLog at INRIA, and as a pedagogical vehicle for teaching distributed systems at UC Berkeley. Most recently, the developers of the Eve programming environment—which is largely inspired by Bloom—raised 2.3M in funding from Andreessen Horowitz.

**Confluence Analysis**

Disorderly programming languages make writing order-insensitive (and hence “eventually consistent”) programs the natural mode. However, in order to remain sufficiently general to program arbitrary systems, such languages do not prevent programmers from writing programs that are sensitive to message ordering. How do I know if I have succeeded in writing a fault-tolerant, eventually consistent distributed program? If I have failed, what is the most efficient repair strategy for ensuring that the program produces correct outcomes in the face of uncertainty?

Given the foundation of disorderly languages for programming distributed systems, my research focused on programming tools that help tame distributed uncertainty and provide assurances about program outcomes.

**Monotonicity Analysis: Ensuring consistent outcomes despite asynchrony**

Some programs are robust in the face of uncertainty, producing deterministic outcomes despite pervasive nondeterminism in their executions. Monotonicity analysis identifies such programs by focusing on when program logic causes distributed data to change in deterministic ways regardless of scheduling nondeterminism.

A unique feature of the Bloom language is its ability to perform a static “consistency analysis” of submitted programs, providing visual programmer feedback (based on an annotated dataflow graph representation of the program logic) that identifies computations that could produce nondeterministic results when evaluated in a distributed system. This static analysis is based on the CALM Theorem [8], which establishes that monotonic programs produce deterministic outcomes despite nondeterminism in message delivery order. Because both nonmonotonic (i.e., order-sensitive) operations and asynchronous (i.e., order-sacrificing) communication are exposed in Dedalus’s syntax, monotonicity analysis can both warn programmers of the potential for inconsistent outcomes and pinpoint individual program statements as repair candidates.

**Blazes: coordination analysis and synthesis**

Monotonicity analysis is essentially a filter; programs that pass are guaranteed to have coordination-free deterministic outcomes in all executions. However, intuition tells us that some programs simply require coordination, just as some operations (e.g., logical negation and set minus) are fundamentally nonmonotonic. Large-scale systems are likely to involve a combination of monotonic and nonmonotonic components—in such cases, must we throw out the baby with the bathwater and fall back on a classic “strongly consistent” system architecture? Can we exploit monotonicity and other application-specific semantics to achieve “minimally-coordinated” executions?

Blazes [2, 4] extends monotonicity analysis into a language-independent framework that not only identifies potential consistency anomalies in under-coordinated systems, but remedies them by augmenting the given program with judiciously-chosen coordination code. Blazes’ analysis is based on a pattern of properties and composition: it begins with key properties of individual software components, including order-sensitivity, statefulness, and replication; it then reasons transitively about compositions of these properties across dataflows that span components. Second, Blazes automatically generates application-aware coordination code to prevent consistency anomalies with a minimum of coordination. The key intuition exploited by Blazes is that even when components are order-sensitive, it is often possible to avoid the cost of global ordering without sacrificing consistency. In many cases, Blazes can ensure consistent outcomes via a more efficient and manageable protocol of asynchronous point-to-point communication between producers and consumers—called sealing—that indicates when partitions of a stream have stopped changing. These partitions are identified and “chased” through a dataflow via techniques from functional dependency analysis.
When systems are written in Bloom or Dedalus, Blazes utilizes static monotonicity analysis to localize the source of consistency anomalies to concrete code segments that become candidates for coordination interposition. If the system is written in a language for which monotonicity analysis is not available (e.g., Apache Storm components implemented in Java), Blazes relies on simple semantic annotations provided by the user that characterize the order-sensitivity and statefulness of each component.

Lineage-driven fault injection

The majority of my research has focused on the problem of asynchrony in distributed systems—identifying programs that are tolerant to message reordering and choosing efficient coordination strategies for programs that are not. The lineage-driven fault injection (LDFI) project rounds out the suite of support tools by ensuring that distributed programs are fault-tolerant—that is, that they produce correct outcomes even when a variety of failures (such as message loss, node failure and network partition) occur during their execution.

Like Blazes and Bloom consistency analysis, LDFI [1] takes advantage of the clean semantic core of Dedalus to reason about the effects of nondeterminism arising in a distributed execution. However, instead of using program syntax to reason statically about the space of possible executions, LDFI uses data lineage produced in concrete executions to directly connect system outcomes to the data and messages that led to them. Fine-grained lineage allows LDFI to reason backwards (from effects to causes) about whether a given correct outcome could have failed to occur due to some combination of faults. Rather than generating faults at random (or using application-specific heuristics), LDFI chooses only those failures that could have affected a known good outcome, exercising fault-tolerance code at increasing levels of complexity. Injecting failures in this targeted way allows LDFI to provide completeness guarantees like those achievable with formal methods such as model checking. When bugs are encountered, LDFI’s top-down approach provides—in addition to a counterexample trace—fine-grained data lineage visualizations to help programmers understand the root cause of the bad outcome and consider possible remediation strategies.

I am currently collaborating with the Netflix engineering team to integrate the LDFI approach into their “simian army” production fault-injection framework. The LDFI publication is currently under review.

Smaller Projects

BloomUnit [5] is an automated declarative testing framework for programs written in the Bloom language. Correctness assertions are posed as Bloom queries over execution traces, while program inputs are automatically generated from first-order constraints. In order to test the resilience of submitted programs to message reordering and loss, BloomUnit interposes on the Bloom runtime to simulate network nondeterminism. To make the search of schedules tractable, BloomUnit uses confluence analysis to pinpoint exactly the program statements that could produce different outcomes under reordering.

Future Work

An advantage of a virtuous circle is that we can enter or exit it at any stage. There are rich opportunities to push farther in the spaces of systems, languages and analysis, both in the long and near term.

Systems: The time is ripe to revisit the BOOM Analytics agenda: use the current generation disorderly languages and confluent analyses to build a next-generation “big data analytics” stack. Using disorderly languages such as Bloom will allow rapid development of storage and computational infrastructure by focusing on data and transformation rather than low-level details such as physical data layout and fine-grained orchestration of computation and communication. Blazes can determine if additional coordination is required to guarantee application-specific correctness properties: if so, it will augment the system with the minimal coordination mechanisms necessary to uphold those guarantees. Finally, LDFI can provide strong guarantees that the stack is resilient to the variety of failures that will become more likely as we increase the scale of the system.

Building large-scale systems from the bottom up using experimental languages has great promise, but does not immediately provide a path to adoption for practitioners in the field. I want to further explore the possibility of exploiting the “grey box” approach used by Blazes to allow my analysis techniques to be applied to existing systems code based on API-level annotations.
Languages: In his Turing Award Lecture, Jim Gray sketched the grand challenge of automatic programming: “we have to (1) have a high level specification language that is a thousand times easier and more powerful that the current languages, (2) computers should be able to compile the language, and (3) the language should be powerful enough so that all applications can be described.” Executable specification languages such as Dedalus hint that it is possible to achieve the first two goals even in a stubbornly difficult domain such as distributed systems. I plan to continue exploring the use of declarative, data-centric languages for general-purpose programming. There is a great deal still to do.

I would like to explore applying the disorderly programming philosophy to the debugging of large-scale data management systems. Existing debugging mechanisms (including statement-level debuggers and tracing and replay infrastructures) reflect the sequential model of computation underlying most current programming languages, and focus the programmer’s attention on stepwise computation rather than on the modification and movement of data. LDFI—which used data lineage extracted from execution traces not just to identify bugs, but to explain those bugs to the programmer at an appropriately high level of abstraction—hints at the promise of “declarative debugging” techniques for large-scale systems.

Analysis: An immediate avenue of future research is to explore opportunities to enrich (and in some cases combine) the confluence analysis techniques sketched above. Just as Blazes identifies and repairs dataflows that (according to its static analysis) could produce nondeterministic outcomes, I want to extend LDFI to augment programs that fail to uphold their fault-tolerance specifications, by applying techniques from software synthesis to provide additional redundant support for the desired outcomes. I am also exploring an input synthesis project that combines features from BloomUnit and LDFI. Instead of generating covering inputs from first-order constraints supplied by the programmer, input synthesis uses the program itself to generate relevant input-generating constraints that provide full coverage of the program code, and can be used together with LDFI to discover bugs. Blazes can be further improved by applying the LDFI methodology: instead of conservatively coordinating based on static analysis results, LDFI can (guided by the Blazes static analysis) exploring targeted messages interleavings to produce concrete visualizations of potential inconsistencies and their root causes.

References