## Project Summary

No software is perfect. Even the most carefully designed programs must be periodically updated to fix bugs, patch security vulnerabilities, add features, and eliminate performance bottlenecks. In systems where brief periods of downtime can be tolerated, updates can be deployed by simply halting the program, updating the code and data, and restarting the program at the new version. But in systems that provide critical infrastructure such as air traffic control, power grid monitoring, and financial processing, even brief service interruptions for maintenance are unacceptable—the system must remain available before, during, and after the update.

The research community has developed a number of techniques for updating programs on the fly. But existing dynamic software update frameworks have a fundamental limitation: they only handle updates to programs executing on a single machine. Of course, dynamic updates are also important in the distributed setting—arguably more important!—but they are also more difficult, because the whole system cannot be updated simultaneously. Hence, during an update, the system will necessarily contain some nodes running the old version of the program and others running the new version. To address this challenge, programmers today use a variety of ad hoc techniques that handle some simple situations but fail dramatically in others, leading to catastrophic and costly errors.

This proposal presents a comprehensive research and education plan designed to transform the way that programmers design, reason about, implement, and deploy updates to distributed systems. The central feature of this project is the use of bidirectional transformations to bridge the gap between different versions of a program, thereby enabling multiple versions of a program to seamlessly inter-operate while an update propagates through the system. To ensure that it starts from a solid foundation, the project will develop a formal model that captures the essential features of distributed systems during updates, and provides a basis for separating correct behavior from incorrect behavior. To replace the ad hoc, manual, and error-prone techniques used today, it will develop new programming constructs for designing, implementing, and deploying updates. To implement these constructs, it will build a practical system that handles all of the low-level details needed to realize updates such as adding versions to data, transforming state, and coordinating behavior between nodes. To help programmers reason about the behavior of a system during an update, it will develop verification tools capable of automatically answering questions such as: "is this update backward compatible?" and "are new behaviors are enabled by this update?" Finally, to integrate these efforts and demonstrate the effectiveness of the approach, it will use the system to develop real applications.

**Intellectual Merit:** The main contributions of this project are: (1) a formal foundation for distributed updates based on bidirectional transformations, (2) new programming constructs for managing distributed updates, (3) verification tools that check formal properties of updates automatically, and (4) an implementation and validation on real applications.

**Education goal:** This project will encourage the next generation of computer scientists by developing curricula at two levels: (1) a workshop for high school students aimed at broadening participation in science generally and in computer science specifically, and (2) a graduate course at the intersection of programming languages and distributed systems that will draw connections between the areas and demonstrate how language-based techniques can be leveraged to solve challenging problems in systems.

**Broader impacts:** This project will have a positive, direct impact on society by providing programmers with constructs for implementing updates that dramatically increase their ability to build correct software. It will encourage a community of researchers in the inter-disciplinary area between programming languages and systems, by developing reusable software infrastructure and making it available under an open-source license. Finally, it will improve K-12 education and increase the participation of underrepresented minorities through a close educational partnership with a technology-focused high school.

**Key Words:** Dynamic software updates, distributed systems, bidirectional transformations, lenses.

## 1 Introduction

No software is perfect. Even the most carefully designed programs must be periodically updated to fix bugs, patch security vulnerabilities, add features, and eliminate performance bottlenecks. In systems where brief periods of downtime can be tolerated, updates can be deployed by simply halting the program, updating the code and data, and restarting the program at the new version. But in systems that provide critical infrastructure such as air traffic control, power grid monitoring, and financial processing, even brief service interruptions for maintenance are unacceptable—the system must remain available before, during, and after the update.

Over the past decade, the research community has developed a number of techniques for updating programs on the fly. In these dynamic software update frameworks, the programmer provides a patch that contains the code for the new program as well as a function that transforms the state from the representation used by the old version to the representation used by the new version. The framework applies the patch to the system as it is running in a way that guarantees that important safety properties, such as type safety, are preserved. Dynamic updates have been successfully applied to programs in a wide range of real-world settings including web servers, databases, operating systems, and many others [3, 14, 54, 55, 58, 36, 11, 1, 52, 6].

Unfortunately, with few exceptions [6, 1, 52], existing dynamic update frameworks have a serious limitation: they assume that the program being updated is executing on a single machine. Of course, dynamic updates are also important in the distributed setting—arguably more important!—but they are also fundamentally more difficult because the whole system cannot be updated simultaneously. Hence, during an update, the system will necessarily contain some nodes running the old version of the program and others running the new version. To address this challenge, programmers today typically use ad hoc techniques that work in some simple scenarios but fail dramatically in others. For example, in a so-called rolling upgrade, individual machines are taken down, upgraded, and brought back up, so that the overall service continues without interruption. This strategy works fine when the new program is fully backward compatible with the old program, so that old and new nodes can inter-operate without issue. But programmers lack tools to check that a given given satisfies the (often unstated) assumptions needed to use an update mechanism correctly. As a result, the effects of an update are difficult for programmers to predict and errors are both frequent and costly.

These concerns are not hypothetical. On the contrary, failures triggered by updates occur often and have catastrophic effects. Here are three recent examples of many: in June 2012, a glitch upgrading back-end servers at the Royal Bank of Scotland delayed payroll, halted payments, and froze the accounts of millions of customers for days [33]; in April 2011, an error during a routine update to network configurations at Amazon triggered a sequence of cascading failures that eventually took down an entire datacenter, along with hundreds of customer websites hosted there [2]; and in June 2011, an upgrade to the air traffic control system at the Federal Aviation Administration caused problems with the radar system and forced flights across the country to be grounded [61]. Each of these episodes took several days to unwind and were extremely costly for the entities involved, with losses on the order of millions of dollars.

Clearly better mechanisms for managing updates are needed.

### 1.1 Research and Education Goals

This proposal presents an integrated research and education plan designed to transform the way that programmers design, reason about, implement, and deploy distributed updates. To replace the ad hoc, manual, and error-prone techniques used today, I will design new constructs for deploying updates based on a solid semantic foundation. To implement these abstractions, I will develop systems that handle all of the low-level details needed to realize updates such as adding versions to data, transforming state, and coordinating behavior across multiple nodes. To help programmers reason about the behavior of a system during an update, I will build verification tools capable of automatically answering questions such as: “is this patch backward compatible?” and “are new behaviors are enabled by this update?” To gain practical experience with distributed updates and to help cultivate a community of researchers in the inter-disciplinary area between programming languages and systems, I will develop reusable infrastructure and applications and disseminate them broadly under an open-source license. Finally, to seed the next generation of computer scientists, I will develop curricula at two levels: a graduate course, and a high-school course through a partnership with teachers from a technology-focused magnet school in Philadelphia, the New York State 4-H, and the US Geological Survey.

## 1.2 Contributions

The fundamental challenge in implementing distributed updates is that it is impossible to update every node simultaneously. Hence, while an update is being propagated to each node, the system will necessarily contain a mixture of nodes—some running the old version of the program and others running the new version. To ensure that the overall behavior of such a system is correct, the entity managing the update must either ensure that nodes running different versions of the program cannot interact, or that whenever they do, they can inter-operate without issue. The ad hoc techniques discussed previously are based on this intuition. For example, rolling upgrades assume backward compatibility, so that new nodes can be safely used as old nodes. This is effective, but it rules out many important updates including ones that fix bugs and add features. A different approach is to use version numbers and prevent nodes running different versions of the program to interact. This strategy eliminates the need for nodes to inter-operate, but it hinders availability—the primary reason for using dynamic updates!—since it partitions the system into two disjoint pieces that cannot interact.

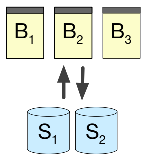
This proposal suggests a better alternative. Instead of forcing programmers to limit themselves to backward compatible updates or restrictive update protocols, it presents a framework for implementing distributed updates that comes equipped with abstractions for allowing one version of a program to inter-operate with another. Using these abstractions, programmers can deploy updates without worrying about harmful interactions between differently-versioned nodes, and they can reason about the overall behavior of the system by understanding the individual programs and the relationships between them.

This approach to implementing dynamic updates is not entirely new. Similar ideas have been used in work on dynamic updates [14, 60] and schema evolution [15, 47]. But previous systems require programmers to construct the simulations manually—a tedious and error-prone task as it involves implementing two transformations. A better idea is to use a language specifically designed for describing bidirectional transformations. In a bidirectional language, every program can be read as a function mapping inputs to outputs, and also as a function mapping outputs back to inputs. Moreover, the semantics of the language typically guarantees important correctness properties by construction. However, although many different bidirectional languages have been proposed in recent years [17, 9, 8, 21, 56, 48, 37, 43, 38, 57, 56, 39, 53, 13, 62, 59, 16, 20, 12], none is sufficient for constructing the simulations needed to implement updates. They either provide weak semantic guarantees or cannot express the kinds of transformations that arise in practice.

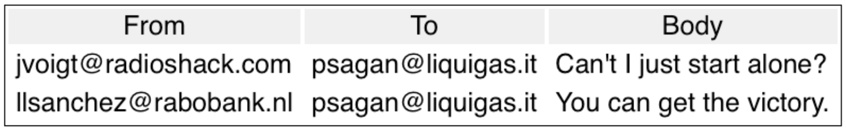
The overall goal of this project is to develop a new foundation for distributed updates based on bidirectional transformations. To that end, work on the project will be organized around four main threads of activity:

- **Semantic foundations:** Distributed updates today are typically implemented using ad hoc mechanisms with unclear semantics. To facilitate making precise mathematical statements about the behavior of a system during an update, I will develop a formal model that captures the essential features of distributed updates and bidirectional transformations, and distill general properties satisfied by updates and transformations that are intuitively reasonable. This model will serve as the foundation for the rest of the project.

- **Programming abstractions:** Programmers today have little help in implementing distributed updates. I



(a)



| From | To | Body |
|---|---|---|
| jvoigt@radioshack.com | psagan@liquigas.it | Can't I just start alone? |
| llsanchez@rabobank.nl | psagan@liquigas.it | You can get the victory. |

(b)

| | **JSON data** | **JavaScript program** |
|---|---|---|
| v1 | see below | see below |
| v2 | see below | see below |

v1 — **JSON data**

```
data =
 [ { "header" : ("From: jvoigt@radioshack.com\n" +
                 "To: psagan@liquigas.it"),
     "body"   : "Can't I just start alone?" },
   { "header" : ("From: llsanchez@rabobank.nl\n" +
                 "To: psagan@liquigas.it"),
     "body"   : "You can get the victory." } ]
```

v1 — **JavaScript program**

```
for (var i = 0; i < data.length; i++) {
  var row = $('<tr/>');
  fs = data[i]["header"].match(/: ([^\n ]+)/g);
  row.append($('<td/>').html(fs[0].slice(2)));
  row.append($('<td/>').html(fs[1].slice(2)));
  row.append($('<td/>').html(data[i]["body"]));
  $("#msgs").append(row);
}
```

v2 — **JSON data**

```
data =
 [ { "unread" : true,
     "from"   : "jvoigt@radioshack.com"
     "to"     : "psagan@liquigas.it",
     "body"   : "Can’t I just start alone?" },
   { "unread" : true,
     "from"   : "llsanchez@rabobank.nl"
     "to"     : "psagan@liquigas.it",
     "body"   : "You can get the victory." } ]
```

v2 — **JavaScript program**

```
for (var i = 0; i < data.length; i++) {
  if (data[i]["unread"]) {
    var row = $('<tr/>');
    row.append($('<td/>').html(data[i]["from"]));
    row.append($('<td/>').html(data[i]["to"]));
    row.append($('<td/>').html(data[i]["body"]));
    $("#msgs").append(row);
  }
}
```

(c)

Figure 1: Web application: (a) system architecture; (b) browser display; (c) JavaScript code.

will develop simple, intuitive abstractions for building distributed updates that guarantee correct behavior by construction. Programmers will be able to specify an update using these constructs, and a compiler and run-time system will automatically generate the low-level artifacts needed to implement them.

- **Verification tools:** Programmers today lack tools for checking formal properties of updates. As a result, the effects of an update are difficult to predict ahead of time, and unexpected interactions triggered by updates happen frequently. I will develop tools for documenting and validating formal assertions about the behavior of a system during an update.
- **Deployment:** To integrate the main threads of this project and provide a basis for evaluating success, I will build several real-world web applications using the technology developed in the rest of this project.

These goals are both foundational and pragmatic. At the foundational level, I seek to discover the principles that underpin distributed updates and design robust abstractions that make it easy for programmers to both construct updates and reason formally about their behavior. On the practical side, I plan to realize these abstractions in a practical system that includes convenient syntax, an efficient implementation, and a suite of examples that other researchers can use, adapt, and build upon.

## 2 Background

This section presents some background material on distributed updates and bidirectional transformations to set the stage for the research described in subsequent sections.

**Example.** To illustrate the challenges that arise with distributed updates, consider a web application that implements a simple email client. The system consists of multiple browsers and back-end servers as shown in Figure 1 (a). The servers maintain the persistent copies of the mailboxes in a database. The browsers perform two tasks: they fetch messages from the servers, and they render those messages graphically. In addition, to reduce latency and to facilitate offline use of the application, each browser stores messages in a local cache.

Initially, the browsers execute the code labeled "v1" in Figure 1 (c). The value on the left shows a cache value represented in JavaScript Object Notation (JSON). It consists of a list of records with two fields, `header` and `body`; the `header` field contains the email addresses of the sender and recipient. The JavaScript code on the right shows the browser code that renders a cache value as a table in the browser as shown in Figure 1 (b). We elide the server program but assume it uses an analogous representation for messages.

Now suppose we change the application, modifying the message representation by adding an `unread` flag and replacing `header` with explicit `to` and `from` fields. The "v2" code in Figure 1 (c) shows the new version of the cache value and JavaScript code. Note that although the "v1" and "v2" programs are quite similar, they cannot inter-operate—executing the "v2" JavaScript code on a "v1" cache value causes a run-time error.

**Dynamic updates.** The simple way to implement the update from "v1" to "v2" is to bring the servers down, migrate their databases to the new schema, and refresh the browsers when the servers come back online. This strategy successfully implements the update, but it has several disadvantages: it makes the service unavailable during the update, and it discards changes made to local caches. A better approach is to update the code and on the fly using a dynamic update framework. The programmer builds a patch containing the "v2" code along with a state transformer that converts the cache from the old version to the new version, and the dynamic update framework applies the patch to the running program without disrupting service or losing state.

Unfortunately, in the distributed setting, things are not so simple. The system consists of multiple nodes that cannot be updated simultaneously. Even if one machine is updated to the new version correctly, that machine may communicate with another that has not been updated—a situation that leads to a run-time error. For example, if a browser is updated before the server it communicates with, the server might send a "v1" value to "v2" browser, causing a run-time error when the JavaScript code attempts to render that message in the DOM tree; conversely, if the server is updated first, it might send a "v2" value to a "v1" browser, leading to a similar error.

**Bidirectional transformations.** To ensure that the system behaves reasonably during an update, we must either ensure that "v1" and "v2" never interact, or develop mechanisms that allow them to inter-operate. Carefully designed update protocols can be used to prevent harmful interactions between machines at different versions, but these protocols make certain nodes unavailable to others and force updates to be applied at particular times, which might be inconvenient. For example, we might prefer to allow active sessions to complete before applying the update to avoid disrupting the graphical interface. We might also wish to transform the state on servers when load is light. But to use either of these more flexible update strategies, we have to execute multiple versions of the program simultaneously on the same node.

To achieve this, we need some way to transform old data into new data, and vice versa. For example, given the "v1" message,

```
{ "header" : "From: jvoigt@radioshack.com\nTo: psagan@liquigas.it",
  "body"   : "Can't I just start alone?" }
```

we need to be able to produce the corresponding "v2" message,

```
{ "unread" : true,
  "from"   : "jvoigt@radioshack.com"
```

```
    "to"     : "psagan@liquigas.it",
    "body"   : "Can't I just start alone?" }
```

and similarly in the reverse direction.

If we had such a bidirectional transformation, each version of the program would be able to masquerade as the other version. For example, if a "v2" browser retrieved a message from a "v1" server, it could convert the message into the new representation using one half of the transformation. Conversely, if a "v1" browser retrieved a message from a "v2" server, it could convert the message into the old representation using the other half of the transformation. Note however that because the transformation is not bijective—in particular, due to the presence of the `unread` field in the "v2" representation—the simulation of a "v2" node by a "v1" machine would be imperfect. In this case, multiple conversions from new to old and back to new again could cause the `unread` field to be reset to a default value. Addressing this problem is one of the goals of this proposal.

**Lenses.** In general, to build a bidirectional transformation, the programmer needs to implement two functions, one in each direction. Moreover, the two functions should not be arbitrary, but should work well together—*e.g.*, they should compose to the identity function in either order. The straightforward way to build a bidirectional transformation is to write two separate functions. But this approach is tedious for programmers, leads to redundant code, and is prone to errors. A better way is to use a domain-specific languages for describing well-behaved bidirectional transformations, often called *lenses*. Programs in these languages can be interpreted both forwards and backwards and their type systems ensure that the functions they denote are well behaved by construction.

Formally, a lens mapping between sets of structures $A$ and $B$ consists of a pair of functions, $get$ and $put$ that obey the laws below for every $a$ and $b$:

$$\begin{array}{l} l.get \in A \to B \\ l.put \in B \to A \to A \end{array} \qquad \frac{}{l.put\ (l.get\ a)\ b = a}\ \text{GetPut} \qquad \frac{}{l.get\ (l.put\ b\ a) = b}\ \text{PutGet}$$

Note that the types of $get$ and $put$ functions are asymmetric. This allows the $get$ to discard information in computing a value of type $B$. The $put$ function takes as arguments a possibly modified $B$ as well as the original $A$. It propagates the information in its $B$ argument and restores the discarded information from its $A$ argument to yield a new $A$. This asymmetric design ensures that the behavior of lenses are reasonable, even when the structures being transformed have different information. For example, to map between the "v1" and "v2" message representations, we can use a lens whose $get$ function maps new messages to old messages (deleting the `unread` field and concatenating the `to` and `from` fields into a single `header` field), and whose $put$ function performs the reverse transformation (splitting the `header` field back into `to` and `from` and restoring the `unread` field from the original new message).

As mentioned above, one limitation of lenses is that they their semantics is somewhat weak with respect to the integrity of data. Specifically, the GetPut law only requires that the $A$ value be restored when $put$ is presented with a $B$ value and the precise $A$ value that generates that $B$ via $get$. If the $B$ value has been modified in any way, the lens laws say nothing about how the discarded information in the $A$ value must be treated. This is the root of the problem with using lenses to map between the different representations of email addresses discussed previously—the `unread` field is discarded by the $get$ function, but the lens laws do not mandate that it be restored by the $put$ function. One way to strengthen the semantics of lenses is to add an additional law,

$$\frac{}{l.put\ b_1\ (l.put\ b_2\ a) = l.put\ b_1\ a}\ \text{PutPut}$$

which states that the effect of performing two $put$s in a row must be the same as just performing the outer $put$. In effect, this law rules out lenses that have any side effects on the discarded $A$ information, since it requires the lens to preserve the information—in some form—after each $put$. To see this, consider the following calculation, which shows the PutPut law provides a way to "undo" the effects of any invocation of $put$.

$$\begin{array}{lll} & l.put\ (l.get\ a)\ (l.put\ b\ a) & \\ = & l.put\ (l.get\ a)\ a & \text{by PutPut} \\ = & a & \text{by GetPut} \end{array}$$

Unfortunately, the PutPut law has some undesirable implications: it rules out transformations such as iteration and conditional operators that are often important in practice. Hence, most lens languages treat it as optional.

## 3 Research Plan

Work on this project will center around four threads of activity, each focusing on a different aspect of distributed updates. This section describes these threads in detail.

### 3.1 Semantic Foundations

Before we can develop abstractions and tools, we must have a clear notion of what it means for a distributed update to be correct. Accordingly, the first thread of activity in this project will investigate the foundations of distributed updates, with the goal of developing a formal model and precise conditions that can be used to separate correct behavior from incorrect behavior. This section discusses three of the most important foundational issues: correctness of updates on a single machine, semantics for bidirectional transformations, and consistency.

**Update correctness.** What does it mean for a dynamically updated program to be correct? A straightforward answer, based on the classic axiomatic notion of correctness, is that a dynamically updated program is correct with respect to a formal property if every trace generated by executing the program satisfies the property. But for programs that are updated dynamically, this notion is awkward to use—it requires reasoning explicitly about the old program version, the new program version, and code that transforms the state of the old version into the representation expected by the new version. Moreover, this reasoning must be performed at each possible program point when an update might be applied—a tricky proposition that quickly becomes unfeasible. Existing frameworks for managing dynamic updates are no help here as they either do not address correctness at all, or they focus exclusively on generic safety properties that rule out obviously wrong behavior but are insufficient for establishing correctness with respect to specific properties [14, 54, 55, 58, 31].

A number of different correctness conditions have been proposed in the literature, but none is completely satisfactory. Early work by Kramer and Magee [35] suggested that an update be considered correct if it preserves all behaviors of the old program. But as noted by Bloom and Day [7], this rules out updates that fix bugs or add new features. Gupta et al. [28] proposed that an update is considered correct if it eventually transitions to a state that is reachable by executing the new version of the program from scratch. But this condition is also unsatisfactory: it imposes no constraints on the behavior of the system between when the update is applied and the eventual convergence to a valid state of the new program. In addition, taken literally, it forces the state transformer to purge any data such as audit logs and system statistics that are not consistent with the new program state. Work by Hayden et al. [30], identified several intuitive categories of updates including backward compatible, post-update, and conformable properties. But although these categories cover many updates that arise in practice, they do not constitute a complete theory of correct updates.

To address the important problem of update correctness, I plan develop a formal model that models programs with updates, and encode a large number of update mechanisms and examples from the literature and practice into this model. This will provide a unified setting for comparing the conditions that have been proposed previously, and make it possible to formally relate them. I also plan to investigate the issue of update correctness from a logical perspective, studying the classes of properties preserved by specific mechanisms. Overall, the goal of this investigation will be to distill general principles that capture the essence of update correctness. But even if no such "grand unified theory" of updates can be found, I am optimistic I will be able to discover and formalize specific notions such as backwards compatibility and Gupta's eventual convergence condition that capture common cases and can be used as a foundation for developing useful language abstractions and tools.

**Bidirectional Transformations.** Another important issue concerns the semantics of the bidirectional transformations needed to ensure that two different versions of a program can inter-operate. Here the desired criterion is clearer: the transformations must have the property that each version of the program simulates the other when composed with the transformations. Lenses and their semantic properties provide a good starting point for designing such transformations, but current proposals for lenses are limited in two essential ways: either their laws do not provide strong enough guarantees to guarantee correct simulation, or they only allow discarding information in one direction. This work will develop extensions of lenses that address these limitations, providing a solid foundation for building transformations that bridge the gap between different versions of a program.

Recall the lens for converting between "v1" and "v2" values described in the preceding section. Beneath the clean abstraction provided by the lens are two serious problems. The first is that the lens laws only guarantee that the information in the source is preserved on round-trips if the view is not changed. If a "v2" node interacts with a "v1" node using the lens described previously, the `unread` field could be reset. The root of this problem is that the lens laws—in particular, the GetPut law—only provide a weak integrity guarantee. The second problem is that lenses cannot be used to implement transformations that discard information in both directions. For example, if old messages contained additional information in the `header` field—*e.g.*, perhaps indicating the likelihood that the message is spam—we would not be able to build a lens that implements the transformations between old and new messages because the PutGet law implies that the $put$ function must be injective in its first argument.

I plan to develop a refined semantic foundation for lenses that overcomes both of these limitations. There are several possible ways this work could proceed. One simple approach is to define a family of lenses that map between a representation that contains the complete information of all versions and the individual versions. This technique is already used in industry for managing updates to network protocols [41], but the semantic properties of such transformations have not been investigated. Another approach is to extend the symmetric lenses proposed by Hofmann et al. [32] with stronger integrity guarantees. In a symmetric lens, the information discarded by each transformation is represented explicitly as a *complement* of type $C$. The $get$ and $put$ functions have symmetric types and obey the laws below for every $a$, $b$, and $c$:

$$\begin{array}{l} l.get \in A \times C \rightarrow B \times C \\ l.put \in B \times C \rightarrow A \times C \end{array} \qquad \frac{l.get\ (a,c) = (b,c')}{l.put\ (b,c') = (a,c')}\ \text{GetPut} \qquad \frac{l.put\ (b,c) = (a,c')}{l.get\ (a,c') = (b,c')}\ \text{PutGet}$$

This design allows both transformations to discard information, but it weakens the round-tripping laws substantially. Specifically, there are few constraints on how the complement is used, so information in the complement is not guaranteed to be preserved on round-trips. I would explore whether adding additional laws concerning the handling of the information in the complement—*e.g.*, PutPut-like laws that rule out harmful side effects—can be used to strengthen symmetric lenses to provided strong enough integrity guarantees. A third approach is to develop a semantics based on invertible schema mappings, which have been proposed as an approach for solving schema evolution problems [15]. A side-benefit of this effort would be to deepen our understanding of the connections between lenses and the large body of work on schema mappings.

**Consistency.** Consistency is a fundamental concern in any distributed system. In the context of distributed updates, consistency issues can arise when nodes interact multiple times at different versions. As a motivating example, consider a data store that implements an access control mechanism. In the initial version of the program, to access a value, the programmer must invoke $o_1$ followed by $o_2$. The first invocation retrieves the value from the store but does not return it, and the second invocation verifies that the user is authorized to access the value and returns it if so. In the new version of the program, the same operations are used to access a value, but they perform the opposite task: $o_1$ handles authentication and $o_2$ retrieves the value. Both versions of the program correctly implement the access control mechanism. However, if the system is updated between the two operations—after the old version of $o_1$ but before the new version of $o_2$—the user may be incorrectly granted access to the value without performing the required access control check.

Intuitively, the problem with this example is that correctly simulating individual operations does not ensure that the overall behavior of a program will be preserved. To put it another way, to check that a given property is an invariant of a program with dynamic updates, it does not suffice to check that the old and new programs each satisfy the property individually. Instead, the programmer must check that it holds for executions obtained by interleaving operations from the old program with operations from the new program. Hence, consistency issues make reasoning about programs significantly more complicated.

To address this problem, I plan to enrich the semantic framework with conditions for ensuring that sequences of operations are executed using a consistent program version. First, I will extend the semantic framework to model the behaviors generated in a distributed system with updates. This model will likely be formalized as a simple process calculus, with multiple nodes executing concurrently and communicating with each other using message passing [5]. Second, I will identify conditions that guarantee the intuitive notion of consistency articulated above. Intuitively, these conditions will require that certain behaviors in the semantics be equivalent to the behaviors generated by a single program version. Third, I will develop update protocols that guarantee consistency by construction. For example, each node could maintain a version number, and the protocol could instrument sequences of instructions to ensure the operations are executed using a single consistent version.

## 3.2 Programming Abstractions

The second thread of work on this project will focus on developing programming abstractions for managing distributed updates. Programmers today have little help when designing, implementing, and deploying updates. As a result, they must explicitly deal with a host of low-level details including building patches, adding version numbers to data and protocol messages, constructing state transformers, and managing the propagation of patches to the nodes in the system. This work will seek to automate these tasks, thereby freeing programmers from having to worry about how to implement them, and preventing them from making errors in doing so.

To ground this investigation, I plan to work in the concrete domain of web applications. Browsers are one of the most popular platforms for developing applications today, and programmers are using them to build increasingly sophisticated applications such as word processors, photo editors, and computer games. Unlike early web applications, where most computation was located on servers and browsers merely rendered pages, most computation in applications these days happens in the browser, which executes programs written in languages such as JavaScript and Flash. In addition, although web applications used to be relatively short-lived, being generated fresh every time the user reloads the page, this is starting to change. Applications like GMail, which uses tens of thousands of back-end servers and has millions of users, allow users to store data locally to support offline use. A change to these applications is very much a distributed update, with multiple versions of the application inter-operating simultaneously within the same overall system.

This section presents a general design for a framework that supports dynamic updates, followed by a discussion of specific research challenges that must be addressed to build it.

**Overview.** For the purposes of this proposal, assume that the application is structured like the email client described in Section 2: the back-end servers provide a persistent database, the browsers cache server objects locally, and JavaScript code builds the pages themselves and handles interaction with the user. To update the program, the programmer provides a patch that contains new schemas for the database and cache, the state transformer, and the new JavaScript code. To apply the update to the application dynamically, the system performs the following tasks in order:

- It instruments the schemas and code so that all data in the system carry a version number. These version numbers will be used by the run-time system to manage the deployment of the update.
- It builds a lens that maps bidirectionally between old version data and new version data. One half of the lens will be given by the state transformer; the system constructs the other half of the lens.
- It generates run-time system code to handle the application of the update to a node locally. This code will determine a safe program point to update the patch; mediate between all interactions with other nodes by invoking the lens as needed to convert data between versions; and enforce consistency controls.
- It installs this code on every node in the system.

Note that although the actual deployment of the update in the final step is performed in an ad hoc manner—by simply installing the code on nodes one by one—the run-time system guarantees that each node simulates other versions of the system during the update, and enforces the consistency constraints imposed by the program.

**Lenses.** Lenses are the key enabling technology that make the simple implementation of distributed updates sketched above possible. In addition to enriching the semantics of lenses with strong integrity guarantees as discussed previously, I also plan to explore new syntax for lenses and new ways of generating lenses automatically from existing programs. Somewhat paradoxically, these languages may be easier to design than traditional lens languages, since imposing additional semantic constraints means that fewer behaviors can be represented. One possible approach is to have the programmer annotate the old and schemas to indicate the data common to both versions, and generate the lens from this description. This would yield a formalism similar to the grammar-based languages that have been proposed in other domains [34, 10, 51, 22]. Another idea is to synthesize the lens from the state transformer itself, as in work on bidirectionalization [38, 57, 56, 26]. A third possibility is to explore the logic-based approach used by the database community in work on schema mappings [15, 40]. Of course, with any of these approaches, I will also develop program analyses that check well behavedness automatically.

**Consistency.** Because it will be possible for multiple versions of a program to execute simultaneously in a given system, programmers will need mechanisms for restricting interactions between different program versions. For example, to correctly implement an authorization check or a financial transaction, it might be important to ensure that the entire sequence of operations is executed using a single program version. I plan to enrich the programming model with additional constructs for grouping sequences of operations into lists that must be executed consistently. I will also extend the run-time system with the controls needed to implement these constructs—*e.g.*, using explicit version numbers.

**Optimization.** Using lenses to mediate between different versions of a program could lead to an unacceptably poor performance, especially if multiple updates are deployed simultaneously in the system. I plan to develop algebraic optimizations that can be applied to replace inefficient lens programs with efficient ones. For example, a lens that first maps a transformation over a large collection and then selects a subset of the resulting items, could be replaced with a lens that selects the items first and applies the map second. I also plan to investigate methods for improving the performance of the overall system using classic techniques from functional programming. For example, lazy evaluation could be used to avoid applying the lens to values that are not used in the rest of the program, and program fusion and partial evaluation could be used to optimize away the intermediate values generated when the lens is composed with JavaScript code.

## 3.3 Verification Tools

One of the most difficult challenges that programmers face when dealing with distributed updates today is that the effects of updates are hard to predict. Reasoning about a distributed systems is already difficult enough—in general it involves reasoning about every possible interleaving of the behaviors exhibited by the individual nodes—and reasoning about a system with updates is even more difficult, as it allows multiple programs to be involved in generating behaviors at each node.

One way that programmers can make this situation more tractable, is by understanding the relationship between the old and new versions of the program. For example, if an update merely improves performance, then the old and new programs provide the exact same functionality, and the programmer can reason about the (functional) correctness of the overall system as if it were running a single program. Unfortunately, programmers today lack ways to document and check assumptions about updates. While they might hope that a given patch implements a mere performance improvement, it could actually contain new functionality, which could then lead to errors when is deployed.

This work is to develop tools that allow a programmer to formally verifying that an update has a particular property. As examples of update properties programmers might want to verify, consider the following:

- **Reachability:** The system eventually transitions to a state that could be reached by executing the new program from scratch. That is, the execution of the system converges to a valid state of the new program, and not some unreachable state. This condition could be used by a programmer to establish that properties satisfied by the old and new programs individually are satisfied before and after an update.

- **Equivalence:** The old and new programs are equivalent: any behaviors generated by the old program can be matched by the new program, and vice versa. This could be used to check that a patch intended to improve the performance of the system (and not change functionality) does not inadvertently introduce new behaviors.

- **Conformability:** The new program simulates old program functionality modulo a specified relation on behaviors. This could be used to ensure that functionality made redundant by the update can still be correctly emulated in the new version.

- **Feature addition:** The new program is a conservative extension of the old program. That is, it adds new functionality while preserving all of the behaviors of the old program. This could be used to verify that a new feature does not interact with existing code in unexpected ways.

- **Feature deletion:** The new program strictly removes functionality from the old program. This could be used to check that a patch intended to fix a bug indeed eradicates it.

Building on the semantic foundation for distributed updates developed in this project, I plan to build program analysis and verification tools for checking these properties and others. These tools will take as input the old version of the program, the patch, and the property to check, and emit as output either an answer, a counterexample, or an indication that the tool was unable to decide whether the property holds (note that several of the properties listed above are not decidable in general).

Although formally verifying software is always challenging, I believe this effort will be successful for several reasons. First, preliminary work by Hayden et al. [30], showed that it is possible to verify several of the properties listed above on actual updates made to real-world C programs. Most of the difficulties they encountered were due to having to reason about low-level memory operations—issues that do not arise in languages like JavaScript. Second, because we are designing the programming abstractions and the verification tools together, if necessary, we can adapt the programming model to make it more amenable to verification. For example, we might impose restrictions on program structure to make the behaviors of the program explicit—*e.g.*, forcing the program to be structured as an event loop that processes user input by invoking pure functions that yield DOM modifications as output. Or we might ask the programmer to provide extra facts that are needed by the verification tool, such as a proof that the lens implements valid simulations between versions. Third, many of the updates that arise in practice do not make dramatic changes to the program. So simple heuristics for analyzing functionality may often suffice. Finally, the techniques and tools used in program analysis continue to rapidly evolve. In particular, there has been a lot of recent work on developing formal semantics and analysis tools for JavaScript [27], so it should be possible to leverage existing tools to perform many low-level reasoning tasks.

### 3.4 Deployment

The core of this project is developing foundations, language abstractions, implementation techniques, and verification tools for distributed updates. Although each piece of the project can be evaluated in isolation, the best way to demonstrate that these results constitute a genuine solution to the distributed update problem is through applications. To this end, the final thread of activity in the project will focus on using our results to develop and deploy applications. Here are three potential applications:

- **Course Pages:** Maintaining course web pages is a frequent task during the semester. To make the pages for my courses easy to edit and maintain, I already store the data for each page in a local cache on the browser and use JavaScript to render the actual DOM elements. So although course pages seem static, under the covers they are actually quite dynamic. Updates can range from adding new assignments or lecture notes to modifying the structure of a page by adding or deleting table columns. This application will serve as a "hello world" example for the system, illustrating its main features on a simple example.

- **Applicant System:** Grad student and faculty candidates currently submit application materials via a third-party website. Unfortunately, this website is complicated, slow, and buggy—*e.g.*, last year, an error caused the main research area for hundreds of applicants to be lost. This application will develop a new interface to the data managed by these systems. Updates will range from changes to the code we write, to schema changes made on the third-party website.

- **Structured Wiki:** Wikis are a popular tool for allowing multiple users to make changes to shared documents. DBWiki[1] is a system being developed by researchers at the University of Edinburgh that allows multiple users to curate structured data such as records, lists, relations, and trees through a wiki-like interface. Updates can arise in DBWiki due to changes to the schemas of the underlying data, as well as changes to the code that implements the interface. I am actively collaborating with the DBWiki designers on a system for managing the bidirectional transformations between back-end servers and browsers. This application will extend that system with support for updates.

These applications will provide valuable experience in this domain, help us to evaluate and ultimately validate our ideas, and serve as documentation for others. I plan to make all of these applications we develop

[1]`http://code.google.com/p/database-wiki`

available on the project website under an open-source license, along with the code for the system itself.

## 4 Education Plan

As a faculty member, the goals of my research and teaching are closely intertwined. So like the research presented in this proposal, my education plan seeks to integrate ideas from programming languages with ideas from systems. I describe two projects: one a new graduate course, and the other an outreach program for high school students.

### 4.1 High School Outreach

The major education component of this proposal is an outreach program for high school students. The goal of this program is to engage these students in science, technology, engineering, and mathematics (STEM) disciplines generally, and computer science in particular. As is well known from the "Gathering Storm" reports from the National Academies [45, 46], the United States is in danger of losing its global competitive edge, due to a shrinking scientifically literate workforce. This project seeks to address this threat head on, as well as several of the NSF's stated broader impacts objectives including improving K-12 education and teacher development, enhancing infrastructure for education (in the actual curriculum materials developed and through the partnerships between schools, universities, and scientists formed during the project), increasing the participation of underrepresented minorities in STEM, and increasing scientific literacy.

The proposed project has two phases, each of which will be conducted in cooperation with the New York State 4-H, the US Geological Survey (USGS), and the Science Leadership Academy (SLA) as partners (see the letters of support attached to this proposal). During the first phase of the project, I will organize a course for 20-30 high school students, run under the auspices of the 4-H's Career Explorations event. Career Explorations brings approximately 500 high school students to campus each June to give them an academic experience in a college setting, with special emphasis on career tracks in STEM disciplines. In the second phase of the project, conducted in years four and five, I will refine, package, and disseminate all course materials on the web including slides, lesson plans, labs, software, and examples. As project partner USGS has monitoring stations in all fifty states and devotes significant resources to outreach, these materials should be easy for educators around the country to put to use.

The course itself will be organized as an extended lab whose goal is developing a simple system for monitoring the local water system. The campus has several small rivers that cut across campus, and the USGS uses a collection of sensors to monitor a variety of statistics including height, volume, and rate of the flow, and also collects actual water samples, which are then tested for pesticides and other contaminants in the lab. The code that drives the collection of these statistics is a BASIC program running on a Sutron data logger. USGS scientists frequently modify these programs to change the way in which statistics and samples are collected to fix bugs, increase precision, or adjust for seasonal fluctuations.

This partnership is interesting for several reasons. First, the USGS's water monitoring system provides a real-world example of a distributed update problem. USGS scientists use multiple versions of their monitoring programs to collect data, but must then manually analyze and interpret this data. Having mechanisms for translating between these data sets—even just putting it into a unified schema—would significantly streamline data processing. Second, water monitoring connects closely to existing topics in the high school math and physics curriculum such as calculating rates of change, interpolating volume from several instantaneous data points, measuring percentage of various contaminants in a sample, etc. Third, introducing computer science through a course on water monitoring will likely be of greater interest to high school students, few of which will have an existing interest in programming languages or distributed systems.

A preliminary outline for the course is as follows:

| | |
|---|---|
| Day 1 | Introductions; overview of basic hydrology; review of required math and physics; programming introduction. |
| Day 2 | Extended programming assignment in the lab using an interpreter-based simulator; presentation by USGS hydrologist; discussion panel on careers in science; campus tour. |
| Data 3 | Field trip with USGS hydrologist to a monitoring station on an on-campus creek; closing discussion, wrap-up, and evaluation. |

To assist in developing the materials for this course, I will make use of two resources. First, I plan to work with undergraduates and a USGS scientist during the 2012-13 academic year to build an interpreter-based simulator and some simple data analysis tools. Second, I plan to work closely with teachers from the Science Leadership Academy (SLA) in Philadelphia. SLA is public technology-oriented magnet school that enrolls approximately 450 students, of which 57% are from underrepresented minority groups (49% African-American and 8% Latino) and 50% are from economically disadvantaged households [44]. I have an existing working relationship with a math teacher at SLA, and the computer science, physics, and biochemistry teachers at SLA have all expressed interest in participating in the project. The SLA principal has agreed to allow at least one staff member to attend the workshop to assist in teaching the course. I plan to work closely with the SLA teachers to develop the curriculum, paying special attention to drawing out connections to existing topics in the high school curriculum. I plan to visit Philadelphia several times during the school year to work with SLA teachers in person and to recruit SLA students to attend the first event. For these students, many of whom will be the first in their families to attend college, the chance to spend several days on a university campus should be a transformative experience.

I will assess the success of the project continuously. Before and after each iteration of the workshop, I will ask participants to complete surveys evaluating their concept of computer science, their interest in science, and their intent to pursue a career in computer science. I will compare the results of these surveys to assess the extent to which the desired outcomes of the course have been achieved. Later, in the second phase of the project, I will use the results of these evaluations to refine the most successful materials into a reusable package that other teachers can use and build on.

## 4.2 Course Development

Although the majority of my activities related to the education component of this proposal will be devoted to the high school outreach program just described, I also plan to develop a new graduate course at the intersection between programming languages and distributed systems. In the fall of 2011, I developed and taught a course on foundations of concurrency, focusing on classic formalisms such as CCS, CSP, and $\pi$-calculus, as well as a number of newer languages and tools. This upcoming year, I plan to develop a graduate level course on that will introduce students to current topics at the intersection of programming languages, formal methods, distributed systems, and networks. In addition to its obvious connection to my research agenda this course will fill an important void in the current offerings from the CS department—although the department has strengths in these areas, no current courses that I am aware of cuts across them. Because this project is inter-disciplinary, having such a course will be important for the research component as well—it will help new graduate students acquire the background in programming languages and distributed systems required to work in this area.

To attract a diverse collection of students, I plan to keep the course balanced evenly between the main areas, but focus slightly on the use of language-based abstractions and formal methods for enabling precise reasoning about distributed systems. These topics represent an area where language-based approaches have the promise of offering significant benefits over current practices, and where systems have a rich domain of problems that will emphasize the effectiveness of these approaches. In the first iteration of the course, I plan to run it as a closely-guided seminar, focusing more heavily on lectures and readings in the first half of the course and on projects in the second half of the semester. If possible, I will also bring in outside speakers from industry such as Yaron Minsky from Jane Street Capital, who gave a guest lecture in my course last fall, and has experienced the challenges of implemented distributed updates first hand [41]. For the course projects, I will have students develop a language or formal-methods-based system that helps make distributed programs easier to write, more reliable, or more efficient. Ideally, these projects will be of a high enough quality that they will spark new collaborations or lead to an eventual workshop or conference submissions.

In subsequent iterations of the course, I plan to incorporate programming projects based on the system described in this proposal, focusing more closely on the development of new lens languages, distributed update protocols, or applications. As the system matures, I also plan to use it as the main testbed for course projects.

### 4.3 Student Mentoring

I am a firm believer of the value of mentoring. I would not have ended up in a research career without the guidance of an early mentor who recruited me to an REU after my junior year of college, and who provided me with constant encouragement and advice as I navigated the early stages of graduate school. I will actively seek out ways to involve undergraduate researchers and under-represented minorities on the project, including applying for supplemental REU funds to involve undergraduates in research. I have a strong track record of successfully working with undergraduates on research. A student, with whom I have co-authored several papers recently won an Honorable Mention award in the CRA Outstanding Undergraduate Researcher competition.

## 5 Work Plan

To evaluate the research contributions of this project, the main threads of work described in this proposal will be integrated into a unified system with verification tools and applications. I plan to release selected components of the system separately—*e.g.*, the dynamic update system and lens language will also be of independent use. All of the software and programming infrastructure developed in the project will be released under an open-source license and made available on the web. I will also encourage use of the system by others and invite transfer of the ideas developed in this project to industry.

The following table summarizes the research and education plans described in this proposal. I expect each activity listed to run for approximately 12-24 months.

| Year | Research | Education |
|---|---|---|
| 1 | Update correctness (§3.1)<br>System design (§3.2)<br>Lens design (§3.1, 3.2) | Workshop #1<br>New graduate course on PL/Systems |
| 2 | System implementation (§3.2)<br>Lens implementation (§3.2)<br>Consistency design (§3.1,3.2) | Workshop #2 |
| 3 | Verification tool design (§3.3)<br>Consistency implementation (§3.2) | Workshop #3<br>Graduate course on PL/Systems |
| 4 | Optimization (§3.2)<br>Verification tool implementation (§3.3)<br>Application development (§3.4) | Workshop #4<br>Refine workshop materials |
| 5 | Application development (§3.4)<br>Software release (§5) | Workshop #5<br>Release workshop materials |