J. Nathan Foster: Research Statement

Some events from the recent headlines: A scientist retracts multiple papers after a bug in a homegrown program inverts several columns of experimental data, invalidating years of research.[1] A Canadian man discovers that he can access other people's passport information over the web by altering the link to his own data.[2] A glitch exchanging information with the Social Security Administration causes Medicare to accidentally issue $50 million in refund checks.[3] Operations for querying, transforming, and integrating data lie at the heart of a growing number of software systems. Unfortunately, these systems are difficult to implement correctly using current tools: not only are the operations complex, correctness itself often hinges on intricate properties of the data and the way they are handled.

The goal of my research is to develop tools that make it easy to build systems that are reliable, efficient, and maintainable. To that end, I am interested in designing programming languages with features like high-level syntax that elegantly expresses complicated transformations on data, precise type systems that track intricate correctness properties, and optimized implementations that make it possible to enjoy these benefits without sacrificing performance. Although my primary training is in programming languages, my approach to research is interdisciplinary: I seek out problems in areas where improved linguistic technology stands to provide substantial benefits. My dissertation on bidirectional programming languages proposes a novel solution to the classic view update problem from databases. I have also done work on data synchronization algorithms, data provenance, type systems, and incremental view maintenance. My future research plans center around developing trustworthy tools for manipulating data securely.

Bidirectional Programming Languages

My dissertation proposes bidirectional programming languages as an effective and elegant mechanism for defining updatable views. Views are usually discussed in the context of databases—in that setting, a view is the structure that results from evaluating a query, and an updatable view is one that can be modified, with the changes propagated back to the underlying source—but the need to edit data through a view also arises in many other areas of computing. There are numerous examples: data converters and synchronizers, parsers and pretty printers, picklers and unpicklers, structure editors, constraint maintainers for user interfaces, software model transformations, dynamic software updating, systems administration tools, etc.

Unfortunately, while the need for updatable views is ubiquitous, the technology for defining them is embarrassingly primitive. Most systems implement updatable views using two separate programs—one to compute the view and another to handle updates—a rudimentary design that is tedious to program, difficult to reason about, and a nightmare to maintain. My dissertation proposes a better alternative: programming languages in which both transformations can be described together. Every program in a bidirectional language can be run forwards and backwards, transforming sources to views in one direction, and conversely in the other. This eliminates the need to write—and maintain!—two separate programs, as well as the need to do any reasoning about their joint behavior: the language can be designed to guarantee correctness.

The contributions of my dissertation span several areas including semantic foundations, language design, and applications. The main ideas, which were developed in collaboration with other members of the Harmony Project, have been described in a series of publications [2, 7, 6, 10, 11] and implemented in Boomerang, an open-source bidirectional language for strings.

Foundations Intuitively, it should be clear that the transformations denoted by a bidirectional program should not be arbitrary functions—the two should be related. I have proposed a semantic space of well-behaved

[1]Greg Miller. A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science*, 314(5807):1856–1857, 2006.

[2]Kenyon Wallace. "Passport Applicant Finds Massive Privacy Breach." *Globe and Mail*, 12 April 2007.

[3]Robert Pear. "Medicare Refund Mixup Part of Larger Tangle." *New York Times*, 25 September 2006.

bidirectional transformations called lenses that captures an intuitive notion of correctness [7]. Formally, a lens consists of a pair of functions—the forward transformation *get* and the backward transformation *put*—that obey behavioral laws governing the handling of data on round trips. These laws, which are closely related to classical conditions on view update translators in databases, form the semantic underpinnings of Boomerang and have also been adopted by the designers of many other bidirectional languages.

I have also proposed two refinements of lenses that are important in practice. Quotient lenses only obey the lens laws modulo specified equivalence relations on the source and view [11]. This relaxation facilitates handling "inessential" details like whitespace that often occur in real-world data formats. Quasi-oblivious lenses ignore differences between sources related by a (different) specified equivalence in the reverse direction [2]. This condition ensures that updates involving reorderings are handled correctly.

Language Design The key challenge in the design of a lens language lies in finding syntax that is rich enough to express the transformations needed in applications and yet simple enough that correctness can be verified using simple and mechanical checks. My strategy for coping with this challenge in Boomerang is to divide the language into two layers: a core set of lens combinators embedded in a full-blown functional language. The typing rules of the combinators ensure the behavioral laws, while the functional infrastructure makes it possible to factor out repeated code into generic libraries and eliminates the need to program using combinators alone. Because the forward and reverse directions need to work well together, the design of specific lens primitives is very much a domain-specific task. Boomerang has primitives based on finite-state string transducers (plus a few additional primitives that go beyond this simple computational model but are important in practice—e.g., operators for reordering strings and for putting two lenses in sequence). I first became interested in strings because I wanted to explore the foundational issues surrounding the bidirectional manipulation of ordered data, and strings are the simplest ordered structures. But there are a lot of strings in the world—-textual databases, structured documents, scientific data, and even XML documents—and it has turned out to be quite convenient to be able to define lenses over these strings directly, without first parsing them into more structured representations.

Implementation and Applications Although much of my work on lenses is theoretical, I have also invested significant time building systems. I enjoy implementation work and consider it a critical component of research: it validates (or debunks) design choices made during the early phases of a project and it exposes critical engineering issues. For example, my experience using Boomerang to develop lenses for real-world data formats validated the overall design of the language but exposed several needs—for quotient lenses and for an efficient regular expression library—that would have remained hidden on paper. Having an implementation also led to an unexpected side benefit: lenses are now being used in industry. A developer at Red Hat Linux recently ported Boomerang's core primitives to C and incorporated them into Augeas, a tool for managing operating system configurations. Augeas contributors have developed lenses to compute views over a large number of `/etc` configuration files.

The main area where I have applied lenses is data synchronization. In that setting, lenses bridge the gap between heterogeneous replicas—e.g., to synchronize an XML address book with one formatted in vCard, we can use lenses to map between the original formats and views in a common format. In previous work as a part of the Harmony project, I designed a generic synchronization algorithm for reconciling changes to unordered trees [6]. I recently modified this algorithm to work with strings, extended it with a heuristic alignment strategy based on `diff3`, and implemented it in Boomerang. A novel feature of this algorithm is that its behavior—in particular, the recognition of conflicts—is driven by the type of the replicas. It also obeys a simple specification guaranteeing that the strings produced by the synchronizer are reasonable and well typed.

Other Work

I have had the opportunity to work on several other topics besides lenses during graduate school.

Data Provenance In collaboration with TJ Green and Val Tannen, I designed a framework for representing and querying annotated XML [5]. We showed how to decorate XML trees with annotations from a commutative semiring, defined a semantics for a large fragment of XQuery, and proved a correctness result, showing that our semantics commutes with applications of semiring homomorphisms. We then developed a number of applications based on annotated trees: XML with provenance, XML with repetitions, incomplete XML, probabilistic XML, and a novel scheme for controlling access to confidential data. In later work, Grigoris Karvounarakis and I studied uses of provenance in two systems with replicated data [8].

Generic Programming With Mary Fernández, Kathleen Fisher, Michael Greenberg, and Yitzhak Mandelbaum from AT&T Research, I extended PADS with an generic programming toolkit [4]. PADS is a declarative data description language. Given a description of a data format, the compiler generates a parser, pretty printer, and canonical in-memory representation for the data. We extended the compiler with a generic interface that programmers can use to build additional tools, and we implemented tools that produce XML and unordered trees as case studies. These make it possible to query PADS-described data as if it were XML and synchronize it using Harmony.

View Maintenance With Ravi Konuru, Jérôme Siméon, and Lionel Villard from IBM Research, I designed an incremental maintenance system for materialized XQuery views [9]. Given a source, a query, and a source update, the system translates the update through the query to obtain a corresponding view update. In many cases—e.g., when the source is large—the cost of calculating and applying this update is much cheaper than recomputing the view from scratch. We implemented a prototype of the system, and ran experiments to validate our approach.

Mechanized Mathematics A few years ago, a group of us at Penn and Cambridge wondered: how close are we to a world where every POPL paper is accompanied by an electronic appendix containing a machine-checked proof of correctness? To help measure progress toward this goal, we proposed a set of formalization benchmarks based on the metatheory of System F extended with subtyping [1]. Although the technical contributions of this work were not deep, it had a big impact within the programming languages community and brought attention to the important problem of representing terms with binders. In later work, Dimitrios Vytiniotis and I developed a mechanized proof of type safety for Featherweight Java in Isabelle/HOL [12].

Object-Oriented Type Systems With Kim Bruce, I designed an extension of Java featuring bounded polymorphism and a new type expression capturing the type of `this` [3]. We showed how these features provide solutions to the problems that arise with binary methods as well as the so-called "expression problem". We implemented a compiler for the language and proved the soundness of a core fragment of the type system as an extension of Featherweight Java.

Future Work

With my next steps, I plan to continue doing research in the general area of programming languages for data processing, focusing my specific attention on issues in security. I believe that the time is ripe for data processing languages designed with security in mind—many systems already manipulate secure data, but few languages provide any tools for establishing that they do so correctly. Because of this mismatch, relatively minor programming bugs can lead to catastrophic errors like the massive privacy breach involving passport data described at the start of this statement. Several promising ideas for equipping languages with features designed to help programmers build secure systems are beginning to emerge from the programming languages and databases communities. I hope to use my experience in both areas to help bring these ideas to fruition in the next generation of secure data processing languages.

Updatable Security Views Security views are a widely-used mechanism for controlling access to confidential data in databases. By only allowing programmers to pose queries over views that do not contain confidential data, database administrators ensure that it cannot be leaked, even if the query has bugs (or the programmer is malicious). However, security views are not usually updatable and for good reason! Propagating updates made by untrusted users can change the source—including the confidential data hidden by the view—in arbitrary and irreversible ways. I am interested in enhancing lenses with capabilities for tracking integrity of data. The idea is to provide a way to ensure that confidential data in the source will not be disturbed by the *put* function. These lenses will provide a powerful mechanism for enforcing security barriers in many applications. For example, they could be used to hide private appointments in an electronic calendar while allowing a colleague to make changes to the public ones, or to redact secret data in a secure document while allowing users with lower levels of clearance to edit the regraded version. I believe they have the potential to have enormous impact.

Information Flow for Regular Types There is a large body of work on languages with information-flow type systems. In these languages, types carry annotations representing security properties of data—e.g., its level of confidentiality or integrity—and the static type system tracks flows of information from input to output. The end result is a non-interference theorem guaranteeing that the low-security parts of the output do not depend on the high-security parts of the input. There is also a large body of work on languages with regular types. These types, which are based on finite-state automata, are a very expressive formalism for describing data at a high level of precision. They enjoy many desirable properties—closure under set-theoretic operations and efficient algorithms for deciding membership, inclusion, and emptiness—and are widely used in languages for data processing. Somewhat surprisingly, no one has combined these two threads. I am interested in developing advanced type systems with both regular types and information flow. In these systems it will be possible to specify security policies as annotated regular types and use the static type checker to mechanically verify that programs respect these policies.

Hybrid Type Checking Type systems are fundamental mechanisms for establishing safety properties in programming languages. In the research community, there is a trend toward more and more precise type systems that express detailed assertions about the behavior of programs and the data they manipulate—e.g. the information-flow type system just discussed. However, precision is a double-edged sword: languages with precise types can be cumbersome for programmers to use and difficult for language designers to implement. One idea for realizing some of their benefits while avoiding their pitfalls is hybrid type checking. Languages with hybrid checking establish safety using a combination of static and dynamic techniques—e.g., straightforward static analyses to rule out gross errors and dynamic checks to detect violations of more intricate properties. Hybrid systems are already an active area of research in the programming languages community. I hope to join this conversation, using the specific issues related to security type systems to ground my investigation.

Language Support for Audit Many applications generate and record logs of their behavior. For example, data synchronizers generate logs recording the actions performed to the replicas along each path. When errors occur, these logs provide critical evidence for determining the cause of the error as well as the actions needed to return the system to a good state. Intuitively, it is obvious that some audit logs are better than others. For example compact logs are preferable to snapshots of the entire system state, but logs should not be so small that errors cannot be detected or actions rolled back. I am interested in developing a foundational theory of correct audit that will make it possible to phrase and evaluate comparisons between different auditing strategies. I also plan to explore formal connections between these audit artifacts and provenance metadata, which is used for similar purposes in many applications.

Model Transformations This final topic lies somewhat further afield. In software engineering, a model transformation expresses a relationship between software models—e.g., between a UML diagram and the corresponding Java implementation, or between a single model and its refactoring. It turns out that technologies like lenses, data synchronizers, and incremental maintenance systems, are all useful for maintaining consistency in systems based on model transformations. I am interested in using my position as an informed outside observer to make connections between the areas of programming languages and databases and model transformations. I recently organized a Dagstuhl-style seminar on bidirectional transformations with Krzyzstof Cznarecki from Waterloo and Zhenjiang Hu from the National Institute of Informatics in Japan that brought together a diverse mix researchers from these areas to discuss problems and share solutions. The three of us have been invited to publish a survey paper at next year's ICMT conference. We plan to use this paper to identify key problems and establish common terminology. Eventually, we also hope to develop a set of canonical benchmark transformations for comparing bidirectional formalisms and their associated correctness properties.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLS), Oxford, UK*, August 2005.

[2] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, CA*, pages 407–419, January 2008.

[3] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *European Conference on Object-Oriented Programming (ECOOP), Oslo, Norway*, volume 3086 of *Lecture Notes in Computer Science*, pages 389–413. Springer-Verlag, June 2004.

[4] Mary Fernandez, Kathleen Fisher, J. Nathan Foster, Michael Greenberg, and Yitzhak Mandelbaum. A generic programming toolkit for PADS/ML: First-class upgrades for third-party developers. In *Symposium on Practical Aspects of Declarative Languages (PADL), San Francisco, CA*, pages 133–149, January 2008.

[5] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: Queries and provenance. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Vancouver, BC*, pages 271–280, June 2008.

[6] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4), June 2007. Short version in DBPL '05.

[7] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007. Short version in POPL '05.

[8] J. Nathan Foster and Grigoris Karvounarakis. Provenance and data synchronization. *IEEE Data Engineering Bulletin*, 30(4):13–21, December 2007. Invited paper for special issue on provenance.

[9] J. Nathan Foster, Ravi Konuru, Jerome Simeon, and Lionel Villard. An algebraic approach to XQuery view maintenance. In *ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X), San Francisco, CA*, page 31, January 2008.

[10] J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X), Nice, France*, pages 80–90, January 2007.

[11] J. Nathan Foster, Alexandre Pilkiewcz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, BC*, pages 383–395, September 2008.

[12] J. Nathan Foster and Dimitrios Vytiniotis. A theory of Featherweight Java in Isabelle/HOL. *Archive of Formal Proofs*, April 2006.