Dissertation Proposal

Computer & Information Science
http://www.cis.upenn.edu

Bidirectional Programming Languages

J. Nathan Foster
jnfoster@cis.upenn.edu

Committee: Zachary Ives
Val Tannen
Philip Wadler
Stephan Zdancewic (chair)

Adviser: Benjamin C. Pierce

August 26, 2008

Abstract

The need to modify source data through a view arises in a large number of applications across many areas of computing. Unfortunately, programming language support for defining updateable views is quite primitive—in most systems, views are defined using one program to compute the view and another to handle updates, a design that is both error-prone and difficult to maintain. In this project, I propose bidirectional programming languages as an effective mechanism for describing updateable views. In a bidirectional language, every program denotes two functions—one that maps a source to a view, and another that combines a modified view with the original source and produces a correspondingly updated source. Starting from the foundations, I develop a semantic space of bidirectional transformations called lenses whose behavior obeys natural conditions capturing intuitive notions of correctness. I then instantiate this general framework with a specific lens language for strings called Boomerang. I develop several extensions to Boomerang: one addressing problems that come up with reorderable data, another for handling inessential data, and a third for tracking some security properties that are important for defining updateable views over data sources containing confidential information. Finally, using examples, I demonstrate that Boomerang can be used to build lenses for transforming a variety of data formats that are used in practice.

Acknowledgements

Occasionally, dissertation projects go like this: the student matriculates, the adviser suggests a topic, the student descends into the library, and resurfaces n years later with a draft. My project, thankfully, has not. The work described in this proposal has emerged from an extended, close collaboration with Benjamin Pierce and Alan Schmitt, with additional major contributions by Aaron Bohannon, Michael Greenberg, *stagier extraordinaire* Alexandre Pilkiewicz, and Steve Zdancewic. While the formulation of this proposal (and any errors it contains) is mine, the systems it describe are very much a product of this team. In particular, portions of this proposal draw on material from papers published with these co-authors: (Bohannon et al., 2008; Foster et al., 2007a,b, 2008b). I have been very lucky to count these talented individuals as my colleagues and friends, and I gratefully acknowledge their deep contributions to this project.

This work has been supported by an NSF Graduate Research Fellowship, and NSF grants CPA-0429836 *Harmony: The Art of Reconciliation*, IIS-0534592 *Linguistic Foundations for XML View Update*, and CT-0716469 *Manifest Security*.

Contents

"Never look back unless you are planning to go that way."

—Henry David Thoreau

1 Introduction

This dissertation proposes bidirectional programing languages as an effective and elegant means of specifying updateable views. Views are usually discussed in the context of databases—a *view* is the structure that results from evaluating a query over some source data, and an *updateable view* is one that can be modified, with the changes propagated back to corresponding updates on the underlying source. However, the need to edit through a view arises in a host of applications across many diverse areas of computing. There are numerous examples:

Data Converters and Synchronizers: views bridge the gap between heterogeneous replicas (Brabrand et al., 2008; Kawanaka and Hosoya, 2006; Foster et al., 2007a).

Data Management: besides traditional applications to relational systems (Bancilhon and Spyratos, 1981; Dayal and Bernstein, 1982), views have been used in systems for data exchange (Miller et al., 2001) and schema evolution (Berdaguer et al., 2007).

Software Engineering: model transformations compute views over formal software models (Schürr, 1995; Stevens, 2007; Xiong et al., 2007).

Serialization: an unpickler computes a view that maps binary representations of data on the file system to structured objects in memory (the corresponding pickler handles updates) (Fisher and Gruber, 2005; Eger, 2005).

Systems Administration: views have been used in a system for managing operating system configurations (Lutterkort, 2008).

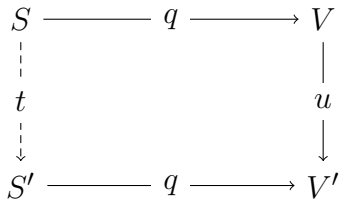
Programming Languages: parsers and pretty printers manipulate views of program sources; in embedded interpreters, run-time values in the object language can be computed as views over corresponding values in the host language (Benton, 2005; Ramsey, 2003).

User Interfaces: structure editors use views to provide convenient editing interfaces for complicated documents (Hu et al., 2008); views have also been used in general-purpose user interface frameworks (Meertens, 1998; Greenberg and Krishnamurthi, 2007).

Unfortunately, although the need for updateable views is ubiquitous, linguistic technology for defining them is embarrassingly primitive. The views in most of the applications listed above are defined using two separate functions—one that computes the view from the source, and another that handles updates—a rudimentary design that is tedious to construct, difficult to

reason about, and a nightmare to maintain. Addressing these limitations is the main motivation for this project.

Let us start by exploring some of the issues surrounding updateable views in relational databases, the context where they were first studied. (The same fundamental issues arise in the other areas listed above.) Suppose that S is a source database, q a query, and $V = q(S)$ the view that results from evaluating q on S. The *view update problem* is as follows: given an update u to the view calculate a source update t (the "translation" of u) such that the following diagram commutes:



View update is a difficult problem because, in general, the update u does not uniquely determine a source update t. For example, when q is not injective, then there are some view updates that have several corresponding source updates. It is possible to impose additional constraints to guide the choice of an update—e.g., requiring that the source update have minimal "side effects"—but for sufficiently expressive query languages, calculating updates satisfying these additional constraints is intractable (Buneman et al., 2002). Even worse, if q is not surjective, then some view updates describe structures that lie outside of the range of q entirely! The system can refuse to propagate these updates—there is no source update it could choose—but doing so breaks the abstraction boundary between the source and view: it adds hidden constraints on how the view may be updated that are only revealed by considering the action of the update on the source.

Because of these problems, views in relational systems (except for ones defined by very simple queries) are generally read-only. To define an updateable view, database programmers fall back on a variant of the rudimentary mechanism described above: they define a separate procedure called a *trigger* that the system executes every time the view is changed. Triggers are arbitrary programs, and so can be written to implement any view update policy the programmer could want, but they are not a robust solution. For one thing, checking that a trigger correctly propagates updates requires intricate, manual reasoning about the way it works in tandem with the query. Moreover, the triggers and queries both embody parts of the source and view schemas, and so will be redundant and difficult to maintain as the schemas evolve.

A better design is to have a language in which the query and the function that handles updates are described together. *Bidirectional programming language* are organized around this idea: every program in a bidirectional language can be read from left to right as a query mapping sources to views, and from right to left as a function mapping modified views back to updated sources. This design eliminates the need to write—and maintain!—two separate programs, as well as the need to do any manual reasoning about the joint behavior of the two transformations: the type system of the language can be designed to guarantee correctness.

The key design challenge lies in balancing the tradeoffs between syntax that is rich enough to express the queries and update policies demanded by applications, and yet simple enough that correctness properties can be formulated as straightforward, compositional, and mechanizable checks.

1.1 Goal and Contributions

The ultimate goal of this dissertation project is to demonstrate that bidirectional languages are an effective tool for defining updateable views. Its contributions will span three main areas: a general semantic foundation for well-behaved bidirectional transformations, a specific bidirectional programming language for processing string data, and a collection of applications demonstrating that the framework and language are both practical.

Many of the results that will eventually go into the dissertation have already been thoroughly described in refereed journal and conference papers Foster et al. (2007b); Bohannon et al. (2008); Foster et al. (2008b, 2007a). I do not make any attempt to redescribe those results in depth in this document. (The full technicalities can be found in those papers; of course, the final dissertation will be a self-contained document with full, formal definitions and accompanying correctness proofs.) Instead, I focus on the part of the work that is unfinished: a design for building trustworthy updateable security views using lenses. However, to give the reader a sense of the overall shape that the final dissertation will take, in the rest of this introductory section, I briefly sketch the other contributions.

Semantic Foundations

The two main advantages of using a bidirectional language are parsimony—two transformations described by a single program—and the correctness guarantees it provides about the way the transformations work together. Parsimony is automatic for bidirectional languages. However, before we can even talk about correctness, we need a characterization of the transformations that have the properties we want. Thus, the first component of this work is a semantic space of *well-behaved bidirectional transformations* called *lenses*. These structures serve as the semantic foundation for the rest of the project.

A lens mapping between a set S of source structures and a set V of views has three components called *get*, *put*, and *create*. The *get* component plays the role of the query and is a total function from S to V. The *put* and *create* components handle updates. The *put* function takes an updated V and the original S and calculates a new S that reflects the changes made to V. The *create* function handles cases where there is no source to use as the "original" S. It calculates a new S directly from its V argument, filling in any information discarded by the query with defaults.

Every lens obeys behavioral laws ensuring that updates to views are propagated back to sources correctly. Additionally, there are also two refinements of lenses that turn out to be important in practice. *Quotient lenses* relax the behavioral laws to versions that are only required to hold modulo specified equivalence relations on the source and view. This facilitates han-

dling "inessential" data such as whitespace, escaping conventions, and duplicated data using lenses. *Quasi-oblivious lenses* strengthen the behavioral laws by requiring that the *put* function ignore differences between sources related by a (different) specified equivalence relation. This extension is useful for guaranteeing that updates to ordered data are handled reasonably.

Boomerang: A Bidirectional Programming Language

The second component of this work is *Boomerang*, a programming language for writing lenses on strings. I choose to work with strings, rather than a richer data model such as trees or complex values, primarily as a matter of research strategy: strings are simple structures that still expose many fundamental issues in bidirectional programming including ones concerning order. But there is also a lot of string data in the world—textual databases, ad-hoc formats, structured documents, scientific data, simple XML, and microformats—and it is useful to have to be able to manipulate it directly, without first parsing it into (and pretty print it from) more complicated representations.

Boomerang is built around a set of core combinators based on finite state transducers (union, concatenation, Kleene star), some generic operators (identity, constant, sequential composition) and some special constructs for dealing with ignorable (canonizers, quotienting) and ordered (chunks, keys) data. These combinators are a quite natural formalism for expressing many transformations on strings of practical interest, but even so, programming with combinators alone would be quite tedious. So instead, in Boomerang, the combinators are embedded in a full-blown functional programming language with a rich collection of features: first-class functions, user-defined data types, polymorphism, dependent types, refinement types, simple modules, and in-line unit tests. Using this infrastructure, it is easy to build large lenses quickly, to factor out common programming idioms into generic libraries, and to write lens programs at a level of abstraction that is appropriate for the application at hand.

Applications

The final component of this project will be a collection of end-to-end applications built using lenses written in Boomerang. These applications will validate the various design choices made in the design of the semantic framework and the language, and will serve as case studies of programming with lenses.

I have already built an implementation of the Boomerang system and have used it to build a number of prototype applications including data converters and synchronizers for a few interesting real-world data formats. I believe that the ideas in this project have the potential to have significant practical impact, so I plan to spend a some of my remaining time carrying each of these initial steps a little further. However, I plan to spend the bulk of my remaining time studying some novel extensions for building *updateable security views* using lenses.

Security views are ones that are designed to control access to confidential data. As an example, consider a program that generates an online directory from an employee record database. The full database contains all the information that goes into the directory—names, email ad-

dresses, and phone numbers—and a great deal of information that does not—salaries, performance evaluations, Social Security numbers, etc. If the program has access to the full database, then a programming error (or a program compromised by a malicious hacker) can result in private data being published online. If instead, the program accesses the database via a view that only exposes the public parts and hides the private parts, then the sensitive information cannot be leaked, no matter how the program behaves.

Security views are already widely used, but current systems do not address two important issues. First, they do not provide formal tools for establishing that the private data in the source is not leaked to the view. For simple queries, it may be obvious whether a piece of the source is leaked by the query. But when the query is complicated, information can flow from source to view in subtle ways. It is therefore important to have tools for establishing end-to-end guarantees about the confidentiality of source data. Second, security views are usually not updateable, and for good reason: propagating changes made by an untrusted user back to a trusted source can modify the source in arbitrary and irreversible ways. Thus, if untrusted users are going to modify source data through a view, mechanisms for tracking the integrity of source data are needed.

I propose enhancements to lenses that addresses both of these problems. For confidentiality, I propose a refinement of Boomerang's type system to track confidentiality using an information-flow analysis. Somewhat surprisingly, although there is a large body of work on information-flow type systems for general-purpose programming languages, very little work has been done on information-flow for type systems with regular types. Thus, this component of the work should be of interest for unidirectional languages as well. For integrity, I propose a generalized semantics for *put* functions that generates logs recording the changes induced on the source. The goal is to output a log with sufficient information for a trusted user to audit the final result, incorporating or rejecting changes as they see fit.

To demonstrate how all of this works, I will develop a collection of lenses for managing security views of MediaWiki documents. These lenses also serve as the main running example for this document.

1.2 Outline

The rest of this proposal is organized as follows. Section 2 introduces Boomerang using a simple transformation for secure documents, and defines the semantic space of basic lenses. Section 3 lenses discusses some problems related to ordered data, and shows how they can be resolved in Boomerang using dictionary lenses. Section 4 develops the example further by illustrating how ignorable information can be handled using quotient lenses. A final extension to the example making it more realistic is described in Section 5. Section 6 discusses the properties of lenses that are needed in security settings, and describes preliminary ideas for extending the lens framework to reason about confidentiality and integrity. Section 7 discusses related work. I conclude in Section 8 with a brief overview to current implementation status and a roadmap for the rest of the project.

2 Basic Lenses

In this first section, I describe a lens that computes a very simple security view of a document. This lens serves both to illustrate the main features of Boomerang, and to introduce the running example used in the rest of this document.

Intelligence organizations often use security clearances to control access to documents containing sensitive information. Every individual and every document in the organization is assigned a clearance—e.g., public, confidential, secret, top secret, etc.—and individuals are only allowed to access a particular document if they have sufficient clearance. This scheme is effective, but a bit coarse: information is classified at the granularity of documents, so individuals who may have clearance to access part of a document are instead denied access if it also contains information classified at a higher clearance. The usual workaround is have a trusted individual *regrade* the document by erasing or redacting the highly classified portions, leaving behind a residual document that can be reclassified at a lower clearance.

In current practice, regrading is a manual process, so there is significant interest in developing document models with support for automatic regrading. *Tearline documents* are a very simple model that has been proposed for documents in MediaWiki format Potoczny-Jones (2008). In a tearline source, there is a clear boundary that separates confidential data (at the top of the document) from public data (at the bottom). (This model can be generalized to more levels of clearance by introducing more boundaries; I will work with just two levels to keep the example simple.) To compute a view from a source, the system "tears" off the public portion at the boundary. This view, which does not contain confidential information, can then be published freely.

The first lens I will show constructs updateable tearline views. To keep the example simple, I will work with an extremely simple subset of MediaWiki: a document consists of a single section; a section has a heading of the form `==Title==` and is followed by a sequence of paragraphs or lists lines; a paragraph consists of blocks of ordinary text; and list items are indicated by lines beginning with "`*`". Also, I exclude the special characters `*`, `%` and `=` from the strings used as section headings, paragraph bodies, and list items. Eliminating these restrictions and scaling up to full-blown MediaWiki should be relatively straightforward. To mark the boundary between the public and confidential sections, I will use the special string

```
%----- TEARLINE -----%
```

(any other distinguished string would work equally well).

Before examining the definition of the lens in Boomerang, let's see how its works on an example. Its *get* function maps a source document

```
==Chefs==
Julia Child worked for the Office of Strategic Services
during World War II.
%----- TEARLINE -----%
* Julia Child
```

to a view

```
==Chefs==
* Julia Child
```

by copying the heading, deleting the confidential paragraph and separator that follow, and copying the final list. Conversely, the *put* function takes an updated view—here I have added Jacques Pépin to the list of chefs

```
==Chefs==
* Julia Child
* Jacques Pepin
```

and combines it with the original source, yielding an updated source that also contains the new list item:

```
==Chefs==
Julia Child worked for the Office of Strategic Services
during World War II.
%----- TEARLINE -----%
* Julia Child
* Jacques Pepin
```

The definition of the lens that does both of these these transformations in Boomerang is as follows. It mentions some identifiers (in capital letters) denoting regular expressions, which I define below:

```
let tearline_section : lens =
  copy HEADER .
  del BODY .
  del TEARLINE .
  copy BODY
```

The primitive lenses `copy` and `del` each take a regular expression as an argument. The *get* component of the `copy` lens copies every string belonging to (the language denoted by) its argument to the view, and conversely in the *put* direction. The `del` lens deletes every string belonging to its argument in the *get* direction, and restores it the *put* direction. The concatenation operator on lenses (`.`) builds a lens that handles large strings from two lenses that handle smaller ones. Its *get* and *put* functions work by splitting the input strings in two, processing these substrings using the corresponding component of the sublenses, and concatenating the final results. The typing rule for concatenation ensures that the strings can be split into substrings in just one way. Putting all these pieces together, we see that the *get* component of the `tearline_section` lens processes a whole section by copying the substring matching `HEADER`, deleting the substrings matching `BODY` and `TEARLINE` that follow, and copying the following `BODY` to the view. The *put* function copies substrings matching `HEADER` and `BODY` from the view and splices in the middle `BODY` and `TEARLINE` from the source.

For completeness, here are the regular expressions used to define `tearline_section`:

```
let INNER_CHAR : regexp = [^\n%*=]
let OUTER_CHAR : regexp = INNER_CHAR - [ ]
let TEXT : regexp = OUTER_CHAR . (INNER_CHAR* . OUTER_CHAR)?
let LINE : regexp = TEXT . NL
let PARAGRAPH : regexp = LINE+
let LIST : regexp = ("* " . LINE)+
let HEADER : regexp = "==" . TEXT . "=="
let BODY : regexp = (NL . (PARAGRAPH | LIST))*
let TEARLINE : regexp = "%----- TEARLINE -----%"
```

The notation for character sets (`[^\n%=*] and [ ]`), repetitions (`*`, `+`, `?`, and `{2}`), and union (`|`) are all standard. Concatenation uses the same symbol as for lenses (`.`). In fact, all of the regular operators are overloaded in Boomerang and may be used to combine regular expressions, lenses, and strings; the type checker automatically promotes strings to the corresponding singleton regular expression, and regular expressions to instances of the `copy` lens as necessary, as in the concatenation of `"* "` and `LINE` in `LIST`.

2.1 Foundations of Basic Lenses

It is not difficult to see that the *get* and *put* components of `tearline_section` work well together. But more generally, we need precise criteria for deciding whether a given pair of functions—a bidirectional transformation—defines a correct updateable view. To this end, I present in this section a semantic space of well-behaved bidirectional transformations called basic lenses.

Let S be a set of sources and V a set of views. A *basic lens* l mapping between S and V of views comprises three functions

$$\begin{aligned} l.\mathit{get} &\in S \to V \\ l.\mathit{put} &\in V \to S \to V \\ l.\mathit{create} &\in V \to S \end{aligned}$$

obeying the following "round-tripping" laws for every $s \in S$ and $v \in V$:

$$l.\mathit{put}\ (l.\mathit{get}\ s)\ s = s \qquad \text{(GETPUT)}$$

$$l.\mathit{get}\ (l.\mathit{put}\ v\ s) = v \qquad \text{(PUTGET)}$$

$$l.\mathit{get}\ (l.\mathit{create}\ v) = v \qquad \text{(CREATEGET)}$$

These laws are closely related to the conditions on view update translators in databases discussed earlier. The GETPUT law stipulates that the *put* function must restore the original source exactly when its arguments are a view and a source that *get* maps to the very same view. It embodies a simple version of the "minimal side effects" condition, for the special case where the update is a no-op. (Imposing the general condition seems to require fixing an update language, and lenses are designed to be agnostic to the way that updates are expressed; see below.)

The PUTGET and CREATEGET laws stipulate that *put* and *create* must propagate all of the information in the updated view to the new source. They ensure that the new source is one that maps back to the updated view via *get*. The set of basic lenses mapping between S and V is written $S \Longleftrightarrow V$.

As an aside, returning to our `tearline_section` lens, we can revise its declaration to assert that it has the lens type we intend:

```
let SOURCE : regexp = HEADER . BODY . TEARLINE . BODY
let VIEW : regexp = HEADER . BODY
let tearline_section : (lens in SOURCE <-> VIEW) =
  copy HEADER .
  del BODY .
  del TEARLINE .
  copy BODY
```

The Boomerang type checker verifies that the lens has the type we have written down. For type system experts: Boomerang's type system has both dependent and refinement types; programs are checked using a hybrid approach—the static checker performs a coarse typing analysis and inserts checks that verify the precise conditions at run-time (Flanagan, 2006; Wadler and Findler, 2007).

The lens laws are not intended to serve as a complete specification of the correct bidirectional manipulation of data, but as a loose guide on the design of lens primitives. Indeed, they fall far short of that mark—e.g., when supplied with a source s and a view v with $v \neq \mathit{get}\ s$, the *put* function is free to choose any source s' such that $\mathit{get}\ s' = v$. Rather, they specify the results that transformations must produce in a few cases where the correct behavior is clear. In this spirit, I have used the lens laws many times to rule out bogus candidate lens primitives.

Although lenses are similar to view update translators in databases, there are a few key differences worth noting. First, in databases views are *virtual* structures that are not represented by any physical table. Thus, modifications to the view can *only* be handled by translating the update back to the source. In contrast, the *put* and *create* components of a lens manipulate whole view states—i.e., in database jargon, the views are *materialized*. Both of these approaches are reasonable and each has advantages. Systems that translate operations have more precise information about the nature of the update, but require tighter integration with applications. Conversely, systems that manipulate view states are agnostic to the way that updates are expressed, and so only require loose coupling to applications. Because I am interested in deploying lenses in a variety of settings, I choose the state-based approach. (An investigation of operation-based lenses is an excellent topic for future research.)

Another difference is that the *put* component of a lens is a total function. Totality, in combination with the lens laws, guarantees that a lens is capable of doing something reasonable with any view and any source. This is a very strong requirement that rules out many transformations and has therefore not been imposed in databases or in most other bidirectional languages. In those systems updates may fail—a design that is maximally flexible since the decision about whether or not to propagate a particular update can be made dynamically.

Unfortunately, it makes views "leaky" abstractions of the source because the semantics of updating a view—in particular, whether it succeeds or fails—depends on properties of the source. While this may be reasonable in some interactive applications, where the user can be provided with immediate feedback on illegal updates, in the applications where I am interested in using lenses—data synchronizers, which are often run in unsupervised environments, and security views, whose very purpose is to hide confidential parts of the source—it is critical that views be robust abstractions that encapsulate the source completely. Therefore, I insist that the *put* components of lenses be total functions.

3 Dictionary Lenses

So far, the tearline lens only handles a single section. In this section, I will scale it up to a lens that handles whole documents. This lens illustrates the treatment of ordered data in Boomerang.

As a first attempt, we can try just iterating the `tearline_section` lens using the Kleene star operator:

```
let tearline : (lens in SOURCE* <-> VIEW* ) =
  tearline_section*
```

Kleene star generalizes lens concatenation in the obvious way. For example, the *get* function of `tearline` maps source documents containing multiple sections, such as the following

```
==Chefs==
Julia Child worked for the Office of Strategic Services
during World War II.
%----- TEARLINE -----%
* Julia Child
==Supreme Court Justices==
Arthur Goldberg worked for the Secret Intelligence Branch.
%----- TEARLINE -----%
* Arthur Goldberg
```

to views where the confidential region in *each* section is deleted:

```
==Chefs==
* Julia Child
==Supreme Court Justices==
* Arthur Goldberg
```

The *put* function takes an updated view

```
==Chefs==
* Julia Child
* Jacques Pepin
==Supreme Court Justices==
* Felix Frankfurter
* Arthur Goldberg
```

and combines it with the original source, splicing in the confidential portion to each section

```
==Chefs==
Julia Child worked for the Office of Strategic Services
during World War II.
%----- TEARLINE -----%
* Julia Child
* Jacques Pepin
==Supreme Court Justices==
Arthur Goldberg worked for the Secret Intelligence Branch.
%----- TEARLINE -----%
* Felix Frankfurter
* Arthur Goldberg
```

as expected.

Unfortunately, although *put* works correctly for these inputs, there are others on which it behaves rather poorly. If, for example, in addition to adding the new list items we swap the order of sections in the view

```
==Supreme Court Justices==
* Felix Frankfurter
* Arthur Goldberg
==Chefs==
* Julia Child
* Jacques Pepin
```

then the *put* function produces a new source in which the confidential regions are spliced in to the wrong sections!

```
==Supreme Court Justices==
Julia Child worked for the Office of Strategic Services
during World War II.
%----- TEARLINE -----%
* Felix Frankfurter
* Arthur Goldberg
==Chefs==
Arthur Goldberg worked for the Secret Intelligence Branch.
%----- TEARLINE -----%
* Julia Child
* Jacques Pepin
```

The root of this problem is the Kleene-star operator, whose *put* function splits the source and view into substrings and invokes the *put* function of the lens being iterated on corresponding (by position!) substrings from the source and view. If the update to the view string breaks the positional association between sections in the source and the view, then the source document it produce will be mangled. This is a show-stopper for our secure document application; we cannot have a lens that mangles documents in this way.

Boomerang provides constructs specifically designed to deal with this problem using an extension of lenses called *dictionary lenses*. The rough idea is that programmers should identify *chunks* of the source and how to compute a *key* for each chunk. These notions induce an association between chunks in the source and substrings of the view, and this association rather than the positional one is used by the *put* function to align the source and view. Here is a dictionary lens that has the behavior we want:

```
let tearline_section : (lens in SOURCE <-> VIEW) =
  key HEADER .
  del (BODY . TEARLINE) .
  copy BODY
let tearline : (lens in SOURCE* <-> VIEW* ) =
  <tearline_section>*
```

Compared to the basic lens version, the first thing to notice is that the occurrence of `tearline_section` in the definition of `tearline` is enclosed in angle brackets. This syntax indicates that the substrings processed by `tearline_section` should be treated as reorderable chunks. The second difference is that the first occurrence of `copy` at the beginning of `tearline_section` has been replaced by `key`. The `key` lens has *get* and *put* components that behave exactly the same as the corresponding components of `copy`, but `key` additionally specifies that the view should be used as the key of the chunk in which it appears. In this example, the key of each section is its heading.

Operationally, the *put* function of a dictionary lens does its work in two phases. The first phase parses the entire source string into a dictionary structure in which chunks are organized by key. In the second phase, the *put* function uses this dictionary (rather than the original source) to locate the missing information for each part of the view. The end effect is that the *put* function aligns sections in the source and view by key rather than position. For example, on the same inputs as before, it yields

```
==Supreme Court Justices==
Arthur Goldberg worked for the Secret Intelligence Branch.
%----- TEARLINE -----%
* Felix Frankfurter
* Arthur Goldberg
==Chefs==
Julia Child worked for the Office of Strategic Services
during World War II.
%----- TEARLINE -----%
* Julia Child
* Jacques Pepin
```

as expected.

Boomerang also supports an extension where keys are matched approximately, using a greedy heuristic to align keys in the view and dictionary by minimizing edit distance. This extension to "fuzzy" alignment addresses the problem of updating views where keys have changed, and also facilitates using dictionary lenses with unstructured text when there is no obvious key

to use and we want to use the full contents of the view as its key. We will see an example of fuzzy alignment in Section 5.

3.1 Foundations of Dictionary Lenses

The second version of the `tearline` lens shows that dictionary lenses can be used to build updateable views that handle updates involving reorderings correctly. Unfortunately, at the level of semantics, the basic lens framework does not provide criteria for distinguishing the first version of `tearline`, which mangles sources, from the second version, which is correct. Both are well-behaved as basic lenses. In the rest of this section, I define a refined space of *quasi-oblivious* lenses that makes such distinctions possible.

Suppose that l is a basic lens mapping between S and V and let $\sim$ be an equivalence relation on S. Then l is a quasi-oblivious lens with respect to $\sim$ if its *put* function obeys the following law:

$$\frac{s \sim s'}{l.\mathit{put}\ v\ s = l.\mathit{put}\ v\ s'} \qquad \text{(EQUIVPUT)}$$

stipulating that the *put* function must ignore differences between sources related by $\sim$.

One way to understand its effect of the EQUIVPUT law is to notice how it extends the range of situations to which the GETPUT law applies. By itself, GETPUT only constrains the behavior of *put* on the exact view computed from the source; with EQUIVPUT, it must behave the same results on all equivalent sources. In particular, if the equivalence is one that relates source strings up to stable reorderings of chunks (i.e., a reordering that does not change the relative order of chunks having the same key) then PUTEQUIV forces the *put* function to handle updates that reorder the view by doing a corresponding reordering on the source.

Like the basic laws, quasi-obliviousness does not lead to a complete specification of the correct handling of ordering in bidirectional languages—in particular, it says nothing about what must happen when the update changes keys. But, it is still very useful as a constraint on the design of lens primitives. Indeed, dictionary lenses, whose *put* functions that operate on dictionaries obtained by parsing the source, are designed to be quasi-oblivious with respect to the reordering equivalence described above.

It is also interesting to consider lenses that are quasi-oblivious with respect to other relations. Every basic lens is trivially a quasi-oblivious lens with respect to equality, the finest equivalence relation. A lens l is *bijective* if its *put* function is completely oblivious—i.e., if

$$l.\mathit{put}\ v\ s = l.\mathit{put}.\ v; s'$$

for every $v \in V$ and $s, s' \in S$. Bijectiveness can also be characterized using the coarsest equivalence relation for which l satisfies EQUIVPUT, called $\sim_{max}$: $c \sim_{max} c'$ iff $\forall v.\, \mathit{put}\ v\ s = \mathit{put}\ v\ s'$. A lens is bijective iff $\sim_{max}$ is the total relation on S. Another important special case of basic lenses also has a characterization using $\sim_{max}$. A lens l is called *very well behaved* if the effect of two *put*s in sequence just has the effect of the second—i.e., if

$$l.\mathit{put}\ v\ (l.\mathit{put}\ v'\ s) = l.\mathit{put}\ v\ s$$

for every $v, v' \in V$ and $s \in S$. Very well behavedness is a strong condition—it turns out to be equivalent to the classical notion of view update translation under "constant complement" (Bancilhon and Spyratos, 1981)—and imposing it on all lenses rules out many useful transformations. In particular, conditional and iteration operators are not very well behaved for reasons that seem pragmatically unavoidable. It turns out that a lens is very well behaved if and only if every equivalence class S_i of S induced by $\sim_{max}$ maps via *get* to all of V, or formally, if *get* $S_i = V$ (lifting *get* from strings to sets in the obvious way).

4 Quotient Lenses

Now let's extend the tearline lens a little further, by developing a lens that converts between MediaWiki and HTML. This lens is useful, for example, for rendering documents for viewing in a web browser, or for manipulating MediaWiki sources using HTML authoring tools.

In the *get* direction, the `html` lens takes a string formatted in the simple subset of MediaWiki I am working with, and produces a string in a corresponding fragment of HTML. For example, it maps

```
==Chefs==
* Julia Child
==Supreme Court Justices==
* Arthur Goldberg
```

to

```
<html>
  <body>
    <h2>Chefs</h2>
    <ul>
      <li>Julia Child</li>
    </ul>
    <h2>Supreme Court Justices</h2>
    <ul>
      <li>Arthur Goldberg</li>
    </ul>
  </body>
</html>
```

and conversely in the *put* direction.

But notice that although this lens is almost bijective, there are artifacts of representing HTML trees as strings—in particular, the amount of whitespace between elements—that make this not literally true. We would like to have the *put* function map an updated HTML document such as

```
<html><body>
  <h2>Chefs</h2>
  <ul><li>Julia Child</li><li>Jacques Pepin</li></ul>
```

```
  <h2>Supreme Court Justices</h2>
  <ul><li>Felix Frankfurter</li><li>Arthur Goldberg</li></ul>
</body></html>
```

to the MediaWiki source

```
==Chefs==
* Julia Child
* Jacques Pepin
==Supreme Court Justices==
* Felix Frankfurter
* Arthur Goldberg
```

but the *get* function maps this MediaWiki source to a view that is, strictly speaking, different than the one we started with:

```
<html>
  <body>
    <h2>Chefs</h2>
    <ul>
      <li>Julia Child</li>
      <li>Jacques Pepin</li>
    </ul>
    <h2>Supreme Court Justices</h2>
    <ul>
      <li>Felix Frankfurter</li>
      <li>Arthur Goldberg</li>
    </ul>
  </body>
</html>
```

Thus, this transformation violates the PUTGET law, which requires that the updated source map back to the updated view exactly. Of course, the differences between this view and the one we started with are not important—the two strings represent the same HTML tree—but Boomerang is a general-purpose language that does not know anything about trees. From this general perspective, the lens is not well behaved.

The observation that the same HTML tree can be encoded by many different strings hints at a solution. Although many real-world transformations do not literally obey the lens laws, they do obey them modulo "insignificant details." Thus, we should loosen the requirements on lenses to allow such unimportant differences to be ignored. In the relaxed semantic space of quotient lenses, we will be able to show that the `html` lens is well behaved modulo an equivalence that relates views encoding the same HTML tree.

4.1 Foundations of Quotient Lenses

Semantically, the definition of quotient lenses is a straightforward generalization of basic lenses. Let $\sim_S$ and $\sim_V$ be equivalence relations S and V. A *quotient lens* l mapping between S

(modulo $\sim_S$) and V (modulo $\sim_V$) has components with the same types as a basic lens, but is only required to obey the lens laws up to $\sim_S$ and $\sim_V$:

$$l.\mathit{put}\ (l.\mathit{get}\ s)\ s \sim_S s \qquad \text{(GETPUT)}$$

$$l.\mathit{get}\ (l.\mathit{put}\ v\ s) \sim_V v \qquad \text{(PUTGET)}$$

$$l.\mathit{get}\ (l.\mathit{create}\ v) \sim_V v \qquad \text{(CREATEGET)}$$

These relaxed laws are just the basic lens laws on the equivalence classes obtained by quotienting S and V, $S/\sim_S$ and $V/\sim_V$ (and when $\sim_S$ and $\sim_V$ are equality they revert to the basic lens laws precisely). But although we want to reason about the behavior of quotient lenses as if they worked on equivalence classes, their component functions actually manipulate representatives—i.e., members of the underlying sets of concrete and abstract structures: the type of *get* is still $S \to V$, not $S/\sim_S \to V/\sim_V$. Thus, we need three additional laws stipulating that the functions respect $\sim_S$ and $\sim_V$.

$$\frac{S \sim_S S'}{l.\mathit{get}\ S \sim_V l.\mathit{get}\ S'} \qquad \text{(GETEQUIV)}$$

$$\frac{v \sim_V v' \qquad S \sim_S S'}{l.\mathit{put}\ v\ S \sim_S l.\mathit{put}\ v'\ S'} \qquad \text{(PUTEQUIV)}$$

$$\frac{v \sim_V v'}{l.\mathit{create}\ v \sim_S l.\mathit{create}\ v'} \qquad \text{(CREATEEQUIV)}$$

These laws ensure that the components of a quotient lenses do not distinguish equivalent structures. The set of quotient lenses mapping between S (modulo $\sim_S$) and V (modulo $\sim_V$) is written $S/\sim_S \Longleftrightarrow V/\sim_V$.

4.2 Canonizers and Quotient Lens Syntax

At the level of syntax, quotient lenses are a bit more interesting. All lens primitives in Boomerang start out as quotient lenses that are well behaved in the strict sense—i.e., that are quotient lenses modulo equality, the finest equivalence relation. To loosen the equivalence on the source or view, we quotient the lens using a new kind of transformation called a *canonizer*. Canonizers have two components: *canonize*, which maps all the strings in a given equivalence class to a string that represents that class, and *choose*, which maps back from strings representing equivalence classes to canonical representatives. Like lenses, canonizers are a kind of bidirectional transformations—indeed, every lens can be converted into a canonizer—but they only have to satisfy weaker requirements: choosing a representative and then re-canonizing must land back in the equivalence class where we stated. This additional flexibility gives enormous latitude for writing canonizing transformations that would not be legal as lenses.

To see how all this works, let us investigate the treatment of whitespace in the `html` lens. First, we need one more lens primitive. The `const` lens takes as arguments a regular expression `E` and two strings, `d` and `e`. Its *get* component maps every source string in `E` to `d` and its *put* component discards the view, which must be `d`, and restores the original source. The *create* component discards `d` and returns a default, `e`. As an example, the `del` lens described previously is derived from `const` in Boomerang:

```
let del (E:regexp where not (is_empty E)) : (lens in E <-> "") =
  const E "" (shortest E)
```

It uses `const` to map all of `E` to the empty string and uses an arbitrary representative of `E` as the default. We can also use `const` to define a sort of dual lens that inserts a string into the view in the *get* direction and removes it in the *put* direction:

```
let ins (s:string) : (lens in "" <-> s) =
  const "" s ""
test (ins "abc").get "" = "abc"
test (ins "abc").put "abc" into "" = ""
```

This lens is almost what we need for inserting whitespace into HTML views, but the strings it inserts into the view are a little too rigid: since the only valid view is the string `s`, any whitespace inserted using `ins` will be uneditable. What we need is a more flexible version whose *get* function inserts whitespace just like `ins`, perhaps according to some pretty printing convention, but whose *put* function accepts—and deletes—arbitrary amounts of whitespace. We can write a quotient lens that has behavior by quotienting `ins` using a canonizer:

```
let qins (E:regexp) (e:string in E) : (lens in "" <-> E) =
  right_quot (ins e) (canonizer_of_lens (const E e e))
```

The `right_quot` operator takes a lens and a canonizer as arguments and quotients its lens argument by wrapping its *get*, *put*, and *create* functions in calls to the canonizer. Specifically, the raw string produced the *get* function of the lens gets post-processed using the canonizer's *choose* component, and the view arguments to *put* and *create* get pre-processed using *canonize*. The `canonizer_of_lens` instance, which builds a canonizer from a lens, uses the *get* function as *canonize* and the *create* function as *choose*. Thus, given the empty string `""`, the *get* component of `qins E e` first applies the *get* of `ins e` to obtain `e` and then the *choose* of the canonizer, which yields `e` again. (For canonizers like this one that normalize some data, the *choose* component of a canonize is usually the identity function; in general, however, there are canonizers whose *choose* components actually do some work.) Given an updated view e', the *put* function first applies *canonize* to obtain `e`, and then invokes the *put* component of `ins e`, yielding `""`. For handling whitespace, this lens has the exact behavior we want, as the following unit tests demonstrate (the `WS` regular expression, defined in the Boomerang standard prelude, denotes whitespace):

```
test (qins WS " ").get "" = " "
test (qins WS " ").put " \n\n " into "" = ""
```

Note that `(qins WS " ")` is not well behaved in the strict sense—the *put* function maps strings such as `" \n\n "` to the empty string, but the *get* function produces the string with a single space. The quotient lens type of `qins` records the fact that the equivalence relation on the view side is the total relation that relates any two strings consisting of whitespace.

Now let's use `qins` to define a lens for building HTML elements. There are actually several different ways we might want to pretty print elements: with whitespace before the opening tag of the element, before its closing tag, or before both tags. To avoid repeating code, and to illustrate the features of Boomerang's functional programming infrastructure, I have factored out the common parts of these lenses into a helper function that takes a parameter specifying which pretty printing convention to use:

```
type loc = Open | Close | Both
let mk_elt_generic (l:loc) (ws:string) (name:string) (body:lens) : lens =
  (match l with
     | Close -> copy EPSILON
     | _     -> qins WS ws : lens) .
  ins ("<" . name . ">") .
  body .
  (match l with
     | Open -> copy EPSILON
     | _    -> qins WS ws : lens) .
  ins ("</" . name . ">")
```

The `l` and `ws` arguments to `mk_elt_generic` represent the location and amount of whitespace that should be inserted around the element, `name` is the name of the element, and `body` is the lens used to process the data it contains. Partially applying this generic helper to each `loc` yields three lens constructors handling each of the pretty printing conventions described above:

```
let mk_outer_elt = mk_elt_generic Close
let mk_simple_elt = mk_elt_generic Open
let mk_elt = mk_elt_generic Both
```

Using these definitions, the rest of the lenses are straightforward. The strings `NL2`, `NL4`, etc. are defined in the standard prelude and denote newlines followed by fixed amounts of indentation and `Xml.esc_string` is a function from Boomerang's XML library that handles the escaping of special characters—e.g., it rewrites `<` to `<` and vice versa (the `invert` primitive flips the direction of a bijective lens):

```
let text : lens = invert (Xml.esc_string [=*%])
let paragraph : lens =
  mk_simple_elt NL4 "p" ((text . copy NL)* . (text . del NL))
let list_elt : lens  =
  mk_simple_elt NL6 "li" (del "* " . text)
let list : lens =
  mk_elt NL4 "ul" (list_elt . del NL)+
let section : lens =
  mk_simple_elt NL4 "h2" (del "==" . text . del "==") .
```

```
  (del NL . (paragraph | list))*
let html : lens =
  mk_outer_elt NL0 "html"
    (mk_elt NL2 "body" section* )
```

This lens has the behavior we want: the *get* function maps MediaWiki documents to pretty-printed HTML, and the *put* and *create* functions map HTML documents containing arbitrary whitespace back to MediaWiki. Using sequential composition, we can build a lens that does regrading and conversion to HTML:

```
let tearline_html : lens =
  tearline ; html
```

Quotient lenses are a critical piece of technology that makes it possible to build the bidirectional transformations that are needed in practice. Besides whitespace, I use quotient lenses to handle escaping, line wrapping, sorting, and filtering, and duplication in many of lenses I have built. Although the problems that quotient lenses solve may appear minor at first sight, they are important for handling real-world examples and attempting to sidestep these problems renders many lenses essentially useless.

As a final example, consider the following quotient lens, which uses `dup2` to generate a table of contents from a second copy of the source:

```
let toc_section : lens =
  mk_simple_elt NL6 "li" (del "==" . text . del "==") .
  del (NL . (PARAGRAPH | LIST)* )
let toc : lens =
  mk_simple_elt NL4 "h1" (ins "Table of Contents") .
  mk_elt NL4 "ul" (toc_section* )
let html_toc : lens =
  mk_outer_elt NL0 "html"
    (mk_elt NL2 "body" (dup2 section* (get toc) (atype toc)))
let tearline_html_toc : lens =
  tearline ; html_toc
```

The *get* function maps the original tearline source to the following view:

```
<html>
  <body>
    <h1>Table of Contents</h1>
    <ul>
      <li>Chefs</li>
      <li>Supreme Court Justices</li>
    </ul>
    <h2>Chefs</h2>
    <ul>
      <li>Julia Child</li>
    </ul>
    <h2>Supreme Court Justices</h2>
```

```
    <ul>
      <li>Arthur Goldberg</li>
    </ul>
  </body>
</html>
```

The *put* component of `tearline_html_toc` does the reverse transformation—in particular, it discards the generated table of contents. The quotient lens type for `tearline_html_toc` records the fact that updates to the the table of contents do not round-trip; the entire table is considered ignoreable information.

5 Fine-grained Secure Documents

The tearline document model we have been working with so far works well, but it requires that users maintain a strict separation between the confidential and public regions of documents. In this section, I design a final lens for computing security views over documents that relaxes this burden using a finer-grained document model in which confidential and public data may be mixed freely. Understanding the details of how this lens works is not important for the discussion that follows in later sections. Therefore, to keep things moving, I just illustrate the main features of this lens using examples, and do not explain how it is implemented in Boomerang; Appendix A contains a complete source code listing.

In the fine-grained document model, individual sections, paragraphs, and list items may be tagged as confidential. The syntax is as follows: confidential sections and paragraph are indicated by including the line "`%% CONFIDENTIAL %%`" before them; confidential list items are indicated by lines that begin with "`%`" (ordinary public items begin with "`*`"). For example, in the following document, the entire first section and the initial paragraphs of the next two sections are confidential:

```
%% CONFIDENTIAL %%
==Office of Strategic Services==
The Office of Strategic Services was the forerunner to
the Central Intelligence Agency.
==Chefs==
%% CONFIDENTIAL %%
Julia Child worked for the Office of Strategic Services
during World War II.

* Julia Child
==Supreme Court Justices==
%% CONFIDENTIAL %%
Arthur Goldberg worked for the Secret Intelligence Branch.

* Arthur Goldberg
```

Allowing confidential data to be scattered throughout the document is convenient for users, because it allows them to control access to particular pieces of the document very precisely.

But this added flexibility do not come without cost: the task of defining updateable security views over fine-grained documents is more complicated than for simple tearline documents. In particular, the designer of a lens has the following additional choices about how its *put* function should behave:

- **Erasing/Redacting:** The *get* function of a lens that computes a tearline view erases the confidential regions completely by tearing off the public regions. In fine-grained documents it is also reasonable for the *get* function to merely redact the the confidential regions by obscuring their content.
- **Alignment:** We have already seen that *put* functions can align the source and view positionally, or using an association calculated from notions of chunk and key. The same issues arise in fine-grained security views. Additionally, because a single region can contain multiple confidential subregions, we can choose to have confidential regions associated to the public subregions that precede or follow them.
- **Nesting/Flattening:** In tearline documents, confidential information appears in exactly one location—just before a tearline. However, in the fine-grained model, confidential information can appear at several different nested levels of document structure. The *put* function can either restore confidential information using a policy that strictly follows this nested structure, or it can use a policy that aligns different kinds of confidential information independently.

Let us examine each of these choices briefly.

First, consider the choice to hide versus redact confidential information. Given the source document shown above, we can easily build a lens whose *get* function erases the confidential information completely, yielding the following view:

```
==Chefs==
* Julia Child
==Supreme Court Justices==
* Arthur Goldberg
```

Alternatively, we can build a lens that redacts by replacing the contents of confidential regions with special strings marking the location of a confidential region in the source:

```
==%% REDACTED %%==
==Chefs==
%% REDACTED %%

* Julia Child
==Supreme Court Justices==
%% REDACTED %%

* Arthur Goldberg
```

Marking data as redacted leaks more information about the source to the regraded view (we will return to this, in our analysis of confidentiality in Section 6). For example, this regraded view leaks the fact that confidential information exists for both Child and Goldberg. It is sometimes useful, however, for public users to be able to see the placement of secrets—e.g., to edit the placement of confidential information—even if they cannot view their contents. Lastly, note that we can choose to erase some information while redacting others. The lens definitions in Appendix A are written so that the choice to erase or redact confidential regions can be picked independently for sections, paragraphs, and list items.

Next, consider the choice of an alignment strategy for the *put* function. As before, we can align data positionally, so that if we have an view consisting of a single list of names

```
* Dick Cheney
* George W. Bush
```

and *put* it back into a source containing the names followed by their Secret Service code names (so here the update has modified Cheney's first name to "Dick", added Bush's initial, and swapped the order of names)

```
* George Bush
% Tumbler or Trailblazer
* Richard Cheney
% Angler
```

we get an updated source

```
* Dick Cheney
% Tumbler or Trailblazer
* George W. Bush
% Angler
```

where the code names are restored positionally. Alternatively, we can use dictionary lenses to use each public item as the key of a chunk consisting of that public item followed by the sequence of confidential items that follow. The *put* function of this lens maps the same updated view and source to

```
* Dick Cheney
% Angler
* George W. Bush
% Tumbler or Trailblazer
```

We can also arrange the chunks so that a public item serves as the key for the confidential items that precedes it. Note that this lens uses "fuzzy" alignment that was discussed earlier—even though we have changed both public items, the *put* function still associates each confidential item to the correct public item.

The third design choice concerns the treatment of nested confidential regions. For example, if a list containing confidential items appears within a public section, then we can either design the *put* function so that confidential lists are only ever restored if it also aligns the enclosing

sections, or so that the alignment of list items and sections are independent. I call the first strategy "nested" and the second "flat". To illustrate the difference, consider the following toy source document,

```
"==First==
 * public
 % confidential
 ==Second==
 text"
```

which maps to

```
"==First==
 * public
 ==Second==
 text"
```

by an erasing *get* function. If we use the nested strategy, and the update to the view moves a list across a section boundary, then the confidential data for that list will be lost. For example, putting

```
"==First==
 text
 ==Second==
 * public"
```

into the source yields

```
"==First==
 text
 ==Second==
 * public"
```

since the confidential data list item is aligned with the section named `First`, which contains no lists. Using the flat approach, then the *put* function produces

```
"==First==
 text
 ==Second==
 * public
 % confidential"
```

instead—the alignment of list items and of sections are completely independent.

This concludes our tour of Boomerang, and of lenses for secure documents. Next I discuss the extensions to lenses for reasoning about properties of trustworthy updateable security views.

6 Trustworthy Security Views

Boomerang's type system guarantees that the components of well-typed lenses are total functions and that they obey the lens laws. These properties go a long way towards ensuring that lenses build correct updateable views, but in security settings, we need more. In particular, we need to be able to establish, in a formal way, that the *get* function of a lens does not leak the information we intend to keep secret, and that its *put* function does not taint source data. This section describes some preliminary ideas towards extending Boomerang to meet both of these needs.

6.1 Confidentiality

The whole reason for defining a security view is to restrict access to particular parts of the source. Thus, it is useful to have a way to establish that the information in the source we intend to keep secret is not leaked into the view. For simple queries, such as the *get* function of the lens for tearline documents, it is not difficult to see that the confidential information—i.e., the contents of the regions above each tearline—is not leaked to the view. But for queries that are not just simple projections, such as the redacting lenses for fine-grained documents, or lenses built using sequential composition, it is not always obvious which information in the source is kept secret.

This section describe a refined type system for Boomerang based on an information-flow analysis. It facilitates precise, end-to-end reasoning about confidentiality properties of lenses. The outline of the development is standard—I show how to decorate lens types with annotations drawn from a lattice $\mathcal{L}$ of labels representing clearances, I extend the typing rules for lenses to track data flows, and I prove a *non-interference* theorem establishing that the low-security parts of the view do not depend on any high-security parts of the input Sabelfeld and Myers (2003)—but the details, in particular the application of information flow to regular types, is novel.

I start by defining the semantics of security-annotated types. Formally, I extend the syntax of regular expressions, which are the main building blocks for lens types in Boomerang, to annotated versions by extending the grammar

$$\mathcal{R} ::= u \mid \mathcal{R}{\cdot}\mathcal{R} \mid \mathcal{R}|\mathcal{R} \mid \mathcal{R}^*$$

with a new production $\mathcal{R} : l$ where l is a label in $\mathcal{L}$. The semantics of an annotated expression R has two components: a set of strings $[\![R]\!]$ and a family of label-indexed binary relations $\sim_l^R$ on $[\![R]\!]$. The set of strings $[\![R]\!]$ is the same set as is denoted by the ordinary regular expression obtained by erasing all the annotations in R. The relations capture the notion that two strings cannot be distinguished by an observer with clearance is l.

Interestingly, there seem to be several reasonable semantics for these observability relations. At this preliminary stage of the investigation, I do not have any evidence that leads me to favor one semantics of annotated regular expressions. Thus, I will describe all three; any of these semantics can be plugged in to the rest of the information-flow type system for lenses.

$$
\begin{array}{rcl}
\mathit{brace}_u(s) & = & = s \\
\mathit{brace}_{(R_1 \cdot R_2)}(s_1 \cdot s_2) & = & \mathit{brace}_{R_1}(s_1) \cdot \mathit{brace}_{R_2}(s_2) \\
\mathit{brace}_{R_1 *}(s_1 \cdots s_n) & = & \mathit{brace}_{R_1}(s_1) \cdots \mathit{brace}_{R_1}(s_n) \\
\mathit{brace}_{R_1 : l}(s) & = & \begin{cases} s & \text{if } l = L \\ (s) & \text{otherwise} \end{cases} \\
\mathit{brace}_{(R_1 | R_2)}(s) & = & \begin{cases} \mathit{brace}_{R_1}(s) & \text{if } s \in [\![R_1]\!] \setminus [\![R_2]\!] \\ \mathit{brace}_{R_2}(s) & \text{if } s \in [\![R_2]\!] \setminus [\![R_1]\!] \\ \mathit{merge\_braces}(\mathit{brace}_{R_l}(s), \mathit{brace}_{R_2}(s)) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 1: Brace insertion.

For simplicity, I work in a two-point lattice with $L < H$; generalizing to more interesting lattices is straightforward. I only give the definition of $\sim_L$ since the relation $\sim_H$ is equality—an observer with the highest level of clearance can distinguish strings exactly when they are different strings.

Let R be an annotated regular expression in which concatenations and iterations are unambiguous (these conditions are required by the unique splittability conditions on the typing rules for lenses anyways). I first define in Figure 1 a function *brace* that maps strings in $[\![R]\!]$ to strings where the secrets regions are delimited using special symbols `(` and `)`. The function *merge_braces* is a function on pairs of well-braced strings and merges their braces by to yield another well-braced string—e.g.: $\mathit{merge\_braces}(\texttt{(ab)c}, \texttt{a(bc)}) = \texttt{(abc)}$. The most interesting cases in this definition are $\mathit{brace}_{R_1 : l}$, which puts the entire string in braces if $l = H$ and does nothing if $l = L$, and $\mathit{brace}_{R_1 | R_2}$ in the case where $[\![R_1]\!]$ and $[\![R_2]\!]$ are not disjoint, which merges the braces assigned by observability relations of R_1 and R_2.

The first semantics intuitively corresponds to "erasing" secrets. Two strings can be distinguished by an individual with L clearance in this semantics if inserting braces and then erasing the braces and all the characters they enclose yields identical strings:

$$s_1 \sim_L^R s_2 \iff \mathit{erase}(\mathit{braces}_R(s_1)) = \mathit{erase}(\mathit{braces}_R(s_1))$$

where *erase* is the obvious function on strings—e.g. $\mathit{erase}(\texttt{(ab)c}) = \texttt{c}$. As an example, if R is `[a-z]:H` then for every string s in $[\![R]\!]$ we have that $\mathit{erase}(\mathit{braces}_R(s))$ is the empty string, so $\sim_L^R$ is the total relation. Likewise, if R is `([a-z]:H)*` then $\sim_L^R$ is also the total relation because for every s in $[\![R]\!]$ we have $\mathit{erase}(\mathit{braces}_R(s))$ is the empty string. Finally, if R is `[a-z]:L.[0-4]:H | [a-z]:L.[5-9]:H` then for any string s in $[\![R]\!]$ where $s = cn$ with $c \in [\![\texttt{[a-z]}]\!]$ and $n \in [\![\texttt{[0-9]}]\!]$ we have $\mathit{erase}(\mathit{braces}_R(s)) = c$. It follows that $cn \sim_L^R c'n'$ if and only if $c = c'$.

The second semantics intuitively corresponds to "redacting" secrets. The observability relation is defined in the same way as in the erasing semantics, except that in the final step, instead of erasing regions enclosed in braces, we use *redact* to replace them with a symbol `#`—e.g., $\mathit{redact}(\texttt{(ab)c}) = \texttt{\#c}$:

$$s_1 \sim_L^R s_2 \iff \mathit{redact}(\mathit{braces}_R(s_1)) = \mathit{redact}(\mathit{braces}_R(s_1))$$

The different between this semantics and the previous one is illustrated best by considering an annotated regular expression that involves Kleene star. If we take R to be `([a-z]:H)*` then a low observer can distinguish `ab` and `abcde` because bracing and redacting maps the first string to `##` and the second string to `#####`. In the erasing semantics, we saw that $\sim_L^R$ was the total relation.

A final semantics, also based on redacting, can be obtained by obscuring each individual character rather than the entire redacted region. For example, if we take R to be the type `([a-z]*:H)` (note the different placement of the Kleene star) then a low security observer can distinguish `ab` from `abcde` in this semantics since marking and redacting maps these strings `##` and `#####`. By contrast, the first redacting semantics maps both strings to `#`, so they are indistinguishable.

Having defined several semantics for annotated regular expressions, I now describe revised the typing rules for the core lens combinators with an information-flow analysis. These revised rules satisfy the following non-interference theorem:

6.1 Theorem [Non-interference]: Let $l \in S \Longleftrightarrow V$ be a lens, $l \in \mathcal{L}$ be a label, and $s, s' \in [\![S]\!]$ with $s \sim_j^S s'$. Then $l.\mathit{get}\ s \sim_j^V l.\mathit{get}\ s'$.

The standard Boomerang typing rules for some of the combinators, `copy`, `const`, and concatenation already satisfy the non-interference property:

$$\frac{E \in \mathcal{R}}{\mathit{copy}\ E \in E \Longleftrightarrow E} \qquad \frac{E \in \mathcal{R} \qquad e \in [\![E]\!]}{\mathit{const}\ E\ d\ e \in E \Longleftrightarrow d} \qquad \frac{\begin{array}{ll} l_1 \in S_1 \Longleftrightarrow V_1 & S_1 \cdot^! S_2 \\ l_2 \in S_2 \Longleftrightarrow V_2 & V_1 \cdot^! V_2 \end{array}}{l_1 \cdot l_2 \in S_1 \cdot S_2 \Longleftrightarrow V_1 \cdot V_2}$$

(The notation $S_1 \cdot^! S_2$ in the typing rule for concatenation asserts that strings in $S_1 \cdot S_2$ be uniquely splittable; ${S_1}^{!*}$, used for Kleene star below, denotes an analogous condition.)

The rules for union and Kleene star, however, do not satisfy non-interference. Intuitively, the reason is that the union combinator fails to have the property is that an observer can deduce facts about the source by examining the view and reasoning backwards from the branch that was selected. In this way, even if no secrets are explicitly copied from the source to view, the observer can sometimes still glean information about the source. For example, in the following lens

```
let l : lens =
     (copy [a-m] . del [5-9])
  || (copy [a-z] . del [0-4])
```

(the notation `||` is a variant of lens union) if the letters are public and the numbers are secret, then observing a view `[n-z]` reveals that the secret number in the source belongs to `[0-4]` and not `[5-9]`.

The standard way to track these implicit flows of information is using a typing rule that escalates the label on the view component of the type by the label of the data that was used to pick a branch. The lens union operator picks a sublens by testing which sublens has a type

that matches the source string. Thus, the label of the data used to pick a branch is the minimal label that observes that the two source types are disjoint:

6.2 Definition [Observes Disjoint]: Let R_1 and R_2 be security-annotated regular expressions and let $l \in \mathcal{L}$ be a label. I will say that l observes $R_1 \cap R_2 = \emptyset$ if and only if for every $r, r' \in [\![R_1 \mid R_2]\!]$ if $r \in [\![R_1]\!]$ and $r' \in [\![R_2]\!]$ then $r \not\sim_L^{R_1|R_2} r'$.

With this definition, the typing rule for union is a straightforward generalization of the standard Boomerang rule:

$$\frac{\begin{array}{c} l_1 \in S_1 \Longleftrightarrow V_1 \qquad l_2 \in S_2 \Longleftrightarrow V_2 \\ j = \mathit{join}\{j' \mid j' \text{ minimally observes } S_1 \cap S_2 = \emptyset\} \end{array}}{l_1 \mid l_2 \in (S_1 \mid S_2) \Longleftrightarrow (V_1 \mid V_2) : j}$$

Similar issues with implicit flows come up with the Kleene star operator. For example consider the lens

```
let l : (lens in [a-z]* <-> "a"* ) =
  (const [a-z] "a" "a")*
```

The `const` lens being iterated can validly be given the type `[a-z]:H <-> "a":L` However, `l` cannot be given the type `([a-z]:H)* <-> ("a":L)*` To see why, observe that in the erasing semantics, an L observer cannot distinguish any source strings, but they can distinguish different views. Thus, we need to escalate the label on the view component of the type by the label that observes the iterability of the source.

6.3 Definition [Observes Iterable]: Let R_1 be a security-annotated regular expression and let $l \in \mathcal{L}$ be a label. I will say that l observes ${R_1}^{!*}$ if and only if for every $r_1, \ldots, r_k, r'_1, \ldots r'_l \in [\![R_1]\!]$ we have that $r_1 \cdots r_k \sim_l^{R_1|R_2} r'_1 \cdots r'_l$ implies $k = l$.

With this definition, the typing rule for Kleene star is a straightforward generalization of the standard rule in Boomerang:

$$\frac{\begin{array}{c} l_1 \in S_1 \Longleftrightarrow V_1 \\ j = \mathit{join}\{j' \mid j' \text{ minimally observes } {S_1}^{!*}\} \end{array}}{{l_1}^* \in ({S_1}^*) \Longleftrightarrow ({V_1}^*) : j}$$

This type system represents a first step toward a system in which it will be possible to decorate the source and target components of a lens type with security annotations, and guarantee that the *get* function does not leak data marked secret. To finish the job, a few tasks remain. First, I would like to study the various semantics for security-annotated regular types further. (For example, I would like to relax the condition that concatenations and unions be unambiguous.) Second, the typing rules must be integrated with all of the other features of Boomerang—in particular, the interaction with quotient lenses may be interesting. Second, to really be able to use this type system in practice, I need to find or design algorithms for

deciding properties such as type equivalence and minimal label observing the disjointness and iterability of security-annotated types. I am optimistic that such procedures can be found, perhaps by adapting previous work on the weighted automata that have been thoroughly studied in the context of natural language processing (Mohri, 2004).

6.2 Integrity

The second security property we need to be able to track is integrity. Allowing security views to be updated opens the door to a whole host of nasty complications because it makes it possible for individuals to modify source data (via the view) that they do not even have clearance to access! Even worse, for many transformations, the modifications to the source are irreversible—we cannot back them out simply by rolling back to the change to the view.

As an example illustrating these problems, consider the following toy lens

```
let l : (lens in ? <-> [a-z]) =
  (copy [a-m] . del [0-4]) | (copy [n-z] . del [5-9])
```

(the notation `? <-> [a-z]` asserts that `l` has a type whose source type is anything and view type is `[a-z]`) and suppose that the letters (`[a-z]` and `[n-z]`) copied from the source to view are public while the numbers (`[0-4]` and `[5-9]`) are secret. The *get* function maps the source `m1` to a view `m` that only contains public information. If a user with public clearance modifies this view to `n`, then the *put* function cannot restore the deleted number by copying it from the original source because `n1` is not valid. So instead, the union lens will discard the original source and invoke the *create* function of the second sublens, yielding `n5` as a result. The key thing to notice is that propagating the changes made by an untrusted user to a public view changes secret data in the source—namely the deleted number goes from `1` to `5`.

A similar problem comes up with the Kleene star operator. Consider the lens

```
let l : (lens in ([a-z] . [0-9])* <-> [a-z]* ) =
  (copy [a-z] . del [0-9])*
```

and suppose that, as in the previous example, the letters are public and the numbers are secret. The *get* function maps a source `a1b2c3` to the view `abc`. If a public user then modifies this view to `ab`, then the *put* function produces the new source `a1b2`. This is a reasonable result, but it means that the secret value `3` is deleted.

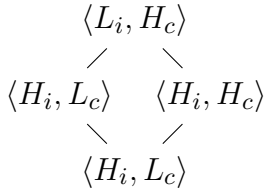
There are two broad approaches we could take to handling this problem: we could restrict the class of transformations for defining security views to ones that do not suffer from the problems illustrated by the examples above, or we could instrument the transformations to provide sufficient information for a trusted user to audit the changes induced by propagating updated views back to the source.

Recall that a lens is called very well behaved if two *put*s in sequence just has the effect of the second:

$$l.\mathit{put}\ v\ (l.\mathit{put}\ v'\ s) = l.\mathit{put}\ v\ s$$

Requiring that all lenses be very well behaved would ensure that updates to the view did not cause the source to be changed irreversibly: to roll back the changes induced after one *put*, we would just invoke *put* again with the old view and the new source. However, as discussed previously, very well behavedness is an extremely restrictive condition that rules out several core combinators that we use all the time in practice. In particular, we would have to give up most instances of union and Kleene star. Thus, requiring very well behavedness in general would restrict security views a class of transformations that is too weak to build many examples of practical interest.

A different approach would be to design a static analysis tracking which parts of the source have been tainted. It has often been noticed that confidentiality and integrity are dual notions. Formally, this idea would work by generalizing our lattice of security clearances to track confidentiality and integrity simultaneously and flipping around the non-interference theorem around so that it applies to the *put* function as well. First, we would generalize the lattice of security labels to a product lattice—the label $\langle l_i, l_c \rangle$ represents data with integrity l_i and confidentiality l_c—with order in the lattice calculated component-wise. For example, if there are just two levels of confidentiality and integrity, then the combined lattice is as follows:



Intuitively, labels that are higher in the lattice place more restrictions on programs that manipulate data carrying that label. For example, the top element, representing "low" integrity and "high" confidentiality (i.e., tainted and secret) data, requires the most care, while "high" integrity and "low" confidentiality (i.e., endorsed and public) data can be used freely.

Unfortunately, using a static type system to track integrity does not seem to give precise enough information about the locations of changes. Consider the lens `l` built using Kleene star from above. The source starts out as a list containing a mixture of endorsed and confidential ($\langle H_i, H_c \rangle$) data and endorsed and public ($\langle H_i, L_c \rangle$) data. Regrading this source using the *get* function yields a uniform list of endorsed and public ($\langle H_i, L_c \rangle$) data. An edit to the list by an untrusted party introduces new data that is tainted and public ($\langle L_i, L_c \rangle$). But now we have a problem: the type of edited lists is a mixture of endorsed and tainted public data ($\langle H_i, L_c \rangle$ and $\langle L_i, L_c \rangle$), and this type does not tell us which list elements are tainted, but only that some elements (may) have be tainted, but not which elements are tainted. Thus, applying a static analysis based on information flow seems to result in a system that is too coarse in practice.

Static techniques do not seem capable of tracking integrity with enough precision to be useful in practice. So instead, let us explore dynamic techniques. Broadly, the idea is that propagating changes to the view should generate metadata about the source —in the form of provenance or an explicit log—that a trusted user can use to audit the changes and verify that all of the changes are reasonable.

The first idea based on a dynamic approach uses *provenance* metadata. Provenance is metadata that records the facts about the history of a piece of data as it flows through a computation. The idea here is to interpret the provenance as integrity. We would first extend the data model so that every piece of the source and view carry an explicit annotation indicating whether it is endorsed or tainted. We would then generalize the semantics of lenses so that the *get*, *put*, and *create* functions operate on these annotated strings—in particular, so that they propagate this metadata as they transforming the underlying strings (Green et al., 2007; Foster et al., 2008a). Source data would originate with "neutral", endorsed provenance. But as untrusted users edit the view, they would tag as tainted the specific pieces of the view they change. The *put* function would then propagate these annotations back to the source, yielding a structure whose annotations precisely identify the pieces of the source that have been changed and that need to be audited. In certain ways, this design is quite attractive—indeed, for tracking integrity it seems to give almost everything that is needed—but it requires deep structural to the data model and the semantics of lenses. Additionally, it means that applications built using lenses will require tighter integration, since they will need to manipulate the provenance metadata of views.

The final idea for tracking integrity, and the one that I propose to actually investigate and implement in Boomerang, approximates the behavior of previous without requiring that all structures carry explicit annotations by imposing a few simplifying assumptions. It is based on the observation that the *put* function already infers when and how it changes the source, and so it can use this information to generate an audit log. For example, returning to the example from before,

```
let l : (lens in ([a-z] . [0-9])* <-> [a-z]* ) =
  (copy [a-z] . del [0-9])*
```

with the source `a1b2c3` and view `ab`, the *put* function can detect that the first two pieces of the source and view will be restored exactly, while the third piece will be deleted, and it can record these facts in a log.

This approach depends on two simplifying assumptions. First, the source only contain endorsed data. This means that data model and *get* function do not need to be generalized to work on annotated strings. Second, it assumes that the edits to the view originate with a single user. This means that the *put* function does not need to track which user has edited the view, only that it has been edited.

In fact, I believe that this second assumption can be relaxed—the architecture can be generalized to multiple untrusted users using a synchronizer. If multiple users need to edit a view then they should first agree on a consistent view states using a synchronizer (possibly using several rounds of synchronization to resolve conflicts). After synchronizing, the log generated by the synchronizer indicates which edits to specific pieces of the view originated with which users. The information in this log is analogous to the annotated strings described in the previous approach. The *put* function can then use this extra information to report the changes to the new source precisely. Returning to our example, if after synchronizing both Alice and Bob agree that `c` should be deleted from the view, then the output of the synchronizer will be

the new view `ab` and a log indicating that `c` was deleted by both parties. The *put* function can combine these pieces of information to produce an appropriate log for the source recording the fact that `c3` was deleted by both.

The main thing lost with the implicit approach compared to the previous one with explicit provenance is the ability to represent sources that contain both endorsed and tainted data—i.e., audits need to be done after each update. The benefits, however, are significant: *get* functions do not have to be retooled to propagate annotations, and applications do not need to indicate which parts of the view they have changed—the synchronizer and the *put* function infer this information instead. Thus, I believe it to be a promising approach to tracking integrity in security applications built on lenses.

7 Related Work

The original paper describing basic lenses (Foster et al., 2007b) includes an extensive survey of work related to lenses. In this section, I highlight just a few of the most relevant pieces of work for completeness, and to provide references for languages that have emerged since that paper was published as well as for security views.

In the area of foundations, the work most closely related to lenses comes from the database literature. The seminal papers of Dayal and Bernstein (1982) and Bancilhon and Spyratos (1981) propose notions of correct view update for relational systems. Dayal and Bernstein's notion of "exactly performing an update" corresponds to the PUTGET law of basic lenses and Bancilhon and Spyratos's "constant complement" approach corresponds to very well behaved lenses. The idea of using constant complements also influenced later work by Lechtenbörger (2003). Hegner (2004) studied a additional condition called *monotonicity*—updates that add more records to the view should be translated to larger source updates (and similarly for deletions). Similar theories based on ordering translations have also been studied by Johnson and Rosebrugh (2007) and Buneman, Khanna, and Tan (2002). In particular, the latter work established the intractability of inferring minimal updates in the relational setting for standard query languages, as discussed in the introduction. Finally, the tradeoffs between update translators that are total and ones that may fail has been studied by Hegner (1990).

There are a large number of programming languages that, in some way, describe bidirectional transformations. Basic lenses for relations, using primitives based on relational algebra, have been developed by Bohannon, Vaughan, and Pierce (2006).

Meertens's constraint maintainers for user interfaces addresses the problem of connecting graphical objects on the screen so that, whenever one of the objects is changed by the user, the change can be propagated by the maintainer to the other object such that the relationship between the objects is maintained (Meertens, 1998). Similar issues were studied by Greenberg in the context of DOM-based interfaces (Greenberg and Krishnamurthi, 2007).

The bidirectional combinator language X is a tree manipulation languages loosely based on basic lenses (Hu et al., 2004; Mu et al., 2004). The main focus in X is on developing mechanisms for handling dependencies that arise when source information is duplicated in

the view, motivated by applications to a structure editor. The semantic foundation of these languages is weaker than lenses; in particular, they relax the lens laws to trip-and-a-half versions and allow updates to fail. X was later used as the basis for a bidirectional interpretation of core XQuery (Liu et al., 2007).

Recent work by some of the same authors has tackled the problem of specifying bidirectional transformations using ordinary λ-notation. Starting from a restricted fragment with affine variables, they show how to derive explicit view complements, which they then use to derive the backwards function (Matsuda et al., 2007). As in previous work from the same group, updates are allowed fail. However, they show how to calculate an update checker that tests if a given update will succeed.

A number of bidirectional languages for writing conversions intended to be bijective modulo "ignorable information" have been proposed. XSugar (Brabrand et al., 2008) is a bidirectional language that maps between XML documents and strings. Transformations are specified using pairs of intertwined grammars. A similar language, biXid (Kawanaka and Hosoya, 2006), converts between XML on both sides. The PADS system (Fisher and Gruber, 2005) has a bidirectional language that describes the parser and pretty printer (and a number of additional software artifacts) for an ad-hoc data formats from a single, declarative description. Kennedy's combinators (2004) describe pickler and unpicklers for serializing data out to the disk or network and reading it back in. Benton (2005) and Ramsey (2003) both describe systems for mapping between run-time values in a host language and values manipulated by an embedded interpreter.

There has been a flurry of recent interest in applying bidirectional languages to problems in software engineering. Stevens (2007) applied the ideas of basic lenses in the context of *model transformations*. This idea has also been pursued by the designers of X (Xiong et al., 2007).

The idea of using views to encapsulate sensitive information in XML was proposed by Stoica and Farkas (2002) and extensively studied by Fan, Chan, and Garofalakis (2004). The latter work develops a system similar to the confidentiality framework for *get* functions proposed here. In particular, they emphasize the need to provide a type (formulated as a DTD) for users of the security view. Unlike this work, however, their views are virtual and cannot be updated.

There are a number of programming languages with static type systems that track confidentiality properties using information flow (Sabelfeld and Myers, 2003; Pottier and Simonet, 2003). Swamy, Corcoran, and Hicks have recently proposed a system that combines static information flow and dynamic provenance propagation in the same system(Swamy et al., 2008; Corcoran et al., 2007). Their approach to security properties is largely the same as that proposed in this work, but their focus is on a mechanisms for specifying a variety of different access control properties whereas mine is on the mechanisms of updateable views in bidirectional languages with a type system based on regular expressions.

The designers of the functional language CDuce proposed a dynamic analysis to track information flow for XML transformations (Benzaken et al., 2003). Their goals are similar, but they state their non-interference result in terms of specific subexpressions in a program,

rather than as an end-to-end property of whole transformations.

8 Implementation Status and Roadmap

In this section, I briefly describe the implementation status of the Boomerang system and several of the prototype applications I have assembled using it. I also sketch a roadmap and rough timeline for the remaining work.

8.1 The Boomerang System

I have developed a prototype implementation of the Boomerang language with major assistance from Alexandre Pilkiewcz and Michael Greenberg. This system includes an interpreter for the surface language, native implementations of the core basic, dictionary, and quotient lens combinators, and several lens libraries for handling escaping, lists, sorting, and XML.

The core combinators in Boomerang rely on functions drawn from a regular expression library. These combinators make heavy use of several slightly non-standard operations including operations to decide whether concatenations and iterations are unambiguous. I have implemented an efficient regular expression library in OCaml based on Brzozowski derivatives Brzozowski (1964). The library makes heavy use of hash consing and memoization to avoid recomputing results, and a clever algorithm for deciding ambiguity due to Møller (2001).

As illustrated in many of the example lenses in this document, Boomerang has a very rich type system with regular expression types, dependent function types, refinement types, datatypes, and polymorphism. The combination of these features makes for a very complicated type system. Rather than trying to identify a decidable, fragment of this beast that could be implemented in a static type checker, I have pursued a hybrid approach. Boomerang's type checker uses a very coarse analysis to rule out obviously ill-formed programs and verifies the precise conditions expressed by types using dynamic tests. Much of the recent work on the type checker has been carried out by Michael Greenberg. In particular, he has designed and implemented an algorithm for inserting checks that addresses some serious efficiency problems with the obvious, naive approach.

Using Boomerang, I have developed a number of lenses, several of which are described in the next section. Among the lens that I have implemented is... this very document! All of the examples displayed in a typewriter font have been generated from its literate source file and type checked and run within Boomerang.

8.2 Data Converters and Synchronizers

One of the most compelling applications of lenses is for converting between and synchronizing data represented in different formats. I have implemented lenses for a variety of real-world data formats including vCard and CSV address books, BibTeX and RIS-formatted bibliographies, a number of transformations between text and XML originally developed as a part of the XSugar project (Brabrand et al., 2008), and a large lens that maps between textual and XML

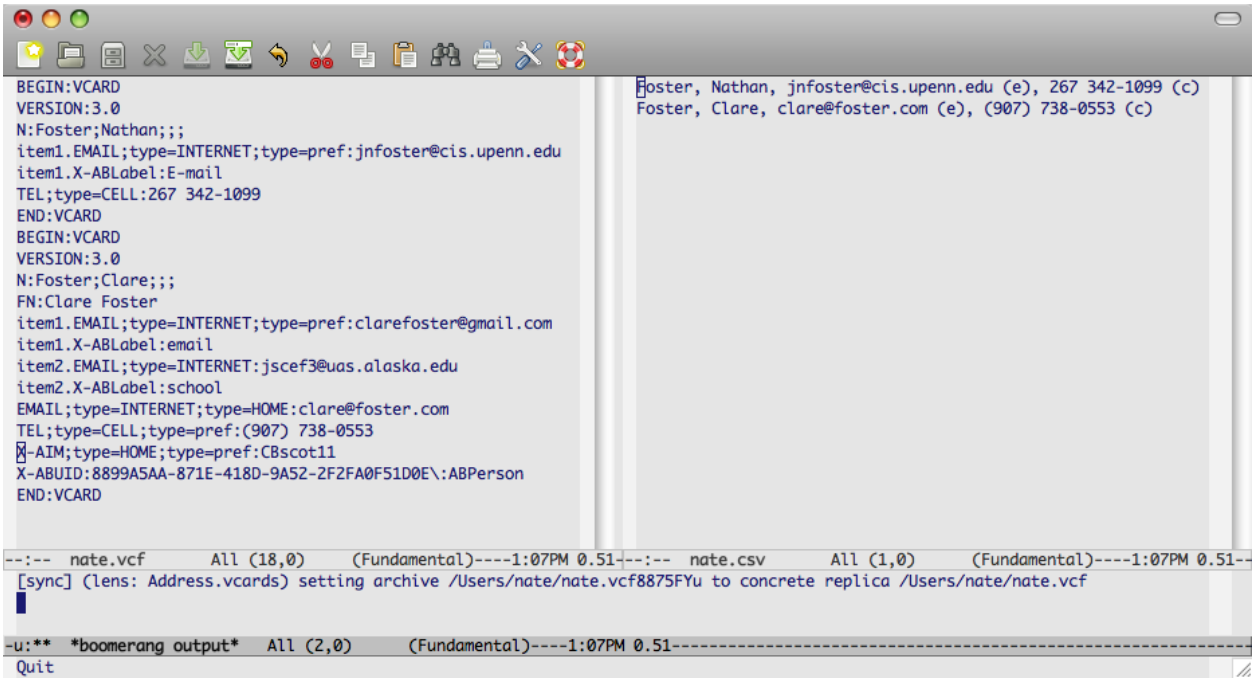


Figure 2: Boomerang Emacs interface.

representations of UniProtKB genomic databases. The Boomerang front-end can be used with these lenses to convert between instances of all of these formats.

With Benjamin Pierce and Alan Schmitt, I have also recently extended a synchronization algorithm, originally developed for trees (Foster et al., 2007a), to strings. This algorithm takes a type as input, which it uses to calculate the alignment of substrings in each replica. This algorithm, whose core uses heuristics based on the Unix utility `diff3`, appears to work well in practice, but we are still studying its formal properties.

8.3 Structure Editor

Another application I have implemented is a simple Emacs interface for editing source and view strings related by a lens. A screenshot of this interface is shown in Figure 2. In this mode, the user has two windows, with the source on the left and the view on the right. They edit either buffer and the Boomerang system propagates the changes from source to view or vice versa when the buffer is saved.

8.4 Roadmap

The remaining work divides cleanly into three main areas. I propose to work on these topics in sequence, with the goal of completing the project by the summer of 2009.

The next immediate step is to complete the investigation into information flow for languages with regular types, and implement the refined type system for lenses in Boomerang. In parallel, I plan to develop the approach sketched in this document for tracking the integrity

of source data using implicit provenance further. I hope that both of these pieces will be finished by the end of the fall semester, and will eventually lead to separate publications—one on information flow for regular types in general, and another on using lenses to build updateable security views. I also plan to spend a significant chunk of time polishing and extending the Boomerang system and its associated applications. Specifically, I hope to develop a few additional lenses for real-world formats, integrate lenses with the Unison file synchronizer, and tune the implementation further by studying algebraic optimizations and building streaming physical implementations of the core lens combinators. It is my firm belief that lenses, even in the relatively simple domain of string transformations, have the potential to have significant practical impact. I am therefore highly motivated to make these ideas accessible to users (and programmers!) by packaging and releasing a robust implementation before the project is finished.

References

François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.

Nick Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4):503–542, 2005.

Véronique Benzaken, Marwan Burelle, and Giuseppe Castagna. Information flow security for XML transformations. In *Advances in Computing Science: Programming Languages and Distributed Computation (ASIAN), Mumbai, India*, volume 2896 of *Lecture Notes in Computer Science*, pages 33–53, 2003.

Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled schema transformation and data conversion for XML and SQL. In *International Symposium on Practical Aspects of Declarative Languages (PADL), Nice France*, volume 4354 of *Lecture Notes in Computer Science*, pages 290–304. Springer-Verlag, 2007.

Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

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.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4–5):385–406, 2008. Preliminary version in DBPL '05.

Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. On propagation of deletions and annotations through views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, Madison, Wisconsin*, pages 150–158, 2002.

Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Combining provenance and security policies in a web-based document management system. In *On-line Proceedings of the Workshop on Principles of Provenance (PrOPr), Edinburgh, Scotland*, November 2007. http://homepages.inf.ed.ac.uk/jcheney/propr/.

Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, September 1982.

David T. Eger. Bit level types, 2005. Unpublished manuscript. Available from http://www.yak.net/random/blt/blt-drafts/03/blt.pdf.

Wenfei Fan, Chee Yong Chan, and Minos N. Garofalakis. Secure XML querying with security views. In *ACM SIGMOD International Conference on Management of Data (SIGMOD), Paris, France*, pages 587–598, 2004.

Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, IL*, pages 295–304, 2005.

Cormac Flanagan. Hybrid type checking. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina, USA*, pages 245–256, 2006.

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 2007a. Preliminary version in DBPL '05.

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 2007b. Preliminary version in POPL '05.

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, Vancouver, BC*, June 2008a.

J. Nathan Foster, Alexandre Pilkiewcz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, BC*, September 2008b. To appear.

Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, Beijing, China*, pages 31–40, 2007.

Michael Greenberg and Shriram Krishnamurthi. Declarative Composable Views, May 2007. Undergraduate Honors Thesis. Department of Computer Science, Brown University.

Stephane J. Hegner. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence*, 40:63–125, 2004. URL http://www.cs.umu.se/~hegner/Publications/PDF/amai03.pdf. Summary in *Foundations of Information and Knowledge Systems*, 2002, pp. 230–249.

Stephen J. Hegner. Foundations of canonical update support for closed database views. In *International Conference on Database Theory (ICDT), Paris, France*, pages 422–436, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 0-387-53507-1. URL http://www.cs.umu.se/~hegner/Publications/PDF/icdt90.pdf.

Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bi-directional transformations. In *Partial Evaluation and Program Manipulation (PEPM)*, pages 178–189, 2004. Extended version to appear in *Higher Order and Symbolic Computation*, 2008.

Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1–2), June 2008. Preliminary version in PEPM '04.

Michael Johnson and Robert Rosebrugh. Fibrations and universal view updatability. *Theoretical Compututer Science*, 388(1-3):109–129, 2007.

Shinya Kawanaka and Haruo Hosoya. bixid: a bidirectional transformation language for XML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Portland, OR*, pages 201–214, 2006.

Andrew J. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

Jens Lechtenbörger. The impact of the constant complement approach towards view updating. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, San Diego, California*, pages 49–55. ACM, June 9–12 2003.

Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of xquery. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM), Nice, France*, pages 21–30, 2007.

David Lutterkort. Augeas–A configuration API. In *Linux Symposium, Ottawa, ON*, pages 47–56, 2008.

Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Freiburg, Germany*, pages 47–58, 2007.

Lambert Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript, available from ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps.

R.J. Miller, M.A. Hernández, L.M. Haas, L. Yan, C.T.H. Ho, R. Fagin, and L. Popa. The Clio Project: Managing Heterogeneity. *ACM SIGMOD Record*, 30(1):78–83, 2001.

Mehryar Mohri. Weighted finite-state transducer algorithms: An overview. *Studies In Fuzziness and Soft Computing*, 148:551–564, 2004.

Anders Møller. The brics automaton package, 2001. URL http://www.brics.dk/automaton/.

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 2–20, November 2004.

Isaac Potoczny-Jones. Tearline Wiki: Information collaboration across security domains, 2008. URL http://www.virtualacquisitionshowcase.com/docs/2008/Galois-Brief.pdf. Whitepaper. Presented at the Navy Opportunity Forum.

Franois Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators (IVME), San Diego, CA*, pages 6–14, 2003.

Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *International Workshop Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, volume 903 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), Nashville, TN*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2007.

Andrei Stoica and Csilla Farkas. Secure XML views. In *IFIP WG 11.3 International Conference on Data and Applications Security (DBSEC), Cambridge, UK*, pages 133–146, 2002.

Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 369–383, May 2008.

Philip Wadler and Robert Bruce Findler. Well-typed programs cant be blamed. In *Workshop on Scheme and Functional Programming, Freiburg, Germany*, pages 15–26, 2007.

Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, GA*, pages 164–173, 2007.

A Listing of Wiki.boom

As described in Section 5, there are several reasonable policies that one can use in updateable security views over MediaWiki documents. Rather than picking a single policy, I have implemented a collection of function that can be instantiated to obtain lenses with any combination of the policies. Formally, all of the lens definitions are parameterized on these choices:

```
type align = Before | After | Positional
type hide = Hide | Redact
type nest = Nest | Flatten
```

To specify the of data at each level of structure, the programmer specifies one policy for each level. For example, the expression

```
let ahn = (after,hide,nest) in
document (ahn,ahn,ahn)
```

builds a lens that handles documents where, at each level of structure (in our simplified format: sections, paragraphs, lists) confidential data appears after public data, the *get* function hides confidential data, and the handling of confidential data follows the nesting structure of the document.

```
(*************************************************************)
(* The Harmony Project                                       *)
(* harmony@lists.seas.upenn.edu                              *)
(* Copyright (C) 2007-2008                                   *)
(* J. Nathan Foster and Benjamin C. Pierce                   *)
(*************************************************************)
(* wiki.boom                                                 *)
(* Fine-grained security views of MediaWiki documents.       *)
(*************************************************************)

module Wiki =

(* ----- regular expressions ----- *)
let NL : regexp = [\n]
let SPACE : regexp = [ ]
let STAR : regexp = [*]
let PCT : regexp = [%]
let NONNL : regexp = [^\n]
let DNL : regexp = NL{2}
let DEQ : regexp = [=]{2}
let DPCT : regexp = [%]{2}
let CONFIDENTIAL : regexp = /%% CONFIDENTIAL %%/
let HEADER : regexp = [^=\n%]+
let not (X:regexp) : regexp =
  ANY - containing X
let PARAGRAPH : regexp =
  let R : regexp = [^*%=\n] in
  R | (R . (not (NL . [=%*\n])) . R)

(* ----- configuration ----- *)
```

```
type align = Before | After | Positional
type hide = Hide | Redact
type nest = Nest | Flatten

let ahn = (After,Hide,Nest)
let ahf = (After,Hide,Flatten)
let arn = (After,Redact,Nest)
let bhn = (Before,Hide,Nest)
let phn = (Positional,Hide,Nest)

let red_config = (arn,arn,arn)
let std_config = (ahn,ahn,ahn)
let flat_config = (ahf,ahf,ahf)


(* ----- helper functions ----- *)
let iter_with_sep (l:lens) (s:lens) =
  l . (s . l)*

let key_or_copy (a:align) : regexp -> lens =
  match a with
    | Positional -> copy
    | _ -> key : (regexp -> lens)

let del_or_copy (h:hide) : regexp -> lens =
  match h with
    | Hide -> del
    | Redact -> copy : (regexp -> lens)

let secret_aux (h:hide) (R:regexp) (red:string)
               (def:string) (SEP:regexp) : lens =
  match h with
    | Hide -> del (R . (SEP . R)* )
    | Redact ->
        begin
          let l = default (R <-> red) def in
          iter_with_sep l (copy SEP)
        end : lens

let sequence_aux (a:align) (h:hide) (n:nest) (pub:lens)
                 (sec:lens) (SEP:regexp) (tag:string) : lens =
  let dsep : lens = del_or_copy h SEP in
  let ksep : lens = key_or_copy a SEP in
  let pub_sec : lens = match a with
    | Before -> sec . dsep . pub
    | _      -> pub . dsep . sec : lens in
  let chunk : lens = pub || pub_sec in
  let m : lens = match n with
    | Nest -> <~{1.0} chunk >
    | Flatten -> smatch "1.0" tag chunk : lens in
  let ms : lens = iter_with_sep m ksep in
  (match a with
     | Before -> ms . (copy "" || (dsep . sec))
     | _      -> (copy "" || (sec . dsep)) . ms) : lens

(* ------ itemized lists ------ *)
```

```
let public_list_elt (conf: (align * hide * nest) *
                           (align * hide * nest) *
                           (align * hide * nest)) : lens =
  let ((a,_,_),_,_) = conf in
  key_or_copy a (STAR . SPACE . NONNL+)

let secret_list_elts (conf:(align * hide * nest) *
                           (align * hide * nest) *
                           (align * hide * nest)) : lens =
  let ((_,h,_),_,_) = conf in
  secret_aux h (PCT . SPACE . NONNL+)
    "% REDACTED"
    "% INSERTED CONFIDENTIAL"
    NL

let list (conf: (align * hide * nest) *
                (align * hide * nest) *
                (align * hide * nest)) : lens =
  let ((a,h,n),_,_) = conf in
  sequence_aux a h n
    (public_list_elt conf)
    (secret_list_elts conf)
    NL "l"

let list_std : lens =
  list std_config

let list_bhn : lens =
  list (bhn,ahn,ahn)

let list_phn : lens =
  list (phn,ahn,ahn)

let list_arn : lens =
  list red_config

(* ----- paragraphs ----- *)
let public_paragraph (conf: (align * hide * nest) *
                            (align * hide * nest) *
                            (align * hide * nest)) : lens =
  let (_,(a,h,n),_) = conf in
  let p : lens = key_or_copy a PARAGRAPH in
  let l : lens = list conf in
  let nl : lens = key_or_copy a NL in
  let pl : lens = p . nl . l in
  p | l | (pl . nl)* . pl . (nl . p)?

let secret_paragraphs (conf: (align * hide * nest) *
                             (align * hide * nest) *
                             (align * hide * nest)) : lens =
  let (_,(_,h,_),_) = conf in
    secret_aux h (CONFIDENTIAL . NL . ctype (public_paragraph conf))
    "%% REDACTED %%"
    "%% CONFIDENTIAL %%\nINSERTED PARAGRAPH."
    DNL
```

```
let paragraphs (conf: (align * hide * nest) *
                      (align * hide * nest) *
                      (align * hide * nest)) : lens =
  let (_,(a,h,n),_) = conf in
  sequence_aux a h n
    (public_paragraph conf)
    (secret_paragraphs conf)
    DNL "p"

let paragraphs_std = paragraphs std_config

(* ------ sections ----- *)
let public_section (conf: (align * hide * nest) *
                          (align * hide * nest) *
                          (align * hide * nest)) : lens =
  let (_,(_,_,_),(a,_,_)) = conf in
  let dnl = key_or_copy a DNL in
  let h : lens = key_or_copy a (DEQ . HEADER . DEQ . NL) in
  let p = paragraphs conf in
  h . forgetkey (paragraphs conf)

let secret_sections (conf: (align * hide * nest) *
                           (align * hide * nest) *
                           (align * hide * nest)) : lens =
  let (_,_,(_,h,_)) = conf in
    secret_aux h (CONFIDENTIAL . NL . ctype (public_section conf))
      "==%% REDACTED %%=="
      "%% CONFIDENTIAL %%\n==INSERTED SECTION==\nINSERTED PARAGRAPH."
      NL

let sections (conf: (align * hide * nest) *
                    (align * hide * nest) *
                    (align * hide * nest)) : lens =
  let (_,_,(a,h,n)) = conf in
  sequence_aux a h n
    (public_section conf)
    (secret_sections conf)
    NL "s"

(* ----- documents ----- *)
let document (conf: (align * hide * nest) *
                    (align * hide * nest) *
                    (align * hide * nest)) : lens =
  sections conf

let document_std : lens =
  document std_config

let document_red : lens =
  document red_config

let document_flat : lens =
  document (ahf,ahf,ahf)
```