Dissertation

Penn
UNIVERSITY of PENNSYLVANIA

Computer & Information Science

BIDIRECTIONAL PROGRAMMING LANGUAGES

John Nathan Foster

Technical Report MS-CIS-10-08
Department of Computer & Information Science
University of Pennsylvania

March 13, 2010

Abstract

The need to edit data through a view arises in a host of applications across many different areas of computing. Unfortunately, few existing systems provide support for updatable views. In practice, when they are needed, updatable views are usually implemented using two separate programs: one to compute the view from the source and another to handle updates. This rudimentary design is tedious for programmers, difficult to reason about, and a nightmare to maintain.

This dissertation describes bidirectional programming languages, which provide an elegant mechanism for describing updatable views. Unlike programs written in an ordinary language, which only work in one direction, programs written in a bidirectional language can be run both forwards and backwards: from left to right, they describe functions that map sources to views, and from right to left, they describe functions that map updated views back to updated sources. Besides eliminating redundancy, these languages can be designed to ensure correctness, guaranteeing by construction that the two functions work well together.

Starting from the foundations, we identify a general semantic space of well-behaved bidirectional transformations called lenses. Then, building on this foundation, we describe a specific language for defining lenses for strings with syntax based on the regular operators. We also present extensions to the basic framework that address the complications that arise when lenses are used to manipulate sources containing inessential, ordered, and confidential data, and we describe the implementation of these features in the Boomerang language.

BIDIRECTIONAL PROGRAMMING LANGUAGES

John Nathan Foster

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2009

Supervisor of Dissertation
Benjamin C. Pierce

Graduate Group Chairperson
Jianbo Shi

Dissertation Committee
Zachary G. Ives
Val Tannen
Philip Wadler
Steve Zdancewic

This report was typeset in Sabon by the author using the LaTeX document processing system.

Preface

This report is a revised version of my PhD dissertation. Its technical content is essentially the same as the submitted version but several aspects of its presentation have been improved. Specifically, an early chapter fixing notation has been eliminated, with the material folded into the other chapters, and Chapter 4 has been revised significantly. Also, examples have been added, proofs streamlined, typos fixed, and writing polished throughout.

Philadelphia, PA
March 13, 2010

Acknowledgments

Some dissertation projects go like this: the student arrives on campus, the adviser sets a topic, the student disappears into the library, and emerges several years later with the finished product. Mine, thankfully, was nothing like this. The material described in this dissertation is the result of an extended collaboration with Benjamin Pierce and Alan Schmitt, with additional contributions from Davi Barbosa, Aaron Bohannon, Julien Cretin, Malo Deniélou, Michael Greenberg, Adam Magee, Stéphane Lescuyer, Danny Puller, Alexandre Pilkiewicz, and Steve Zdancewic. I am lucky to count these talented individuals as colleagues and friends and I gratefully acknowledge their contributions to this dissertation.

Throughout my time at Penn my adviser Benjamin Pierce, has always been available to brainstorm, crank through a proof, hack on a piece of code, or polish some writing. Thank you for setting the bar high, and for encouraging, nurturing, and providing help along the way.

Zack Ives, Val Tannen, Phil Wadler, and Steve Zdancewic all graciously agreed to serve on my dissertation committee. I am grateful for the time they spent reading my proposal document and this dissertation. Their many insightful comments and helpful suggestions have improved it in numerous ways.

Kim Bruce gave me my first research experience as an undergraduate nearly ten years ago and has been a mentor ever since. Alan Schmitt helped me navigate the transition to graduate school and has continued to be a collaborator and close friend. Alan also hosted me during an enjoyable visit to INRIA Rhône-Alpes in the spring of 2006. Ravi Konuru, Jérôme Siméon, and Lionel Villard were my mentors during a productive internship at IBM Research in the summer of 2007.

To the members of Penn's PLClub and Database Groups, thank you for creating a friendly and stimulating research environment. To my close friends in the department, Colin Blundell, TJ Green, Greg Karvounarakis, Micah Sherr, Nick Taylor, and Svilen Mihaylov, thank you for all the lunches, coffee breaks, and pleasant distractions from work. Thank you to my running partners, Adam Aviv, John Blitzer, Pavol Černý, and Jeff Vaughan for many miles of good conversation. To Dimitrios Vytiniotis, it's been... OK :-) To Mike Felker and Gail Shannon, thank you for solving so many bureaucratic puzzles.

To my family, especially Clare, Mollie, Tita Mary, Ahmad, my parents, stepparents, and grandparents, thank you for keeping me grounded and always believing in me.

To Sara, thank you for reminding me how fun life can be. I'll meet you for coffee anytime.

Work supported in part by an National Science Foundation Graduate Research Fellowship and grants CPA-0429836 *Harmony: The Art of Reconciliation*, IIS-0534592 *Linguistic Foundations for XML View Update*, and CT-T-0716469 *Manifest Security*. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of the National Science Foundation.

Contents

Chapter 1

Introduction

Most programs work in only one direction, from input to output. Indeed, as Baker observed when he wrote “the S combinator cheerfully copies ... the K combinator knowingly kills” [Bak92], the very fundamentals of computation often seem intrinsically unidirectional. However, the world is full of situations where after computing the initial output of a program we need to be able to modify it and then “compute backwards” to find a correspondingly modified input. There are numerous examples across many different areas of computing where these *bidirectional transformations* are needed:

- **Synchronization:** bidirectional transformations bridge the gap between replicas in heterogeneous formats [BMS08, KH06, FGK^{+}07].
- **Data Management:** bidirectional transformations provide mechanisms for propagating updates to views [FGM^{+}07, BVP06, BS81, DB82] and for transforming data from one schema to another in data exchange [MHH^{+}01] and schema evolution [BCPV07].
- **Software Engineering:** bidirectional transformations provide a way to maintain the consistency of software models and the underlying source code [Sch95, Ste07, XLH^{+}07].
- **Security:** bidirectional transformations enable fine-grained data sharing of documents with a mixture of public and private information [FPZ09].
- **Serialization:** bidirectional transformations translate between the on-disk and in-memory representations of ad hoc [FG05] and binary data [Ken04, Ege05].
- **Systems Administration:** bidirectional transformations map between flat, low-level representations and more structured representations of operating system configurations [Lut08].
- **Programming Languages:** bidirectional transformations handle boxing and unboxing of run-time values [Ben05, Ram03] and they map between programs written in different high-level languages [EG07].
- **User Interfaces:** bidirectional transformations provide convenient interfaces for manipulating complex documents [HMT08] and can be used to maintain the consistency of graphical user interface elements [Mee98, GK07].

Unfortunately, although the need for bidirectional transformations is ubiquitous, the linguistic technology for defining them is embarrassingly primitive. Most of the applications listed above are implemented using two separate functions—one to transform inputs to outputs and another to map outputs back to inputs—a rudimentary design that is tedious to for programmers, difficult to reason about, and a nightmare to maintain.

This dissertation proposes a different approach: languages in which every program can be run both forwards and backwards—from left to right as a function mapping inputs to outputs and from right to left as a function that propagates updated outputs back to updated inputs. Our thesis is that these *bidirectional programming languages* are an effective and elegant mechanism for describing bidirectional transformations.

1.1 The View Update Problem

Let us start by exploring some of the fundamental issues related to bidirectional transformations in the context of databases, an area where they have been extensively studied because of their close connection to the classic *view update problem*. Suppose that s is a source database, q is a query, and $v = q(s)$ is the view that results from evaluating q on s. (We will adopt this terminology throughout this dissertation, referring to inputs as "sources" and outputs as "views".) The view update problem is as follows: given an update u that transforms the view from v to v', calculate a source update t—the "translation" of u—that transforms s to s' and makes the following diagram commute.

$$\begin{tikzcd}
s \arrow[r, "q" description] \arrow[d, dashed, "t" description] & v \arrow[d, "u" description] \\
s' \arrow[r, "q" description] & v'
\end{tikzcd}$$

Unfortunately, despite years of study, this problem remains largely unsolved. The reason it has proven so challenging is that, in general, the view update u does not uniquely determine a source update t. For example, when q is not injective, some updates to the view have many corresponding source updates. It is possible to impose additional constraints on the problem to guide the choice of an update—e.g., requiring that t have "minimal side effects" on the source—but when the query and schema languages are sufficiently expressive, calculating updates that satisfy these additional constraints is intractable [BKT02]. Even worse, when q is not surjective, some updates to the view yield structures that are outside of the range of the query! It is not hard to see that it is impossible to propagate these update back to the source—there is no t that makes the diagram commute. Some systems reject these "untranslatable" updates, but doing this makes views "leaky abstractions"—it adds hidden constraints on how the view may be updated that are only revealed when user attempts to propagate an update back to the source.

Because of these difficulties, views in relational systems are generally read-only (except for views defined by very simple queries). In situations where updatable views are needed, programmers have to rely a variant of the rudimentary mechanism described above. They define a separate procedure called a *trigger* and instruct the system to execute this procedure whenever the view is modified. Triggers are arbitrary programs, so they can be used to implement every reasonable policy for propagating updates, but they are not a very attractive solution. For one thing, to check that a trigger correctly implements an updatable view, the programmer must perform intricate, manual reasoning about the way that the trigger behaves in tandem with the query. Moreover, the trigger and query will necessarily be redundant—each will embody (at least) the correspondence between the source and view—so they will be difficult to maintain when the schemas evolve.

1.2 Bidirectional Programming Languages

A better approach is to define the view and its associated update policy together. *Bidirectional programming languages* are organized around this idea: every program denotes both a function that computes a view as well as a function that propagates updated views back to updated sources. This eliminates the need to write—and maintain!—two separate programs, as well as the need to do any manual reasoning about correctness because the language can be designed to guarantee it. The main challenge in the design of a bidirectional language 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 can be verified using straightforward, compositional, and, ultimately, mechanizable checks.

To illustrate the tradeoffs between these two approaches, consider a simple example. Suppose that the source is an XML document representing the names, dates, and nationalities of a collection of classical music composers

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <dates>1865-1956</dates>
    <nationality>Finnish</nationality>
  </composer>
  <composer>
    <name>Aaron Copland</name>
    <dates>1910-1990</dates>
    <nationality>American</nationality>
  </composer>
  <composer>
    <name>Benjamin Briten</name>
    <dates>1913-1976</dates>
    <nationality>English</nationality>
  </composer>
</composers>
```

and the view is a list of lines of ASCII text representing the name and dates of each composer:

```
Jean Sibelius, 1865-1956
Aaron Copland, 1910-1990
Benjamin Briten, 1913-1976
```

After computing the initial view, we might want to edit it in some way—e.g., correcting the error in Sibelius's death date and the misspelling in Britten's name

```
Jean Sibelius, 1865-1957
Aaron Copland, 1910-1990
Benjamin Britten, 1913-1976
```

and push the changes back into the original XML format:

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <dates>1865-1957</dates>
    <nationality>Finnish</nationality>
  </composer>
  <composer>
    <name>Aaron Copland</name>
    <dates>1910-1990</dates>
    <nationality>American</nationality>
  </composer>
  <composer>
    <name>Benjamin Britten</name>
    <dates>1913-1976</dates>
    <nationality>English</nationality>
  </composer>
</composers>
```

Here is a bidirectional program, written in the language of *basic lenses* described in Chapter 2, that describes both transformations, from XML to ASCII and from ASCII to XML:

```
(* regular expressions *)
let WHITESPACE : regexp = [\n\t ]
let ALPHA : regexp = [A-Za-z]+
let YEAR : regexp = [0-9]{4}

(* helper functions *)
let xml_elt (t:string) (l:lens) : lens =
    del WHITESPACE*
  . del ("<" . t . ">")
  . l
  . del WHITESPACE*
  . del ("</" . t . ">")

let del_default (R:regexp) (u:string) : lens =
  default (del R) (fun (x:string) -> u)

(* helper lens *)
let composer : lens =
  xml_elt "composer"
    ( xml_elt "name" (copy (ALPHA . " " . ALPHA) )
    . ins ", "
    . xml_elt "dates"  (copy (YEAR . "-" . YEAR) )
    . xml_elt "nationality"
        (del_default ALPHA "Unknown") )

(* main lens *)
let composers : lens =
  xml_elt "composers"
    (copy "" | (composer . (ins "\n" . composer)* ) )
```

In the forward direction, this program can be read an ordinary function on strings. The first few lines define regular expressions (using standard POSIX notation) for whitespace, alphabetic strings, and years. The helper function `xml_elt` defined next takes a string `t` and a lens `l` as arguments and returns a lens that processes an XML element named `t`. It first removes all of the XML formatting for the element and then processes the children of the element using the lens `l`. The concatenation operator (`.`) combines lenses in the obvious way. The `del_default` helper takes a regular expression `R` and a string `u` as arguments and deletes a string matching `R`. The string `u` is used a default value in the reverse direction. The composer lens defined next instantiates xml_elt several times to construct a lens that handles an XML element for a single composer. It copies the name of the composer, inserts a comma and a space into the view, copies the birth and death dates, and deletes the nationality. The composers lens processes a sequence of composer elements. It uses union (|) and Kleene star (*) to iterate the composer lens over the list of composers, inserting a newline character between each line in the view.

Because this is a bidirectional program, we can also run it in the other direction. The backwards function propagates the names and dates in the view and restores the nationalities from the original source. The details of how this works are not important for now (see Chapter 2 for precise definitions). The key point is that we can use the same lens to combine a modified ASCII view with the original XML source to obtain an updated XML source.

A natural question to ask at this point is whether bidirectional languages are worth the trouble. After all, how hard could it be to just write the two functions as separate programs in a general-purpose language? To explore this idea (and ultimately reject it), let us consider separate implementations of the forward and backward transformations that make up the composers lens in the OCaml language. The composers_forward transformation can be written as follows:

```
let composers_forward src =
  let src_seq = children (parse_string src) in
  List.fold_left
    (fun acc src_i ->
       let [xname;xdates;_] = children src_i in
       let sep = if acc <> "" then "\n" else "" in
       (acc ^ sep ^ pcdata xname ^ ", " ^ pcdata xdates))
    ""
    src_seq
```

The parse_string, children, and pcdata functions come from a library for manipulating XML structures. It parses the source src into an XML tree and then folds down the sequence of composer elements, extracting the name and dates from each element and adding them to the view.

The corresponding composers_backward function is written in OCaml as follows:

```
let composers_backward view src =
  let view_seq = split "\n" view in
  let src_xseq = children (parse_string src) in
  let rec aux acc view_seq src_xseq = match view_seq with
    | [] ->
      to_string (element "composers" (List.rev acc))
    | view_h::view_t ->
      let [name;dates] = split ", " view_h in
      let xnationality,src_xt = match src_xseq with
        | [] ->
          ("Unknown",[])
        | src_xh::src_xt ->
          (List.nth (children src_xh) 2,src_xt) in
      let xname = pcdata_element "name" name in
      let xdates = pcdata_element "dates" dates in
      let xseq = [xname; xdates; xnationality] in
      let xcomposer = element "composer" xseq in
      aux (xcomposer::acc) view_t src_xt in
    aux [] view_seq src_xseq
```

The functions pcdata_element, element, and to_string come from the same XML library that was used to define the forward transformation. It takes the updated view view and the original source src as arguments and weaves them together, propagating the names and dates in view and restoring the nationalities from src.

One advantage of the lens program compared to the OCaml version is parsimony—we only need to write one program instead of two. Another is that we can prove—automatically!—that the lens program correctly implements an updatable view while the OCaml version would be easy to get wrong. The type system for lenses presented in Chapter 2 ensures a number of natural well-behavedness properties—e.g., that the two functions are totally defined functions on the sets of strings representing XML sources and ASCII views, and that composing the forward and backward transformations in either order yields the identity function. In contrast, to verify that the two OCaml programs implement the view correctly, we would need to check these properties by hand. We can do this, of course, but it would involve a *lot* of manual pencil-and-paper reasoning about fairly low-level properties—e.g., we would need to check that lines concatenated in the aux loop of the composers_forward function are split the same way in composers_backward, and so on.

Perhaps the most significant advantage of the lens program is that it is much easier to maintain. Suppose that we decided to change the representation of dates in the XML source from

```
<dates>1910-1990</dates>
```

to:

```
<born>1910</born>
<died>1990</died>
```

Updating the lens program to accommodate this change only requires a small change in the body of the composer lens:

```
let composer : lens =
  xml_elt "composer"
    ( xml_elt "name" (copy (ALPHA . " " . ALPHA) )
    . ins (", ")
    . xml_elt "born" (copy YEAR)
    . xml_elt "died" (copy YEAR)
    . xml_elt "nationality"
        (del_default ALPHA "Unknown") )
```

The transformations denoted by the lens both reflect the change and the typechecker verifies the well-behavedness properties automatically.

Updating the OCaml program, however, requires multiple, coordinated changes to both functions—something that is very easy to get wrong! Here is the revised OCaml definition of the forward function

```
let composers_forward src =
  let src_seq = children (parse_string src) in
  List.fold_left
    (fun acc src_i ->
       let [xname;xborn;xdied;_] = children src_i in
       let sep = if acc <> "" then "\n" else "" in
       (acc ^ sep ^ pcdata xname ^ ", " ^
        pcdata xborn ^ "-" ^ pcdata xdied))
    ""
    src_seq
```

and here is the revised definition of the backward function:

```
let composers_backward view src =
  let view_seq = split "\n" view in
  let src_xseq = children (parse_string src) in
  let rec aux acc view_seq src_xseq = match view_seq with
    | [] ->
      to_string (element "composers" (List.rev acc))
    | view_h::view_t ->
      let [name;dates] = split ", " view_h in
      let [born;died] = split "-" dates in
      let xnationality,src_xt = match src_xseq with
        | [] ->
          ("Unknown",[])
        | src_xh::xsrc_xt ->
          (List.nth (children src_xh) 2,src_xt) in
      let xname = pcdata_element "name" name in
      let xborn = pcdata_element "born" born in
      let xdied = pcdata_element "died" died in
      let xseq = [xname; xborn; xdied; xnationality] in
      let xcomposer = element "composer" xseq in
      aux (xcomposer::acc) view_t src_xt in
  aux [] view_seq src_xseq
```

After making these changes, we would then need to update and re-verify the well-behavedness proof by hand. Thus, even for this almost trivial example, the solution written in the bidirectional language is a much more attractive option.

1.3 Goals and Contributions

The goal of this dissertation is to demonstrate that bidirectional languages are an effective way of defining updatable views. Its contributions are divided between three broad areas:

1. **Foundations:** we identify a mathematical space of bidirectional transformations characterized by natural behavioral laws that govern the handling of data in the source and view. These well-behaved transformations, called *lenses*, provide the semantic foundation for the rest of this dissertation.

2. **Language Design:** we develop a core language of lens primitives for strings and a type system for this language that guarantees well behavedness. We also study extensions that address the complications that come up when lenses are used to manipulate structures containing inessential, ordered, and confidential data.

3. **Implementation:** we describe the design and implementation of a full-blown functional programming language based on our core lens primitives. This language, called Boomerang, includes a variety of features designed to make it easy for programmers to construct large programs.

The rest of this chapter expands on each of these contributions in detail.

Foundations

Many systems that use updatable views take an informal approach to correctness. In contrast, a bidirectional language can be designed to guarantee correctness by construction—indeed, this is one of their main advantages. However, before we can talk about what it means for a bidirectional transformation to be correct, we need to precisely characterize the properties we want them to have. The first contribution of this dissertation is a semantic framework of well-behaved bidirectional transformations called *lenses*. This framework organizes the whole area of transformations that—in some way—implement updatable views and lays a solid foundation for designing bidirectional languages. Chapter 2 defines the framework of *basic lenses* which are transformations obeying natural behavioral laws similar to the conditions on view update translators that have been proposed in databases. Chapter 3 defines *quotient lenses* which relax the basic lens laws by allowing certain specified portions of the source and view to be treated as "inessential." This generalization is motivated by experience developing lenses for real-world data formats which often contain unimportant details such as whitespace. Chapter 4 describes *resourceful lenses* which address the critical issue of alignment that comes up when the source and view are ordered structures. Resourceful lenses come equipped with new mechanisms for computing and using alignments between the pieces of the view and the pieces of the underlying source. They also obey new properties which ensure that they use alignment information correctly. Finally, Chapter 5 describes *secure lenses*, which extend basic lenses with additional guarantees about the confidentiality and integrity of data in the source and view.

Language Design

In order to interpret programs bidirectionally, we need to change the way that we write programs—some constructs that make sense in unidirectional languages do not make sense as lenses. The second contribution of this dissertation is a design for a core language of lenses with natural, compositional syntax and a type system that guarantees the lens laws. For simplicity, we focus our efforts in this area on languages for manipulating for strings rather than richer structures such as trees or complex values. However, even though strings are simple structures, they still expose many fundamental issues. Additionally, because there is a lot of string data in the world—textual databases, structured documents, scientific data, simple XML, and many different kinds of ad hoc data—having a language for developing lenses on strings is actually quite useful. Chapter 2 introduces the operators studied throughout this dissertation. It includes generic operators (identity, constant, sequential composition), the regular operators (union, concatenation, Kleene star), and some additional operators that we have found useful in applications (filter, swap, merge, etc.). Types play a central role in this language. Some operators only make sense as lenses when certain side conditions are met and we use types to express and check these conditions. To ensure that typechecking can be automated, we use regular languages as types. Regular languages balance the tradeoffs between precision and decidability—they describe data formats at a high level of detail, but all of the operations on regular languages we need for typechecking are decidable. The later chapters in this dissertation all describe extensions of, or refinements to, this basic language of string lenses: Chapter 3 extends it with new constructs (canonizers and quotient operators) for dealing with ignorable data. Chapter 4 adds new features (chunks, keys, and thresholds) for aligning data, and Chapter 5 extends the type system of the language with a more refined analysis that tracks information flow.

Implementation

The third contribution of this dissertation is an implementation of our ideas in a full-blown functional programming language called Boomerang. This language, described in Chapter 6, demonstrates our thesis that bidirectional languages are an effective way of defining updatable views. Boomerang includes a number of features designed to make it easier for programmers to develop large lens programs including first-order functions, an expressive type system, modules, unit tests, etc. Additionally, Boomerang has several features specifically designed for describing string transformations: built-in regular expressions, overloading, subtyping, and syntax for lenses based on grammars. We have implemented a full working prototype of Boomerang and used it to develop lenses for a number of real-world data formats. Although this prototype is not yet industrial strength (in particular, we have not made any serious attempts to optimize for performance) it is robust enough to handle examples of realistic size—e.g., our largest lens is a 4KLoc program that computes views over XML documents representing scientific databases. Our design for Boomerang has recently been adopted in industry. Augeas [Lut08], a tool developed by RedHat, Inc. for managing operating system configurations, is directly based on Boomerang (it also includes some constructs for building trees). Augeas programmers have developed lenses for nearly all of the standard configuration file formats usually found in a Linux system. This independent application of our ideas provides additional evidence that the idea of lenses is robust and that our design for Boomerang is sensible.

1.4 Acknowledgments

This dissertation describes work performed in cooperation with many different colleagues and portions of it are based on papers written in collaboration with them. In particular, the semantics of basic lenses described in Chapter 2 is based on an article by Foster, Greenwald, Moore, Pierce, and Schmitt [FGM$^+$07] and the string lenses presented in that chapter are based on material from a paper by Bohannon, Foster, Pierce, Pilkiewicz, and Schmitt [BFP$^+$08]. Chapter 3 is a revised version of a paper by Foster, Pilkiewicz, and Pierce [FPP08]. Chapter 4 describes recent work with Barbosa, Cretin, Greenberg, and Pierce [BCF$^+$10]. Chapter 5 is based on a paper by Foster, Pierce, and Zdancewic [FPZ09]. The design and implementation of Boomerang described in Chapter 6 was done in collaboration with Davi Barbosa, Julien Cretin, Michael Greenberg, Alexandre Pilkiewicz, Benjamin Pierce, and Alan Schmitt. Boomerang's precise type system was designed and implemented in cooperation with Michael Greenberg. Its grammar notation was designed with help from Adam Magee and Danny Puller. Finally, Chapter 7 is an expanded version of the survey originally published in the paper by Foster, Greenwald, Moore, Pierce, and Schmitt [FGM$^+$07]. Of course, any errors are mine.

Chapter 2

Basic Lenses

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

—Henry David Thoreau

This chapter presents the semantic space of basic lens and describes a core language of basic lenses for strings. The main constructs in this language are based on finite state string transducers [Ber79]. Although its computational power is somewhat limited, this language is a natural formalism that is capable of expressing many useful transformations on strings. It also cleanly illustrates many of the design choices and tradeoffs that arise in richer settings.

We begin the chapter by defining the semantic framework of basic lenses in Section 2.1. We develop some elementary properties of basic lenses in Section 2.2. Section 2.3 presents syntax and typing rules for a core set of string lenses combinators. We conclude the chapter in Section 2.4.

2.1 Semantics

Before we can delve into language design, we need a framework in which we can say precisely when a bidirectional transformation implements an updatable view correctly and when it does not. This section presents a mathematical space of well-behaved bidirectional transformations called *basic lenses* that captures an intuitive notion of correctness. Although we will primarily focus on lenses for strings in this dissertation, the semantics of basic lenses can be formulated over arbitrary structures. Thus, in this section, we work in a completely generic setting, parameterizing all of our definitions on a universe $\mathcal{U}$ of objects. Later in the chapter, we will instantiate the framework by picking $\mathcal{U}$ to be the set of strings.

2.1.1 Definition [Basic Lens]: Fix a universe $\mathcal{U}$ of objects and let $S \subseteq \mathcal{U}$ and $V \subseteq \mathcal{U}$ be sets of sources and views. A *basic lens* l from S to V comprises three total functions

$$\begin{array}{rcl} l.\mathit{get} & \in & S \rightarrow V \\ l.\mathit{put} & \in & V \rightarrow S \rightarrow S \\ l.\mathit{create} & \in & V \rightarrow S \end{array}$$

obeying the following laws for every $s \in S$ and $v \in V$:

$$l.\mathit{get}\ (l.\mathit{put}\ v\ s) = v \qquad (\textsc{PutGet})$$

$$l.\mathit{get}\ (l.\mathit{create}\ v) = v \qquad (\textsc{CreateGet})$$

$$l.\mathit{put}\ (l.\mathit{get}\ s)\ s = s \qquad (\textsc{GetPut})$$

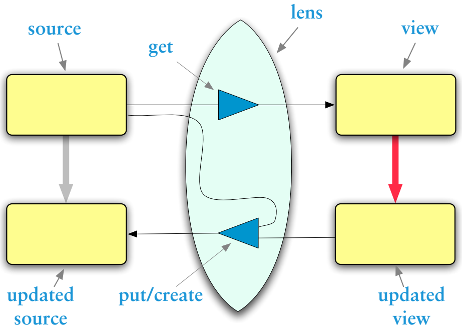


Figure 2.1: Basic Lens Architecture

The set of all basic lenses mapping between S and V is written $S \Longleftrightarrow V$.

The intuition behind the names *get*, *put* and *create* is that the *get* function "lifts" a view out of a source structure, while *put* "pushes down" an updated view into the original source, yielding a new source that reflects the modifications made to the view. We will often say "put v into s (using l)" rather than "apply l's *put* function to v and s". Note that *put* takes the original source as an argument because, in general, the *get* function may discard some of the source information in computing the view. It weaves its two arguments together, propagating the information in the new view and restoring any information discarded from the old source. The *create* function handles the special case where we need to construct a source from a view, but we have no source to use as the original. It manufactures a new source "from scratch," filling in any missing information with defaults. Figure 2.1 depicts the components of a lens graphically.

Broadly speaking, lenses are designed to guarantee three main properties:

1. Lenses implement robust abstractions. Users can make arbitrary modifications to the view without having to consider whether their changes are consistent with the underlying source.

2. Lenses propagate view updates "exactly" to the source.

3. When possible, lenses preserve any source information that is not reflected in the view.

Formally, these properties are ensured by the requirement that *put* be a total function and by the "round-tripping" laws in Definition 2.1.1, which govern the handling of data as it is mapped between the source and view. The next few paragraphs describe how these formal conditions guarantee the informal properties enumerated above and discuss their benefits and tradeoffs.

Robustness A fundamental choice in the design of a bidirectional language is whether it handles every update to the view or if it is allowed to treat certain updates as untranslatable. Hegner has coined the terms "closed views" and "open views" to describe these alternatives [Heg90].

Systems that allow updates to fail have enormous flexibility, since the decision about whether to propagate a given update to the view can be made dynamically, on a case-by-case basis. This allows these systems to support transformations that would not be valid as lenses. However, allowing updates to fail has a significant drawback: it makes views "leaky abstractions" of the source. Users cannot use a view like an ordinary data structures because whenever they modify the view, they need to consider the possibility that propagating the update will fail.

The *put* component of every lens is a total function. Totality is a simple but powerful condition which ensures that lenses are capable of doing something reasonable with every view and every source, even when the view has been edited dramatically. In the applications where we use lenses—e.g., in data synchronizers run in unsupervised modes of operation [FGK$^+$07]—it critical that the abstraction provided by the view be robust.

An interesting side effect of this choice is that, in practice, lens languages usually need to have very precise type systems. The only way that a lens can free itself from the obligation to handle a particular view is to exclude it from the set V mentioned in its type. Thus, lenses that manipulate the source and view in complicated ways typically need to have types that describe those structures at a correspondingly high level of precision.

Exact Translation Another fundamental consideration in the design of basic lenses is the set of conditions that govern how updates must be handled. Most systems require that updates to the view be translated "exactly" to the source—i.e., that the new source produced by *put* reflect all of the changes made to the view. In the lens framework, the PutGet and CreateGet laws guarantee this property. Formally, they stipulate that given an updated view and an old source, the *put* function must yield a new source that the *get* function maps back to the very same view, and similarly for the *create* function.

As an example of a transformation that does not obey PutGet and CreateGet (it does obey GetPut), let S be the set of strings over a finite alphabet Σ, let V be the set $\Sigma^* \times \mathbb{N}$ of pairs of strings and natural numbers, and define functions *get*, *put*, and *create* as follows:

$$\begin{array}{rcl} l.\mathit{get}\ u & = & (u, 0) \\ l.\mathit{put}\ (u', n)\ u & = & u' \\ l.\mathit{create}\ (u', n) & = & u' \end{array}$$

If we apply *put* to an updated view (*"abc"*$, 1)$ and an original source *"xyz"* it produces a new source *"abc"*. However, if we then map this source back to a view using *get* we obtain (*"abc"*$, 0)$, which is different than (*"abc"*$, 1)$. Intuitively, the reason that PutGet fails is that the *put* function does not propagate all of the information contained in the view back to the source. As a result, some updates to the view are lost when we run *put* followed by *get*. It is not hard to see that the PutGet law implies that *put* must be injective with respect to its first argument—see Lemma 2.2.1 (2) below.

Source Integrity A third consideration in the design of lenses concerns the handling of source data that is not exposed in the view. To the extent possible, we would like *put* to avoid making unnecessary changes to the underlying source. However, in many situations, to accurately reflect the modification made to the view, *put* needs to modify the source, including the hidden parts that are not reflected in the view.

It turns out that there are range of conditions one can impose to control the side effects that *put* can have on the source. Basic lenses include a simple condition, embodied in the GetPut law, which stipulates that the *put* function must restore the original source exactly whenever its arguments are a view v and a source s that generates the very same view.

As example of a transformation that does not obey GetPut (it does obey PutGet and CreateGet), let S be the set $\Sigma^* \times \mathbb{N}$ of pairs of strings and natural numbers, let V be the set Σ^* of strings, and define functions *get*, *put*, and *create* as follows:

$$\begin{array}{rcl} l.\mathit{get}\ (u, n) & = & u \\ l.\mathit{put}\ u'\ (u, n) & = & (u', 0) \\ l.\mathit{create}\ u' & = & (u', 0) \end{array}$$

If we use l to compute a view from (*"abc"*$, 1)$ and immediately put it back, we get (*"abc"*$, 0)$, which is different than the source we started with. The problem with this transformation is that the *put* function has extra side effects on the source—it sets the number to 0 even when the view has not been changed.

The GetPut law restricts the effects that *put* can have on the source by forcing it to have no effect at all whenever it is possible for it to do so without violating the other laws.

It is tempting to go a step further and require that *put* always have "minimal" side effects, and not only when the view has not been changed. Unfortunately, even stating this condition seems to require building a notion of what constitutes an update into the semantics—we need to be able to compare two updates to determine which one has a "smaller" effect. Lenses are designed to be agnostic to the way that updates are expressed—the *put* function takes the whole state of the updated view as input (in database terminology, a materialized view) rather than an explicit operation in an update language. This state-based architecture makes it easy to deploy lenses in a variety of different contexts since applications do not need to be retooled to manipulate views via special operations in an update language. It also facilitates using lenses with data in non-standard and ad hoc formats, which do not usually come equipped with canonical update languages. However, being state-based makes it difficult to impose conditions formulated in terms of updates. In particular, it leads us to only impose the GetPut law, which can be stated abstractly and without assuming a particular notion of update. Although GetPut only provides a relatively loose constraint on behavior, it is still a useful tool for designing lens primitives. We have used it many times to generate, test, and reject candidate lenses.

Another idea for ensuring the integrity of source data is to require that the *put* function preserve *all* of the information in the source that is not reflected in the view. This idea has been explored extensively in the database literature, where it is known as the *constant complement* condition [BS81]. The idea is that the source S should be isomorphic to $V \times C$, a product consisting of the view V and a "complement" C that contains all of the source information not reflected in the view. The *get* function uses the function witnessing the isomorphism in one direction to transform the source s into a pair (v, c) and then projects away c. The *put* function pairs up the new view v' with the old complement c and applies the other witness to the isomorphism to (v', c) to obtain the new source. As the *put* function is implemented by an injective function from $V \times C$ to S, it follows that *all* of the information contained in the complement is reflected in the new source—i.e., the complement is held constant. We can formulate a law that captures the essence of the constant complement condition by stipulating that the source obtained after doing two *put*s in a row must be the same as doing just the second:

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

Although this law does not mention complements, it forces *put* function to restore all of the hidden information in the underlying source—i.e., a complement—because the intermediate source produced by the first *put* must contain that information for it to be available for the second *put*. Unfortunately, requiring that every lens obey PutPut is a draconian condition that rules out many transformations that are indispensable in practice.

As an example, of a lens that does not obey PutPut, let S be the set $(\Sigma^* \times \mathbb{N})$ *list* of lists of pairs of strings and natural numbers, let V be the set Σ^* *list* of lists of strings, and define functions *get*, *put* and *create* as follows:

$$\begin{array}{rcl} l.\mathit{get}\ [(u_1, n_1), \ldots, (u_k, n_k)] & = & [u_1, \ldots, u_k] \\ l.\mathit{put}\ [u'_1, \ldots, u'_l]\ [(u_1, n_1), \ldots, (u_k, n_k)] & = & [(u'_1, n'_1), \ldots, (u'_l, n'_l)] \\ \text{where } n'_i = \begin{cases} n_i \text{ for } i \in \{1, \ldots, \min(l, k) \\ 0 \ \text{ for } i \in \{k+1, \ldots, l\} \end{cases} & & \\ l.\mathit{create}\ [u'_1, \ldots, u'_l] & = & [(u'_1, 0), \ldots, (u'_l, 0)] \end{array}$$

The *get* component of this lens takes a list of pairs of strings and numbers and projects away the numbers. The *put* component takes a view and a source and weaves them together, propagating the strings in the view and restoring the numbers from the source. To see why the PutPut law fails to hold in general, observe that when the list of strings in the view has fewer elements than the source list, the *put* function must discard some of the numbers in the source to satisfy PutGet. For example, putting [*"a"*, *"b"*] into $[(\text{“}a\text{”}, 1), (\text{“}b\text{”}, 2), (\text{“}c\text{”}, 3)]$ discards the 3 and yields $[(\text{“}a\text{”}, 1), (\text{“}b\text{”}, 2)]$. This accurately reflects the change made to the view, but if we *put* [*"a"*, *"b"*, *"c"*] into this intermediate source we obtain $[(\text{“}a\text{”}, 1), (\text{“}b\text{”}, 2), (\text{“}c\text{”}, 0)]$, which is different than $[(\text{“}a\text{”}, 1), (\text{“}b\text{”}, 2), (\text{“}c\text{”}, 3)]$, the source we would have obtained if we had put this view into the original source without doing the first *put*.

Although we do not require that every basic lens obey the PutPut law, we pay special attention to lenses that do, calling them *very well behaved*.

2.1.2 Definition [Very Well Behaved Lens]: A lens $l \in S \Longleftrightarrow V$ is *very well behaved* if and only if it obeys the PutPut law for all views v and v' in V and sources s in S.

Interestingly, the weaker integrity guarantee embodied in the GetPut law can be formulated as a special case of PutPut. The PutTwice law stipulates that the effect of doing two *put*s in a row using the same view must be the same as doing just doing one *put*:

$$l.put\ v\ (l.put\ v\ s) = l.put\ v\ s \qquad \text{(PutTwice)}$$

Every basic lens obeys the PutTwice law—see Lemma 2.2.2.

Another important class of lenses are those whose *put* functions do not use their source argument at all. We call such lenses *oblivious*.

2.1.3 Definition [Oblivious Lens]: A lens $l \in S \Longleftrightarrow V$ is *oblivious* if

$$l.put\ v\ s = l.put\ v\ s'$$

for all views v in V and sources s and s' in S.

The components of an oblivious lenses are all bijective functions. Every oblivious lens is trivially very well behaved, since the *put* function does not use its source argument.

2.2 Properties

Now we use the semantics of basic lenses to derive some simple properties. We establish a collection of elementary facts about basic lenses and use these facts to give an alternate characterization of lenses in terms of *put* functions.

2.2.1 Lemma: For every $l \in S \Longleftrightarrow V$, we have the following facts:

1. *l.get* and $(uncurry\ l.put)$ are surjective functions,[1]
2. *l.put* is *semi-injective* in the following sense: for all views v and v' in V and all sources s and s' in S, if $l.put\ v\ s = l.put\ v'\ s'$ then $v = v'$,
3. and *l.create* is an injective function.

Proof: Let $l \in S \Longleftrightarrow V$ be a basic lens. We prove each fact separately.

[1] Recall that when f is a function in $A \rightarrow B \rightarrow C$, the function $uncurry\ f$ is a function in $A \times B \rightarrow C$ defined as $uncurry\ f \triangleq \lambda(a,b) : (A \times B).\ f\ a\ b$. Note that the surjectivity of $uncurry\ l.put$ and $l.put$ are different conditions: the former means that for every $s \in S$ there exists a $v \in V$ and $s' \in S$ such that $l.put\ v\ s' = s$ while the latter means that for every $g \in S \rightarrow S$ there exists a $v \in V$ such that the partially-applied function $l.put\ v$ and g are equivalent.

1. Let v be a view in V and let $s' = (l.put\ v\ (l.create\ v))$ be the source obtained by putting v into $l.create\ v$. By the PutGet law for l we have $l.get\ s' = v$. As v was arbitrary, we conclude that $l.get$ is surjective.

   Similarly, let s be a source in S and let $v = l.get\ s$. By the GetPut law for l we have $l.put\ v\ s = s$. By the definition of *uncurry* we have $(uncurry\ l.put)\ (v, s) = s$. As s was arbitrary, we conclude that $uncurry\ l.put$ is surjective.

2. Let v and v' be views in V and let s and s' be sources in S such that $l.put\ v\ s = l.put\ v'\ s'$. We calculate as follows:

   $$\begin{aligned} v &= l.get\ (l.put\ v\ s) && \text{by PutGet for } l \\ &= l.get\ (l.put\ v'\ s') && \text{by assumption} \\ &= v' && \text{by PutGet for } l \end{aligned}$$

   That is, $v = v'$. We conclude that $l.put$ is semi-injective.

3. Let $v \in V$ and $v' \in V$ be views such that $l.create\ v = l.create\ v'$. We calculate as follows:

   $$\begin{aligned} v &= l.get\ (l.create\ v) && \text{by CreateGet for } l \\ &= l.get\ (l.create\ v') && \text{by assumption} \\ &= v' && \text{by CreateGet for } l \end{aligned}$$

   That is, $v = v'$. We conclude that $l.create$ is injective. □

Next we prove that every lens obeys the PutTwice law.

2.2.2 Lemma: Let l be a basic lens in $S \Longleftrightarrow V$. Then l obeys the PutTwice law.

Proof: Let $l \in S \Longleftrightarrow V$ be a lens, $v \in V$ a view, and $s \in S$ a source. We calculate as follows

$$\begin{aligned} & l.put\ v\ (l.put\ v\ s) \\ = {} & l.put\ (l.get\ (l.put\ v\ s))\ (l.put\ v\ s) && \text{by PutGet for } l \\ = {} & l.put\ v\ s && \text{by GetPut for } l \end{aligned}$$

and obtain the required equality. □

A natural question to ask is whether lenses are semantically complete—i.e., given a total and surjective function g from S to V does there exist a lens l that has g as its *get* component? We answer this question positively in the next lemma.

2.2.3 Lemma: For every total and surjective function $g \in S \rightarrow V$ there exists a basic lens l in $S \Longleftrightarrow V$ such that $l.get = g$.

Proof: Let $g \in S \rightarrow V$ be a total and surjective function from S onto V. For every $v \in V$, let $\widehat{v}$ denote an arbitrary element of S satisfying $g\ \widehat{v} = v$. As g is surjective, $\widehat{v}$ exists. We define the components of l as follows:

$$\begin{aligned} l.get &= g \\ l.put\ v\ s &= \begin{cases} s \text{ if } v = g\ s \\ \widehat{v} \text{ otherwise} \end{cases} \\ l.create\ v &= \widehat{v} \end{aligned}$$

By construction, we have $l.\mathit{get} = g$. We show that l is well behaved by proving each of the lens laws directly:

▶ **GetPut**: Let $s \in S$. We calculate as follows

$$\begin{array}{ll} & l.\mathit{put}\ (l.\mathit{get}\ s)\ s \\ = & l.\mathit{put}\ (g\ s)\ s & \text{by definition } l.\mathit{get} \\ = & s & \text{by definition } l.\mathit{put} \end{array}$$

and obtain the required equality.

▶ **PutGet**: Let $v \in V$ and $s \in S$. We calculate as follows

$$\begin{array}{lll} & l.\mathit{get}\ (l.\mathit{put}\ v\ s) & \\ = & \begin{cases} l.\mathit{get}\ s & \text{if } v = g\ s \\ l.\mathit{get}\ \widehat{v} & \text{otherwise} \end{cases} & \text{by definition } l.\mathit{put} \\ = & \begin{cases} g\ s & \text{if } v = g\ s \\ g\ \widehat{v} & \text{otherwise} \end{cases} & \text{by definition } l.\mathit{get} \\ = & v & \text{as either } v = g\ s \text{ by assumption} \\ & & \text{or } v = g\ \widehat{v} \text{ by definition } \widehat{v} \end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof of PutGet. □

The next lemma gives an analogous completeness result for the reverse direction.

2.2.4 Lemma: For every total and semi-injective function $p \in V \to S \to S$ such that $\mathit{uncurry}\ p$ is surjective and $p\ v\ (p\ v\ s) = p\ v\ s$ for every v in V and s in S, there exists a basic lens $l \in S \Longleftrightarrow V$ such that $l.\mathit{put} = p$.

Proof: Let $p \in V \to S \to S$ be a total and semi-injective function such that $\mathit{uncurry}\ p$ is surjective and $p\ v\ (p\ v\ s) = p\ v\ s$ for every view v in V and source s in S. We will prove that for every source $s \in S$ there is a unique view v such that $(p\ v\ s) = s$. Let s be a source in S.

First we demonstrate that there is at least one such v. By the assumption that $\mathit{uncurry}\ p$ is surjective, there exists a view $v \in V$ and source $s' \in S$ such that $p\ v\ s' = s$. We calculate as follows

$$\begin{array}{lll} & p\ v\ s & \\ = & p\ v\ (p\ v\ s') & \text{by } p\ v\ s' = s \\ = & p\ v\ s' & \text{by } p\ v\ (p\ v\ s') = p\ v\ s' \\ = & s & \text{by } p\ v\ s' = s \end{array}$$

and obtain the desired equality.

Next we show that v is unique. Let v' be a view in V satisfying $p\ v'\ s = s$. As p is semi-injective, we have $v' = v$. Hence, v is unique.

Using these facts, we define a lens l from p. For every source $s \in S$, let $\widehat{s} \in V$ denote the unique view $v \in V$ satisfying $p\ \widehat{s}\ s = s$. Define the components of l as follows:

$$\begin{array}{rcl} \mathit{get}\ s & = & \widehat{s} \\ \mathit{put} & = & p \\ \mathit{create}\ v & = & p\ v\ (\mathit{representative}(S)) \end{array}$$

where $representative(S)$ denotes an arbitrary element of S. By construction, we immediately have $l.put = p$. We prove that l is well behaved by showing each of the lens laws directly:

▶ **GetPut**: Let $s \in S$. We calculate as follows

$$\begin{array}{ll} & l.\mathit{put}\ (l.\mathit{get}\ s)\ s \\ = p\ \widehat{s}\ s & \text{by definition } l.\mathit{get} \text{ and } l.\mathit{put} \\ = s & \text{by definition } \widehat{s} \end{array}$$

and obtain the required equality.

▶ **PutGet**: Let $v \in V$ and $s \in S$. We calculate as follows

$$\begin{array}{ll} & l.\mathit{get}\ (\mathit{put}\ v\ s) \\ = \widehat{p\ v\ s} & \text{by definition } l.\mathit{get} \text{ and } l.\mathit{put} \\ = \widehat{s'} & \\ \quad \text{where } s' = p\ v\ s & \\ = v' & \text{by definition } \widehat{\cdot} \\ \quad \text{where } p\ v'\ s' = s' & \\ = v & \text{by semi-injectivity of } p \text{ and } p\ v\ s = s' \end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof of PutGet. □

Now we turn to the main technical result in this section: a semantic characterization of lenses entirely in terms of *put* functions. We start by showing that the *get* component of every lens is determined by its *put* component.

2.2.5 Lemma: For all basic lenses $l_1 \in S \Longleftrightarrow V$ and $l_2 \in S \Longleftrightarrow V$, if $l_1.\mathit{put}\ v\ s = l_2.\mathit{put}\ v\ s$ for every view v in V and source s in S, then $l_1.\mathit{get}\ s = l_2.\mathit{get}\ s$ for every source s in S.

Proof: Let $l_1 \in S \Longleftrightarrow V$ and $l_2 \in S \Longleftrightarrow V$ be lenses such that $l_1.\mathit{put}\ v\ s = l_2.\mathit{put}\ v\ s$ for every view v in V and source s in S. Let s be a source in S. We calculate as follows

$$\begin{array}{ll} & l_1.\mathit{get}\ s \\ = l_2.\mathit{get}\ (l_2.\mathit{put}\ (l_1.\mathit{get}\ s)\ s) & \text{by PutGet for } l_2 \\ = l_2.\mathit{get}\ (l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ s) & \text{as } l_1.\mathit{put} = l_2.\mathit{put} \\ = l_2.\mathit{get}\ s & \text{by GetPut for } l_1 \end{array}$$

and obtain the required equality. □

Next, we show that the *create* component of every lens is consistent with its *put* component.

2.2.6 Lemma: For every lens $l \in S \Longleftrightarrow V$ and view v in V there exists a source s in S such that $l.\mathit{create}\ v = l.\mathit{put}\ v\ s$.

Proof: Let $l \in S \Longleftrightarrow V$ be a lens and let v be a view in V. We calculate as follows

$$\begin{array}{ll} & l.\mathit{put}\ v\ (l.\mathit{create}\ v) \\ = l.\mathit{put}\ (l.\mathit{get}\ (l.\mathit{create}\ v))\ (l.\mathit{create}\ v) & \text{by CreateGet for } l \\ = l.\mathit{create}\ v & \text{by GetPut for } l \end{array}$$

and obtain the required equality, with $l.\mathit{create}\ v$ as the source. □

Lens programmers often feel like they are writing the forward transformation (because the names of primitives typically connote the forward transformation) and getting the backward transformation "for free," but these last few lemmas demonstrate that the opposite is actually true: they write the reverse transformation and get the forward transformation for free. Lemma 2.2.5 shows that the *get* component of a lens is determined by its *put* function. Lemma 2.2.6 shows that the *create* and *put* functions are partially redundant—their behaviors could be merged into a single function *put′* $\in V \to S\ option \to S$ where the optional S argument indicates whether to do a *put* or a *create*. Lemmas 2.2.1 and 2.2.4 precisely characterize the properties of *put* functions. Putting all these results together, we have a semantics of lenses defined only in terms of *put′* functions. However, although *put′* suffices for a purely semantic characterization of lenses, the construction of *get* in Lemma 2.2.4 is not effective—it requires finding a view v in V satisfying *put′* $v\ (Some\ s) = s$. So, in the rest of this dissertation, we will work with the original definition of basic lenses and give the *get*, *put*, and *create* functions of each lens explicitly.

2.3 Syntax

With the semantics of basic lens in place, we now turn our attention to syntax. From this point on, we will restrict our attention to objects drawn from the set Σ^* of strings over a fixed alphabet Σ. Why strings and not richer structures such as trees, relations, or complex values? There are many reasons. First, strings expose many fundamental issues without the distractions of more complicated data model (in particular, strings expose all of the issues having to do with ordered data—see Chapter 4). Second, there is a lot of string data in the world, so it is convenient to have a language for defining views over strings directly, without first having to parse them into other formats. Third, programmers are already familiar with standard string transformation languages based on regular expressions so a bidirectional language also based on regular expressions should have broad appeal.

Before we can define the primitives in our language formally, we need to fix notation for strings, languages, regular expressions, etc.

Notation

2.3.1 Definition [Alphabet]: Fix a finite set $\Sigma = \{c_1, \ldots, c_n\}$ of symbols (e.g., ASCII). We call the set Σ the *alphabet* and we call symbols $c \in \Sigma$ *characters*.

2.3.2 Definition [String]: A *string* is a finite sequence $(c_1 \cdots c_k)$ of characters in Σ. The set of all strings over Σ is written Σ^*.

2.3.3 Notation [Empty String]: We write ϵ for the empty string.

2.3.4 Notation [Length]: We write $|u|$ for the *length* of a string $u \in \Sigma^*$.

2.3.5 Notation [String Concatenation]: We write $u{\cdot}v$ for the concatenation of strings u and v.

2.3.6 Definition [Language]: A *language* is a subset of Σ^*.

2.3.7 Notation [Representative]: The $representative(\cdot)$ function takes a language L as an argument, which must not be empty, and yields a string belonging to L.

2.3.8 Definition [Language Concatenation]: The *concatenation* of a language L_1 and a language L_2, written $L_1{\cdot}L_2$, is the language containing every concatenation of a string in L_1 and a string in L_2:

$$L_1{\cdot}L_2 \triangleq \{u{\cdot}v \mid u \in L_1 \text{ and } v \in L_2\}$$

2.3.9 Definition [Language Iteration]: The n-fold *iteration* of a language $L \subseteq \Sigma^*$, written L^n, is the language containing every concatenation of n strings from L. It is defined formally by induction on n as follows:

$$\begin{array}{rcl} L^0 & \triangleq & \{\epsilon\} \\ L^{i+1} & \triangleq & L^i \cdot L \end{array}$$

Note that $L^1 = L^0 \cdot L = \{\epsilon\} \cdot L = L$.

2.3.10 Definition [Kleene Closure]: The *Kleene closure* of L, written L^*, is the union of every iteration of L:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Some of the primitives in our language require that every string belonging to the concatenation of two languages have a unique factorization into a pair of substrings belonging to the concatenated languages.

2.3.11 Definition [Unambiguous Concatenation]: Two languages L_1 and L_2 are *unambiguously concatenable*, written $L_1 \cdot^! L_2$, if for all strings $u_1 \in L_1$ and $u_2 \in L_2$ and all strings $v_1 \in L_1$ and $v_2 \in L_2$ if $(u_1 \cdot u_2) = (v_1 \cdot v_2)$ then $u_1 = v_1$ and $u_2 = v_2$.

Likewise, some of the primitives require that every string belonging to the Kleene closure of a language have a unique decomposition into a list of substrings belonging to the iterated language.

2.3.12 Definition [Unambiguous Iteration]: A language $L \subseteq \Sigma^*$ is *unambiguously iterable*, written $L^{!*}$, if for all strings $u_1 \in L$ to $u_m \in L$ and all strings $v_1 \in L$ to $v_n \in L$, if $(u_1 \cdots u_m) = (v_1 \cdots v_n)$ then $m = n$ and $u_i = v_i$ for every i from 1 to n.

The types of the basic string lenses defined in this chapter will be given by regular languages—a class of languages that enjoys many desirable properties including closure under the boolean operators and algorithms for deciding equivalence, inclusion, disjointness, and the ambiguity conditions just defined. We will describe regular languages using regular expressions.

2.3.13 Definition [Regular Expressions]: The set of *regular expressions* over Σ is the smallest set generated by the following grammar

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

where u ranges over strings (including ϵ) in Σ^*.

2.3.14 Definition [Regular Expression Semantics]: The language $[\![R]\!]$ denoted by a regular expression R is defined by structural induction as follows:

$$\begin{array}{rcl} [\![\emptyset]\!] & \triangleq & \{\} \\ [\![u]\!] & \triangleq & \{u\} \\ [\![R_1 \cdot R_2]\!] & \triangleq & [\![R_1]\!] \cdot [\![R_2]\!] \\ [\![R_1 \,|\, R_2]\!] & \triangleq & [\![R_1]\!] \cup [\![R_2]\!] \\ [\![R_1{}^*]\!] & \triangleq & [\![R_1]\!]^* \end{array}$$

2.3.15 Definition [Regular Language]: A language L is *regular* if and only if there exists a regular expression R in $\mathcal{R}$ such that $L = [\![R]\!]$.

2.3.16 Fact [Closure Properties]: Let $L_1 \subseteq \Sigma^*$ and $L_1 \subseteq \Sigma^*$ be regular languages. The following languages are also regular [HU79]:

- the intersection $(L_1 \cap L_2)$ of L_1 and L_2,
- the difference $(L_1 - L_2)$ of L_1 and L_2,
- the left $(L_1 \backslash L_2)$ and right (L_1/L_2) quotient of L_1 and L_2 where

$$\begin{array}{l} (L_1 \backslash L_2) \triangleq \{v \in \Sigma^* \mid \exists u \in L_1.\ u{\cdot}v \in L_2\} \\ (L_2/L_2) \triangleq \{u \in \Sigma^* \mid \exists v \in L_2.\ u{\cdot}v \in L_1\} \end{array}$$

2.3.17 Fact [Emptiness]: It is decidable whether a regular language $L \subseteq \Sigma^*$ is empty [HU79]. As regular languages are effectively closed under negation and intersection, it is also decidable whether one regular language is included in another and whether two regular languages are equivalent.

Now we show that it is decidable if two regular languages are unambiguously concatenable and if a regular language is unambiguously iterable.

2.3.18 Lemma: It is decidable whether the concatenation of two regular languages L_1 and L_2 is ambiguous.

Proof: Let L_1 and L_2 be regular languages. Define languages S_1 and P_2 as follows:

$$\begin{array}{rclcl} S_1 & \triangleq & (L_1 \backslash L_1) & = & \{v \mid \exists u \in L_1.\ u \cdot v \in L_1\} \\ P_2 & \triangleq & (L_2/L_2) & = & \{u \mid \exists v \in L_2.\ u \cdot v \in L_2\} \end{array}$$

Intuitively, S_1 is the set of "suffixes" that can be concatenated to the end of a word in L_1 to produce a word in L_1 and P_2 is the set of "prefixes" that can be concatenated to the beginning of a word in L_2 to produce a word in L_2. We will prove that $(S_1 \cap P_1) - \{\epsilon\} = \{\}$ if and only if L_1 and L_2 are unambiguously concatenable.

($\Rightarrow$) Suppose, for a contradiction, that $((S_1 \cap P_2) - \{\epsilon\}) = \{\}$ but L_1 and L_2 are not unambiguously concatenable. Then there exist strings $u_1 \in L_1$ and $u_2 \in L_2$ and $v_1 \in L_1$ and $v_2 \in L_2$ such that $(u_1{\cdot}u_2) = (v_1{\cdot}v_2)$ and $u_1 \neq v_1$ or $u_2 \neq v_2$.

Without loss of generality, suppose that $|u_1| > |v_1|$. As $(u_1{\cdot}u_2) = (v_1{\cdot}v_2)$ there exists a string $w \in \Sigma^*$ such that $w \neq \epsilon$ and $u_1 = (v_1{\cdot}w)$ and $v_2 = (w{\cdot}u_2)$. Hence, $w \in (S_1 \cap P_2) - \{\epsilon\}$, a contradiction. We conclude that L_1 and L_2 are unambiguously concatenable.

($\Leftarrow$) Suppose, for a contradiction, that L_1 and L_2 are unambiguously concatenable but $((S_1 \cap P_2) - \{\epsilon\}) \neq \{\}$. Then there exists a string $w \in \Sigma^*$ such that $w \neq \epsilon$ and $w \in S_1$ and $w \in P_2$. Moreover, by the definition of S_1 and P_2, there exist strings $u_1 \in L_1$ such that $(u_1{\cdot}w) \in L_1$ and $u_2 \in L_2$ such that $(w{\cdot}u_2) \in L_2$. Hence, as concatenation is associative, we have

$$(u_1{\cdot}w){\cdot}u_2 = u_1{\cdot}(w{\cdot}u_2)$$

but $u_1{\cdot}w \neq u_1$ and $u_2 \neq w{\cdot}u_2$. This contradicts the assumption that the concatenation of L_1 and L_2 is unambiguous so we conclude that $((S_1 \cap P_2) - \{\epsilon\}) = \{\}$.

The required result is immediate as regular languages are closed under intersection and difference and as emptiness is decidable for regular languages. □

2.3.19 Fact: It is decidable whether a regular language $L \subseteq \Sigma^*$ is unambiguously iterable.

Proof: Let $L \subseteq \Sigma^*$. We will prove that $\epsilon \notin L$ and $L \cdot^! L^*$ if and only if L is unambiguously iterable.

($\Rightarrow$) Suppose, for a contradiction, that $\epsilon \notin L$ and $L \cdot^! L^*$ but L is not unambiguously iterable. Then there exist strings u_1 to u_m in L and v_1 to v_n in L such that $(u_1 \cdots u_m) = (v_1 \cdots v_n)$ but either $m \neq n$ or $u_i \neq v_i$ for some i between 1 and n. We will prove by induction on m that $m = n$ and $u_i = v_i$ for i from 1 to n.

Case $m = 0$**:** From $\epsilon = (v_1 \cdots v_n)$ and $\epsilon \notin L$ we have that $n = 0$. We also have $u_i = v_i$ for every i from 1 to n vacuously.

Case $m > 0$**:** As $(u_1 \cdots u_m) = (v_1 \cdots v_n)$ and $\epsilon \notin L$ we have $n > 0$. As $L \cdot^! L^*$ we also have $u_1 = v_1$ and $(u_2 \cdots u_n) = (v_2 \cdots v_m)$. By the induction hypothesis, we have $(m - 1) = (n - 1)$ and $u_i = v_i$ for i from 2 to n.

In either case we have a contradiction to the existence of $u_1 \cdots u_m$ and $v_1 \cdots v_n$ with $m \neq n$ or $u_i \neq v_i$ for some i in 1 to n. We conclude that L is unambiguously iterable.

($\Leftarrow$) Suppose, for a contradiction, that L is unambiguously iterable but either $\epsilon \in L$ or not $L \cdot^! L^*$. On the one hand, if $\epsilon \in L$, then we immediately have $\epsilon \cdot \epsilon = \epsilon$ which contradicts the assumption that L is unambiguously iterable. On the other hand, if it is not the case that $L \cdot^! L^*$, then there exist strings $u_1 \in L$ and $u_2 \in L^*$ and $v_1 \in L$ and $v_2 \in L^*$ such that $(u_1 \cdot u_2) = (v_1 \cdot v_2)$ but $u_1 \neq v_1$ or $u_2 \neq v_2$. From these facts it follows that $u_1 \neq v_1$ and $u_2 \neq v_2$. However, as $(u_1 \cdot u_2) \in L^*$ and $(v_1 \cdot v_2) \in L^*$ and L unambiguously iterable, we also have that $u_1 = v_1$, a contradiction. Hence, we conclude that $\epsilon \notin L$ and $L \cdot^! L^*$. □

Atomic Lenses

With this notation in place, we are now ready to define some lens primitives. The types of the source and view for each primitive defined in this chapter (with one exception, the *dup* E lens) are regular languages. Working with regular languages ensures that we can build an efficient typechecker that verifies all of the conditions needed to guarantee well behavedness automatically.

Let us warm up with a few primitives that do simple rewritings on strings.

Copy The *copy* lens is parameterized on a regular expression $E \in \mathcal{R}$. It behaves like the identity function on $[\![E]\!]$ in both directions. The components of *copy* are defined precisely in the box below.

$$\frac{E \in \mathcal{R}}{\mathit{copy}\ E \in [\![E]\!] \Longleftrightarrow [\![E]\!]}$$

$$\begin{aligned} &\mathit{get}\ e &&= e \\ &\mathit{put}\ e'\ e &&= e' \\ &\mathit{create}\ e &&= e \end{aligned}$$

In the *get* direction, *copy* E copies a string belonging to (the language denoted by) E from the source to the view. In the *put* direction, it copies the view and ignores its source argument. Intuitively, this behavior makes sense: because the view is obtained by copying the source string verbatim, updates to the view should also be propagated verbatim to the source. It is also forced by the PutGet law.

The typing rule in the box above can be read as a lemma asserting that if E is a regular expression then *copy* E is a basic lens from $[\![E]\!]$ to $[\![E]\!]$. As this is our first lens, we prove this lemma explicitly. We include analogous well-behavedness proofs for each of the lens primitives defined in this dissertation. However, since they are largely calculational we defer the rest of these proofs to the appendix.

2.3.20 Lemma: Let E be a regular expression. Then *copy* E is a basic lens in $[\![E]\!] \Longleftrightarrow [\![E]\!]$.

Proof: We prove each basic lens law separately.

▶ **GetPut:** Let e be a string in $[\![E]\!]$. We calculate as follows

$$\begin{array}{ll} \ (\textit{copy}\ E).\textit{put}\ ((\textit{copy}\ E).\textit{get}\ e)\ e & \\ = (\textit{copy}\ E).\textit{put}\ e\ e & \text{by definition } (\textit{copy}\ E).\textit{get} \\ = e & \text{by definition } (\textit{copy}\ E).\textit{put} \end{array}$$

and obtain the required result.

▶ **PutGet:** Let e and e' be strings in $[\![E]\!]$. We calculate as follows

$$\begin{array}{ll} \ (\textit{copy}\ E).\textit{get}\ ((\textit{copy}\ E).\textit{put}\ e'\ e) & \\ = (\textit{copy}\ E).\textit{get}\ e' & \text{by definition } (\textit{copy}\ E).\textit{put} \\ = e' & \text{by definition } (\textit{copy}\ E).\textit{get} \end{array}$$

and obtain the required result.

▶ **CreateGet:** Let e be a string in $[\![E]\!]$. We calculate as follows

$$\begin{array}{ll} \ (\textit{copy}\ E).\textit{get}\ ((\textit{copy}\ E).\textit{create}\ e) & \\ = (\textit{copy}\ E).\textit{get}\ e & \text{by definition } (\textit{copy}\ E).\textit{create} \\ = e & \text{by definition } (\textit{copy}\ E).\textit{get} \end{array}$$

and obtain the required result. □

By inspection, *copy* E is an oblivious lens. It follows that it is also very well behaved.

Constant (and derived forms) The next lens behaves like the constant function in the *get* direction. The lens *const* E u takes as arguments a regular expression $E \in \mathcal{R}$ and a string $u \in \Sigma^*$. In the *get* direction maps every string in $[\![E]\!]$ to u and in the *put* direction it discards the view and restores the original source. The *create* function maps u to an arbitrary string $\mathit{representative}(E) \in [\![E]\!]$. The side condition $[\![E]\!] \neq \{\}$ in the typing rule below ensures that a representative exists. Note that *const* E u satisfies the PutGet law because its view type is a singleton set.

$$\frac{E \in \mathcal{R} \qquad [\![E]\!] \neq \emptyset \qquad u \in \Sigma^*}{\textit{const}\ E\ u \in [\![E]\!] \Longleftrightarrow \{u\}}$$

$$\begin{array}{ll} \textit{get}\ e & = u \\ \textit{put}\ u\ e & = e \\ \textit{create}\ u & = \mathit{representative}(E) \end{array}$$

2.3.21 Lemma: Let E be a regular expression and u a string such that $[\![E]\!] \neq \emptyset$. Then *const* E u is a basic lens in $[\![E]\!] \Longleftrightarrow \{u\}$.

We will often write $E \leftrightarrow u$ instead of *const* E u, especially in examples. Note that the constant lens is not oblivious, but it is very well behaved. This is easy to see because it propagates its entire source argument in the *put* direction.

Several useful lenses can be expressed as derived forms using *const*:

$$\boxed{\begin{array}{lcl} \mathit{del}\ E & \in & [\![E]\!] \Longleftrightarrow \{\epsilon\} \\ \mathit{del}\ E & \triangleq & E \leftrightarrow \epsilon \end{array}} \qquad \boxed{\begin{array}{lcl} \mathit{ins}\ u & \in & \{\epsilon\} \Longleftrightarrow \{u\} \\ \mathit{ins}\ u & \triangleq & \epsilon \leftrightarrow u \end{array}}$$

The *get* component of *del* E takes any source string belonging to E and deletes it, adding nothing to the view. Its *put* component restores the deleted string. Conversely, *ins* u inserts a fixed string u into the view in the *get* direction and removes it in the *put* direction. We used these lenses in the composers lens in Chapter 1 to delete parts of the source (e.g., `(del ALPHA)`, which deletes the nationality of a composer) and to add strings to the view (e.g., `(ins ", ")`, which inserts a separator between the names and dates of a composer).

Default The *create* component of the *const* lens we just described produces an arbitrary element of the source type. In many situations, however, the choice of a default is important. The *default* combinator gives programmers a way to control these choices. It takes a lens $l \in S \Longleftrightarrow V$ and a total function $f \in V \to S$ as arguments and overrides the *create* component of l with a call to *put*, using f to manufacture a source string from the view.

$$\boxed{\dfrac{\begin{array}{c} l \in S \Longleftrightarrow V \\ f \in V \to S \end{array}}{\mathit{default}\ l\ f \in S \Longleftrightarrow V}}$$

$$\boxed{\begin{array}{ll} \mathit{get}\ s & = l.\mathit{get}\ s \\ \mathit{put}\ v\ s & = l.\mathit{put}\ v\ s \\ \mathit{create}\ v & = l.\mathit{put}\ v\ (f\ v) \end{array}}$$

2.3.22 Lemma: Let $l \in S \Longleftrightarrow V$ be a basic lens and $f \in V \to S$ a total function. Then *default* l f is a basic lens in $S \Longleftrightarrow V$.

The *default* lens can be used to change the behavior of *const*. Suppose that we want to delete a date from the source, as in the following example, written in the Boomerang language described in Chapter 6.

```
let DATE : regexp = DIGIT{4} . ("-" . DIGIT{2}){2}
let l : lens = del DATE
test (l.get "2010-03-13") = ""
```

The first line defines a regular expression for valid date strings and the second line defines a lens that deletes a date string from the source. The third line is a unit test. The Boomerang implementation verifies that the expressions on the left and right sides evaluate to the same value. We use these unit tests throughout this dissertation to illustrate the behavior of specific lenses on simple examples. The *create* function for `l` produces an arbitrary representative of the language denoted by `DATE`:

```
test l.create "" = "0000-00-00"
```

Obviously this string is not always the best choice for a default date. The *default* operator lets us to override it with a better choice—e.g., the first day of the Unix epoch

```
let epoch (x:string) : string = "1970-01-01"
test (default l epoch).create "" = "1970-01-01"
```

or even the current date:

```
let today (x:string) : string =
  Sys.exec "date +%Y-%m-%d|tr -d '\n'"
test (default l today).create "" = "2010-03-13"
```

Recall that the *create* function must be consistent with *put* by Lemma 2.2.6. We can use the *default* combinator to equip a basic lens with every computable *create* function it can have and still be well behaved.

Regular Operators

Next we present some operators that build bigger lenses out of smaller ones. The combinators described in this section are based on the regular operators—union, concatenation, and Kleene star. Although their details are somewhat tailored to strings, they illustrate essential issues that also come up with richer structures—e.g., see the tree lenses defined in the original paper on lenses [FGM+07].

Concatenation Let us start with concatenation, which is the simplest regular operator.

$$\frac{\begin{array}{ll} l_1 \in S_1 \Longleftrightarrow V_1 & S_1 \cdot^{!} S_2 \\ l_2 \in S_2 \Longleftrightarrow V_2 & V_1 \cdot^{!} V_2 \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2) \Longleftrightarrow (V_1 \cdot V_2)}$$

$$\begin{aligned} \mathit{get}\ (s_1 \cdot s_2) &= (l_1.\mathit{get}\ s_1) \cdot (l_2.\mathit{get}\ s_2) \\ \mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) &= (l_1.\mathit{put}\ v_1\ s_1) \cdot (l_2.\mathit{put}\ v_2\ s_2) \\ \mathit{create}\ (v_1 \cdot v_2) &= (l_1.\mathit{create}\ v_1) \cdot (l_2.\mathit{create}\ v_2) \end{aligned}$$

2.3.23 Lemma: Let $l_1 \in S_1 \Longleftrightarrow V_1$ and $l_2 \in S_2 \Longleftrightarrow V_2$ be basic lenses such that $S_1 \cdot^{!} S_2$ and $V_1 \cdot^{!} V_2$. Then $l_1 \cdot l_2$ is a basic lens in $(S_1 \cdot S_2) \Longleftrightarrow (V_1 \cdot V_2)$.

In the *get* direction, the concatenation lens splits the source string into two smaller strings that belong to S_1 and S_2, applies the *get* components of l_1 and l_2 to these strings, and then concatenates the results to form the view. The *put* and *create* functions are similar. To lighten the notation, we have written $(s_1 \cdot s_2)$ in the box above to indicate that s_1 and s_2 are strings belonging to S_1 and S_2 rather than making the split function explicit. We use this convention throughout the rest of this dissertation. The languages used to split the string will be clear from context.

The typing rule requires that the concatenation of the source types and the concatenation of the view types each be unambiguously concatenable. This ensures two important properties:

1. The functional components of the lens, defined using the convention just described, are well-defined functions.

2. The lens obeys the round-tripping laws.

To see what would go wrong if we omitted these conditions, consider the ill-typed transformation l_{ambig}:

$$l_{ambig} \triangleq (\mathtt{a} \leftrightarrow \mathtt{a} \,|\, \mathtt{aa} \leftrightarrow \mathtt{aa}) \cdot (\mathtt{a} \leftrightarrow \mathtt{b} \,|\, \mathtt{aa} \leftrightarrow \mathtt{b})$$

We assume that "$\leftrightarrow$" binds tighter than " | ", which is the union combinator on lenses and is defined next. The first issue with the l_{ambig} lens is that its *get* component is not even a function! According to the specification of the concatenation operator given above, $l_{ambig}.get$ aaa = ab if we split aaa into a and aa and $l_{ambig}.get$ aaa = aab if we split it into aa and a. We might try to sidestep this issue by allowing the programmer to specify a policy for choosing among the multiple possible parses of ambiguous strings—e.g., using a shortest match heuristic we would split aaa into a and aa so $l_{ambig}.get$ aaa would only yield ab as a result. But the lens laws do not always hold when the source and view are split heuristically. Intuitively, just because we split the source string using one policy does not mean we can use the same policy to split the view. As an example, consider the (ill-behaved) lens l_{bogus} defined as $k{\cdot}k$ where:

$$k \triangleq (\mathtt{a} \leftrightarrow \mathtt{bb} \,|\, \mathtt{aa} \leftrightarrow \mathtt{a} \,|\, \mathtt{b} \leftrightarrow \mathtt{b} \,|\, \mathtt{ba} \leftrightarrow \mathtt{ba})$$

Using the shortest match policy to split the source we have:

$$\begin{aligned} l_{bogus}.get\ \mathtt{aaa} &= (k.get\ \mathtt{a}) \cdot (k.get\ \mathtt{aa}) \\ &= \mathtt{bb} \cdot \mathtt{a} \\ &= \mathtt{bba} \end{aligned}$$

Then, using the same shortest match policy to split the view, we have

$$\begin{aligned} l_{bogus}.put\ \mathtt{bba}\ \mathtt{aaa} &= (k.put\ \mathtt{b}\ \mathtt{a}) \cdot (k.put\ \mathtt{ba}\ \mathtt{aa}) \\ &= \mathtt{b} \cdot \mathtt{ba} \\ &= \mathtt{bba} \\ &\neq \mathtt{aaa} \end{aligned}$$

which violates the GETPUT law because the new source bba and the original source aaa are different. It is easy to construct similar counterexamples to the PUTGET law.

It turns out that the condition that the source types be unambiguously concatenable is essential for ensuring the lens laws. However, the condition on the view types is not—we can replace the *put* component of the concatenation lens with a more complicated version that uses its source argument to check if the old and views can be split "in the same way" and use this to split the view if so.[2] This *put* function obeys the GETPUT law because the old and new views are split the same way if possible, which will always be the case when they are identical. On the same strings as above, the *put* of l_{bogus} would produce the original source string, as required by GETPUT:

$$\begin{aligned} l_{bogus}.put\ \mathtt{bba}\ \mathtt{aaa} &= (k.put\ \mathtt{bb}\ \mathtt{a}) \cdot (k.put\ \mathtt{a}\ \mathtt{aa}) \\ &= \mathtt{a} \cdot \mathtt{aa} \\ &= \mathtt{aaa} \end{aligned}$$

Unfortunately, the same technique can not be used on the source side. The *get* function only takes a single argument, so we have no way to force it to split the source "in the same way" as the strings

[2] Here is how this revised variant of the concatenation lens would work, as originally suggested by Julien Cretin. Call the position of a split in the set of all possible splits (ordered lexicographically according to the lengths of the strings in the pairs resulting from each split) its *index*. For example, the index of the split obtained using the shortest match is 0 and the index of the longest match is the size of the set of all splits. Revise the *put* function so that the view is split by the index of $(l_1.get\ s_1, l_2.get\ s_2)$ if the index corresponds to a valid split and by any other index if not.

produced by l_1 and l_2's *put* functions unless the concatenation of S_1 and S_2 is already unambiguous. Consider the following example where we use the shortest match heuristic to split the source string:

$$\begin{aligned}
l_{bogus}.\mathit{get}\ (l_{bogus}.\mathit{put}\ \texttt{babb}\ \texttt{baa}) &= l_{bogus}.\mathit{get}\ ((k.\mathit{put}\ \texttt{ba}\ \texttt{b})\cdot(k.\mathit{put}\ \texttt{bb}\ \texttt{aa}))\\
&= l_{bogus}.\mathit{get}\ (\texttt{ba}\cdot\texttt{a})\\
&= l_{bogus}.\mathit{get}\ (\texttt{baa})\\
&= (k.\mathit{get}\ \texttt{b})\cdot(k.\mathit{get}\ \texttt{aa})\\
&= (\texttt{b}\cdot\texttt{a})\\
&= \texttt{ba}\\
&\neq \texttt{babb}
\end{aligned}$$

The *get* function has no way to determine that bba needs be split into (ba·a) to satisfy PutGet so it blindly uses shortest match, which yields a bad result. In general, it is impossible to give a sound typing rule for the concatenation lens without the ambiguity condition on the source types because of such examples where ambiguity can be exploited to cause part of the string generated by l_1.*put* to be passed to l_2.*get*, or vice versa.

In fact, although it is interesting to observe that we could relax the typing rule to allow the concatenation of the view types to be ambiguous, we use the stricter condition originally stated. One reason for this is that, as discussed in Section 2.1, the GetPut law is a rather weak constraint on behavior, intended more as a loose guide than as a complete specification of correctness. Splitting the view in a different way when we know we are not in danger of violating GetPut obeys the letter of the law but violates its spirit—i.e., the idea that the *put* function should preserve the integrity of the underlying source string to the extent possible. The strict version of the typing rule preserves very well behavedness—i.e., yields a very well behaved lens when applied to very well behaved arguments—while the relaxed version does not. Another reason is that, in our experience, programmers are not very good at reasoning about ambiguity. Most of the ambiguous concatenations we have encountered in real-world examples have turned out to be bugs. We believe that programmers would find the relaxed version of the concatenation lens unintuitive because minor edits could cause the view to be split in dramatically different ways.

Union The union combinator behaves like a conditional operator on lenses.

$$\frac{\begin{array}{c} S_1 \cap S_2 = \emptyset \\ l_1 \in S_1 \Longleftrightarrow V_1 \\ l_2 \in S_2 \Longleftrightarrow V_2 \end{array}}{l_1 \mid l_2 \in (S_1 \cup S_2) \Longleftrightarrow (V_1 \cup V_2)}$$

$$\begin{aligned}
\mathit{get}\ s &= \begin{cases} l_1.\mathit{get}\ s & \text{if } s \in S_1 \\ l_2.\mathit{get}\ s & \text{if } s \in S_2 \end{cases}\\
\mathit{put}\ v\ s &= \begin{cases} l_1.\mathit{put}\ v\ s & \text{if } v \in V_1 \text{ and } s \in S_1 \\ l_2.\mathit{put}\ v\ s & \text{if } v \in V_2 \text{ and } s \in S_2 \\ l_1.\mathit{create}\ v & \text{if } v \in (V_1 - V_2) \text{ and } s \in S_2 \\ l_2.\mathit{create}\ v & \text{if } v \in (V_2 - V_1) \text{ and } s \in S_1 \end{cases}\\
\mathit{create}\ v &= \begin{cases} l_1.\mathit{create}\ v & \text{if } v \in V_1 \\ l_2.\mathit{create}\ v & \text{if } v \in (V_2 - V_1) \end{cases}
\end{aligned}$$

2.3.24 Lemma: Let $l_1 \in S_1 \Longleftrightarrow V_1$ and $l_2 \in S_2 \Longleftrightarrow V_2$ be basic lenses such that the intersection $S_1 \cap S_2$ of the source types is empty. Then $l_1 \,|\, l_2$ is a basic lens in $(S_1 \cup S_2) \Longleftrightarrow (V_1 \cup V_2)$.

Like a conditional in an ordinary, unidirectional language, the union lens selects one of its branches by testing its inputs. The *get* function selects l_1 or l_2 by testing whether the source string belongs to S_1 or S_2. The typing rule for union requires that these two types be disjoint, so the choice is deterministic. The *put* function is more complicated, because the typing rule allows the view types to overlap. It first tries to select one of l_1 or l_2 using the view and only uses the source to disambiguate if the view belongs to both V_1 and V_2. The *create* function is similar, except that it uses l_1 in cases where the view belongs to V_1 and V_2 (it has no source argument to use). This is an arbitrary choice but it is not a limitation: to use l_2 instead, the programmer can use $(l_2 \,|\, l_1)$. It does mean, however, that the union operator is not commutative.

Because *put* is a total function, it needs to handle situations where the view comes from one side of the union, say $V_1 - V_2$, and the source comes from the other side, say S_2. The only way that the union lens can be sure to produce a source that will map back to the same view is to use one of the components of l_1. This basic version of union simply discards the source and uses l_1's *create* function. However, that this is not the only thing it could do—although the source belongs to S_2 it might still contain information that could be represented as a string belonging to S_1. We might like the *put* function to reintegrate this information with the new source. For example, consider the lens `l` defined as follows

```
let l1 : lens = copy [A-Z] . del [0-9]
let l2 : lens = del [0-9]
let l : lens = (l1 | l2)
```

and suppose that we put `A` into `3` using `(l1 | l2)`. We might like the *put* function to propagate the `A` in the view and restore the `3` from the source, but it invokes `l1`'s *create* function, which fills in the number with a default:

```
test l.put "A" into "3" = "A0"
```

In the original paper on basic lenses [FGM+07], we described a union combinator with "fixup" functions—mappings from S_2 to S_1 and from S_1 to S_2. The idea was that the *put* function would use these functions to extract information from sources on one side of the union for use with views on the other side of the union rather than discarding the source and invoking the *create* function. Semantically fixup functions are exactly what is needed—one can show that the union combinator described in that paper is most general. But syntactically they are quite cumbersome—the programmer has to write down two additional total functions on the source types! We refrain from introducing fixup functions in this chapter because resourceful lenses, discussed in Chapter 4, offer a different mechanism for passing information from one side of a union to the other that seems to balance the semantic and syntactic tradeoffs between these two extremes nicely.

By analogy with concatenation, one might wonder why we allow the view types to overlap in the union lens. One reason for this choice is that, in our experience, programmers are much more comfortable reasoning about disjointness than they are reasoning about ambiguity. Another reason is that many examples depend on overlapping union. In general, the union lens only preserves very well behavedness when the view types of the sublenses are identical, making it impossible to modify the view from one side of the union to the other, or when the sublenses are both oblivious so there is no hidden source data to preserve. For example, consider what happens when the edit changes the view from a string on one side of the union to a string on the other side:

```
let l1 : lens = copy [a-m] . del [0-4]
let l2 : lens = copy [n-z] . del [5-9]
test (l1|l2).put "n" into "a1" = "n5"
```

Because the union lens discards the source string and invokes the *create* component of `l1`, the PutPut law does not hold:

```
test (l1|l2).put "a" into ((l1|l2).put "n" into "a1") = "a0"
test (l1|l2).put "a" into "a1" = "a1"
```

It is tempting to add a condition to the typing rule for union to ensure that it preserves very well behavedness. However, many of our examples depend on both the overlapping and disjoint forms of union, so we use a typing rule that places no constraints on the view types.

Kleene Star The final combinator described in this section iterates a lens. It combines the behavior of concatenation and union in the usual way.

$$\frac{\begin{array}{c} S^{!*} \qquad V^{!*} \\ l \in S \Longleftrightarrow V \end{array}}{l^* \in S^* \Longleftrightarrow V^*}$$

$$\begin{array}{ll}
\mathit{get}\ (s_1 \cdots s_n) & = (l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n) \\
\mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) & = s'_1 \cdots s'_n \\
\qquad \text{where } s'_i = \begin{cases} l.\mathit{put}\ v_i\ s_i\ i \in \{1, \ldots, \min(n, m)\} \\ l.\mathit{create}\ v_i\ i \in \{m+1, \ldots, n\} \end{cases} & \\
\mathit{create}\ (v_1 \cdots v_n) & = (l.\mathit{create}\ v_1) \cdots (l.\mathit{create}\ v_n)
\end{array}$$

2.3.25 Lemma: Let $l \in S \Longleftrightarrow V$ be a basic lens such that $S^{!*}$ and $V^{!*}$. Then l^* is a basic lens in $S^* \Longleftrightarrow V^*$.

In the *get* direction, the Kleene star lens takes the source string, splits it (unambiguously) into a list of substrings belonging to the source type of the iterated lens l, applies the *get* component to each string in this list, and concatenates the results. The *put* and *create* functions are similar. Note that the *put* function needs to handle situations where the source and view have different numbers of substrings. To satisfy PutGet, the *put* function must produce a source string with the same number of substrings as the view. When there are more source substrings than view substrings it discards the extras and when there are more view substrings it processes the extras using $l.\mathit{create}$.

Because it sometimes discards parts of the source, Kleene star does not preserve very well behavedness, as shown in the following example:

```
let l = copy [a-z] . del [0-9]
test l*.put "xyz" into (l*.put "xy" into "a1b2c3") = "x1y2z0"
test l*.put "xyz" into "a1b2c3" = "x1y2z3"
```

This is unfortunate but unavoidable: we either have to allow the *put* component of Kleene star to discard parts of the source, sacrificing very well behavedness, or we have to restrict *put* to only accept views that have at least as many strings as the source, sacrificing totality. Since we take totality to be the more fundamental property, we choose the first option.

Extensions

The combinators described in the previous section correspond to unambiguous (due to the side conditions in the typing rules) finite state string transducers [Ber79]—a class of transformations that is powerful enough to express a large collection of practical examples. Some applications, however, require just a little more power. It is not difficult to extend our set of combinators with additional primitives—the only thing we require is that their types be expressible as regular languages, to ensure that typechecking remains decidable. In this section, we present a few of the other primitives that we have found useful in applications built using basic lenses.

Composition It is often convenient to express a transformation as the sequential composition of two simpler ones, even when it can be expressed as a more complicated single-pass transformation. The composition operator puts two lenses in sequence.

$$\frac{\begin{array}{c} l_1 \in S \Longleftrightarrow U \\ l_2 \in U \Longleftrightarrow V \end{array}}{l_1;l_2 \in S \Longleftrightarrow V}$$

$$\begin{array}{ll} \mathit{get}\ s & = l_2.\mathit{get}\ (l_1.\mathit{get}\ s) \\ \mathit{put}\ v\ s & = l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s)) \\ \mathit{create}\ v & = l_1.\mathit{create}\ (l_2.\mathit{create}\ v) \end{array}$$

2.3.26 Lemma: Let $l_1 \in S \Longleftrightarrow U$ and $l_2 \in U \Longleftrightarrow V$ be basic lenses. Then $l_1;l_2$ is a basic lens in $S \Longleftrightarrow V$.

The *get* component of the composition lens processes the source string by first applying the *get* functions of l_1 and l_2 in that order. The *put* component applies the *put* functions of l_1 and l_2 in the opposite order and uses l_1's *get* function to manufacture a string to use as the source argument for l_2's *put*. The *create* function is similar. The typing rule for composition requires that the view type of l_1 and the source type of l_2 be identical. This ensures that the intermediate strings belonging to U have the appropriate type.

As an example, recall the composers lens from Chapter 1. It requires that the name of each composer be a string described by the regular expression `(ALPHA . " " . ALPHA)`. It is not difficult to extend the lens so that the name can be an arbitrary string, but we need to be careful to respect the escaping conventions of the XML and ASCII formats. That is, we need to escape ‘`&`’, ‘`<`’, ‘`>`’, etc. characters on the XML side and ‘`,`’, ′`\n`′, and ‘`\`’ characters on the ASCII side. If we already have lenses `xml_unesc` and `csv_esc` that handle unescaping for XML and escaping for ASCII, then it is much simpler to compose these lenses than would be to write an end-to-end escaping lens from scratch. Here is a revised version of the composers lens that allows arbitrary strings as names:

```
let composer : lens =
  xml_elt "composer"
    ( xml_elt "name" ( xml_unesc ; csv_esc )
    . ins ", "
    . xml_elt "dates"  (copy (YEAR . "-" . YEAR) )
    . xml_elt "nationality"
        (del_default ALPHA "Unknown") )
```

This lens maps the XML source

```
let rec str_filter $S$ $xs$ = match $xs$ with
  | $\epsilon \rightarrow \epsilon$
  | $x{\cdot}xs'$ when $x \in S \rightarrow x{\cdot}$(str_filter $S$ $xs'$)
  | $x{\cdot}xs'$ when $x \notin S \rightarrow$ str_filter $S$ $xs'$

let rec str_unfilter $T$ $ys$ $xs$ = match $ys, xs$ with
  | $\_, \epsilon \rightarrow ys$
  | $\_, x{\cdot}xs'$ when $x \in T \rightarrow x{\cdot}$(str_unfilter $T$ $ys$ $xs'$)
  | $\epsilon, x{\cdot}xs'$ when $x \notin T \rightarrow$ str_unfilter $T$ $\epsilon$ $xs'$
  | $y{\cdot}ys', x{\cdot}xs'$ when $x \notin T \rightarrow y{\cdot}$(str_unfilter $T$ $ys'$ $xs'$)
```

Figure 2.2: Pseudocode for str_filter and str_unfilter.

```
<composer>
  <name>Duke Ellington &amp; His
Orchestra</name>
  <dates>1899-1974</dates>
  <nationality>American</nationality>
</composer>
```

to the ASCII view:

```
Duke Ellington & His\nOrchestra, 1899-1974
```

Note that the special characters in the name of Ellington's group are transformed according to the escaping conventions for each format.

Filter The next lens removes elements of the list of substrings of the source in the forward direction and restores them in the reverse direction. The *filter* operator takes regular expressions E and F as arguments and produces a lens that filters away the Fs from a string consisting of Es and Fs. Its *get* and *put* functions are given by helper functions str_filter and str_unfilter, which are defined in Figure 2.2.

$$\frac{[\![E]\!] \cap [\![F]\!] = \emptyset \qquad ([\![E]\!] \cup [\![F]\!])^{!*}}{\textit{filter}\ E\ F \in ([\![E]\!] \cup [\![F]\!])^* \Longleftrightarrow [\![E]\!]^*}$$

$$\begin{aligned} \textit{get}\ s &= \text{str\_filter}\ E\ s \\ \textit{put}\ v\ s &= \text{str\_unfilter}\ F\ v\ s \\ \textit{create}\ v &= v \end{aligned}$$

2.3.27 Lemma: Let E and F be regular expressions such that the intersection $[\![E]\!] \cap [\![F]\!]$ of $[\![E]\!]$ and $[\![F]\!]$ is empty and $([\![E]\!] \cup [\![F]\!])^{!*}$. Then *filter* E F is a basic lens in $([\![E]\!] \cup [\![F]\!])^* \Longleftrightarrow [\![E]\!]^*$.

It is tempting to define *filter* E F as $(\textit{copy}\ E \,|\, \textit{del}\ F)^*$ but the typing rules for Kleene star do not allow it—the view type of the iterated lens is not unambiguously iterable because it contains the empty string. Additionally, its *put* function would not behave the same—it would sometimes discard Fs in the source while str_unfilter always restores all of the Fs.

As an example illustrating the use of *filter*, suppose that we wanted to compute a view containing all the composers born in the 20th century. To achieve this, we revise the lens as follows. First, we parameterize the composer lens so that it takes a regular expression representing the birth date:

```
let composer (BIRTH:regexp) : lens =
  xml_elt "composer"
    ( xml_elt "name" (copy (ALPHA . " " . ALPHA) )
    . ins ", "
    . xml_elt "dates"  (copy (BIRTH . "-" . YEAR) )
    . xml_elt "nationality"
        ( del_default ALPHA "Unknown") )
```

Next, we instantiate composer twice, once for composers born in the 20th century, and another time for composers born in other centuries:

```
let YEAR_20c : regexp = "19" . DIGIT{2}
let composer_20c : lens = composer YEAR_20c
let composer_other : lens = composer (YEAR - YEAR_20c)
```

The main composers lens filters away non-20th century composers (the function stype extracts a regular expression that represents the source type of a lens) and processes the remaining composers using composer_20c.

```
let composers : lens =
  xml_elt "composers"
    ( filter (stype composer_20c) (stype composer_other);
      ( copy "" | composer_20c . (ins "\n" . composer_20c)* ) )
```

On the original XML source, this lens produces a view

```
Aaron Copland, 1910-1990
Benjamin Briten, 1913-1976
```

that does not contain an entry Sibelius, who was born in 1865. In the reverse direction, the *filter* lens always restores the filtered composers from the source. For example, here is the source obtained by putting the empty string into the original source:

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <dates>1865-1956</dates>
    <nationality>Finnish</nationality>
  </composer>
</composers>
```

Because its *put* function restores all of the filtered elements, *filter* E F is a very well behaved lens.

Swap The *get* components of every lens we have defined so far are expressible as one-way finite state string transducers [Ber79]. This class has a fundamental limitation: the restriction to finite state means that transformations cannot "remember" arbitrary amounts of data. In particular, we cannot use finite-state transducers to implement a variant of the composers lens whose *get* component inverts the order of the name and dates in the view because the name can be arbitrarily long. Fortunately, lifting this restriction poses no semantic problems. The *swap* combinator is like concatenation, but inverts the order of strings in the view.

$$\frac{\begin{array}{cc} l_1 \in S_1 \Longleftrightarrow V_1 & S_1 \cdot^! S_2 \\ l_2 \in S_2 \Longleftrightarrow V_2 & V_2 \cdot^! V_1 \end{array}}{l_1 \sim l_2 \in (S_1 \cdot S_2) \Longleftrightarrow (V_1 \cdot V_2)}$$

$$\begin{array}{ll} \mathit{get}\ (s_1 \cdot s_2) & = (l_2.\mathit{get}\ s_2) \cdot (l_1.\mathit{get}\ s_1) \\ \mathit{put}\ (v_2 \cdot v_1) & = (l_1.\mathit{put}\ v_1\ s_1) \cdot (l_2.\mathit{put}\ v_2\ s_2) \\ \mathit{create}\ (v_2 \cdot v_1) & = (l_1.\mathit{create}\ v_1) \cdot (l_2.\mathit{create}\ v_2) \end{array}$$

2.3.28 Lemma: Let $l_1 \in S_1 \Longleftrightarrow V_1$ and $l_2 \in S_2 \Longleftrightarrow V_2$ be basic lenses such that $(S_1 \cdot^! S_2)$ and $(V_2 \cdot^! V_1)$. Then $(l_1 \sim l_2)$ is a basic lens in $(S_1 \cdot S_2) \Longleftrightarrow (V_2 \cdot V_1)$.

As in the concatenation lens, the *get* component of swap splits the source in two and applies the *get* component of l_1 and l_2 to each. However, before it concatenates the results, it swaps them, which puts the results in the opposite order. The *put* and *create* functions are similar: they split the view into two strings, swap them, apply the *put* or *create* component of l_1 and l_2 to each, and concatenate the results.

To illustrate the use of *swap*, let us implement the variant of the composers lens described above where the name and dates for each composer are swapped in the view:

```
let composer : lens =
  xml_elt "composer"
    ( ( xml_elt "name" (copy (ALPHA . " " . ALPHA) )
      ~ ( xml_elt "dates"  (copy (YEAR . "-" . YEAR) )
        . ins ", " ) )
    . xml_elt "nationality" ( del_default ALPHA "Unknown" ) )
```

Compared to the previous version, this lens has two changes: First, we have moved the lens `ins ", "` down so that it follows the lens for the `dates` element. Second, we have replaced the concatenation (`.`) operator used to combine the lenses for the `name` and `dates` elements with swap (`~`). On the original XML source, this lens computes the following view:

```
1865-1956, Jean Sibelius
1910-1990, Aaron Copland
1913-1976, Benjamin Briten
```

It turns out that the concatenation and swap lenses are both instances of a more general combinator, *permute*, that is parameterized on a permutation σ on $\{1, \ldots, n\}$ and a list $[l_1, \ldots, l_n]$ of n lenses. In the *get* direction, it splits the source string into n substrings, processes each substring using the corresponding lens from the list, permutes the resulting list of strings using σ, and concatenates the results. The *put* and *create* functions are similar. For concatenation, we take σ to be the identity permutation on $\{1, 2\}$ and let the list of lenses be $[l_1, l_2]$. For swap, we let σ be the transposition on $\{1, 2\}$ and again let the list of lenses be $[l_1, l_2]$.

Merge The next operator merges two distinct pieces of the source. The *merge* lens is parameterized on a regular expression E, which must be unambiguously concatenable with itself. Its *get* function takes a source consisting of two Es and discards the second one. The *put* function has two cases. If the two Es in the source are equal, then it propagates the view to both copies. If they are not equal, then it propagates the view to the first copy and restores the second copy.

$$\frac{[\![E]\!]\cdot^{!}[\![E]\!]}{\mathit{merge}\ E \in ([\![E]\!]\cdot[\![E]\!]) \Longleftrightarrow [\![E]\!]}$$

$$\begin{aligned}
&\mathit{get}\ (e_1{\cdot}e_2) &&= e_1\\
&\mathit{put}\ e'\ (e_1{\cdot}e_2) &&= \begin{cases}(e'{\cdot}e') & \text{if } e_1 = e_2\\ (e'{\cdot}e_2) & \text{otherwise}\end{cases}\\
&\mathit{create}\ e' &&= e'{\cdot}e'
\end{aligned}$$

2.3.29 Lemma: Let E be a regular expression such that $([\![E]\!]\cdot^{!}[\![E]\!])$. Then *merge* E is a basic lens in $([\![E]\!]\cdot[\![E]\!]) \Longleftrightarrow [\![E]\!]$.

We often use the *merge* lens to handle cases where the source contains two piece of data, the view contains just one piece of data, and we want to update both pieces of source data consistently. For example, suppose that the source string contains a start date and an end date

```
START:2010-03-13 END:2010-03-13
```

and the view contains only the end date:

```
2010-03-13
```

Here is a lens whose *get* component does this transformation:

```
let l : lens =
  ( del "START:"
  . copy DATE
  . del " END:"
  . copy DATE );
  merge DATE
```

It deletes the `START` and `END:` tags as well as the space, and merges the two date strings that remain. The *put* function propagates updates to the view to both dates in the source if they are equal

```
test l.put "2010-02-14"
      into "START:2010-03-13 END:2010-03-13"
         = "START:2010-02-14 END:2010-02-14"
```

and only to the first date if they are different:

```
test l.put "2010-02-14"
      into "START:2010-03-01 END:2010-03-13"
         = "START:2010-02-14 END:2010-03-13"
```

Because *merge* sometimes uses the entire source and sometimes only uses part of it, it is not very well behaved:

```
let l : lens = merge [A-Z]
test l.put "A" into (l.put "B" into "AB") = "AA"
test l.put "A" into "AB" = "AB"
```

Duplicate A natural question to ask at this point is whether we can have a lens whose *get* component duplicates the source. By Lemma 2.2.3, we know that this question has a positive answer. The *dup* lens takes a regular expression E as an argument which must be unambiguously concatenable with itself. It makes two copies of the source string in the *get* direction and deletes the second copy in the *put* direction.

$$\frac{[\![E]\!]\cdot^{!}[\![E]\!]}{\mathit{dup}\ E \in [\![E]\!] \Longleftrightarrow \{e_1{\cdot}e_2 \in ([\![E]\!]{\cdot}[\![E]\!]) \mid e_1 = e_2\}}$$

$$\begin{aligned}
&\mathit{get}\ e &&= (e{\cdot}e)\\
&\mathit{put}\ (e_1{\cdot}e_2)\ e &&= e_1\\
&\mathit{create}\ (e_1{\cdot}e_2)\ e &&= e_1
\end{aligned}$$

2.3.30 Lemma: Let E be a regular expression such that $[\![E]\!]\cdot^{!}[\![E]\!]$. Then *dup* E is a basic lens in $[\![E]\!] \Longleftrightarrow \{e_1{\cdot}e_2 \in ([\![E]\!]{\cdot}[\![E]\!]) \mid e_1 = e_2\}$.

Semantically, *dup* is a perfectly valid basic lens. However, its type is not a regular language because it involves an equality constraint. This constraint is essential for guaranteeing the PutGet law—if we allow the *put* function to accept views where the two copies are different, then running *put* followed by *get* sometimes yields a different view than the one we started with. However, since the type of *dup* cannot be expressed using regular expressions (except when E is finite), Boomerang does not support it. We define it here for completeness and to illustrate the challenges that duplication raises in the context of basic lenses. In the next chapter, we will define a variant of *dup* that has a regular type and is included in Boomerang.

2.4 Summary

Basic lenses are a natural class of bidirectional transformations that provide a semantic foundation for bidirectional programming languages. Their design emphasizes both robustness and ease of use, guaranteeing totality as well as strong well-behavedness conditions formulated as round-tripping laws. Many transformations can be interpreted as basic lenses including the identity and constant functions, composition, iteration, conditional, product, and more.

Chapter 3

Quotient Lenses

"Good men must not obey the laws too well."
—Ralph Waldo Emerson

The story described in the previous chapter is an appealing one... but unfortunately it is not perfectly true! Most bidirectional transformations do *not* obey the basic lenses laws. Or rather, they obey them in spirit—modulo "unimportant" details—but not to the letter. The nature of these details varies from one application to another: examples include whitespace, artifacts of representing richer structures (relations, trees, and graphs) as strings, escaping of atomic data (XML PCDATA and CSV), ordering of fields in record-structured data (BibTeX fields and XML attributes), breaking of long lines of text, and computed values (tables of contents and aggregates).

To illustrate, consider the composers lens again. The information about each composer could be larger than comfortably fits on a single line of ASCII text. We might then want to relax the type of the view to allow lines to be broken (optionally) using a newline followed by at least one space, so that

```
Jean
  Sibelius, 1865-1957
```

and

```
Jean Sibelius, 1865-1957
```

would be accepted as equivalent, alternate presentations of the same view. But now we have a problem: by Lemma 2, we know that the PutGet law is only satisfied when *put* is semi-injective. This means that the `composer` lens must map these views, which we intuitively regard as equivalent, to different sources—i.e., the presence or absence of linebreaks in the view must be reflected in the source. We *could* design a lens that does this—e.g., storing the line break inside of the PCDATA

```
<composer>
  <name>Jean
  Sibelius</name>
  <dates>1865-1957</dates>
  <nationality>Finnish</nationality>
</composer>
```

but this "solution" isn't very attractive. For one thing, it places an unnatural demand on the XML representation—indeed, possibly an unsatisfiable demand if the application using the source forbids PCDATA containing newlines. For another, writing the lens that handles and propagates linebreaks involves extra work. Moreover, this warping of the XML format and complicated lens programming is all for the purpose of maintaining information that we don't actually care about! A much better alternative is to relax the lens laws to accommodate this transformation. Finding a way to do this elegantly is the goal of this chapter.

Several different ways of treating inessential data have been explored in previous work.

1. Some systems adopt an informal approach, stating the lens laws in their strict form and explaining that they "essentially hold". To support this claim, these systems often provide a description of how unimportant details are processed algorithmically. For example, the biXid language, which describes conversions between XML formats using pairs of intertwined tree grammars, provides no explicit guarantees about round-trip behavior but its designers clearly intend it to be "morally bijective" [KH06]. The PADS system is similar [FG05]. Overall, this approach is often quite reasonable, although it requires giving up formal properties.

2. Other systems weaken the lens laws. The designers of the X language have argued that the PutGet law should be replaced with a weaker "round-trip and a half" version [HMT08]:

$$\frac{s' = \mathit{put}\ v\ s}{\mathit{put}\ (\mathit{get}\ s')\ s' = s'} \qquad \textsc{(PutGetPut)}$$

   Their reason for advocating this law is to support a *duplication* operator. Having duplication makes it possible to express many useful transformations as lenses—e.g., augmenting a document with a table of contents—but because the duplicated data is not always preserved on round trips (consider making a change to just one copy of the duplicated data), it breaks the PutGet law. The weaker PutGetPut law imposes some constraints on the behavior of lenses, but it opens the door to a wide range of unintended behaviors—e.g., lenses with constant *put* functions, lenses whose *get* component is the identity function and whose *put* component is $\mathit{put}\ v\ s = s$, etc.[1]

3. Other systems split bidirectional transformations into a "core component" that is a lens in the strict sense and "canonization" phases that operate at the perimeters of the transformation and standardize the representation of inessential data. See Figure 3.1. For example, in our previous work on lenses for trees, the end-to-end transformations on concrete representations of trees in the filesystem only obey the lens laws up to an equivalence relation induced by a *viewer*—a parser and pretty printer that map between raw strings and more structured representations [FGM$^+$07]. Similarly, XSugar, a language for converting between XML and ASCII, guarantees that its transformations are bijective modulo a fixed relation on input and output structures obtained by normalizing "unordered" productions, "ignorable" non-terminals, and the representation of XML [BMS08].[2] This approach is quite workable when the data formats and canonizers are generic. However, for ad hoc data such as textual databases, bibliographies, configuration files, etc., it rapidly becomes impractical—the two components of the canonization transformation themselves become difficult to write and maintain. In particular, the schema of the data is recapitulated, redundantly, in the lens and in each canonizer! In other words, we end up back in the situation that lenses were designed to avoid.

[1] In the journal version of the paper [HMT08], they exclude such transformations by decorating the view with "edit tags" and adding a new law stipulating that doing a *put* followed by a *get* must yield a "more edited" view.

[2] In XSugar, XML canonization is treated as a distinct "pre-processing" phase, but canonization of other ignorable data is interleaved with the rest of the transformation; in this respect, XSugar can be regarded as a special case of the framework proposed in this chapter.

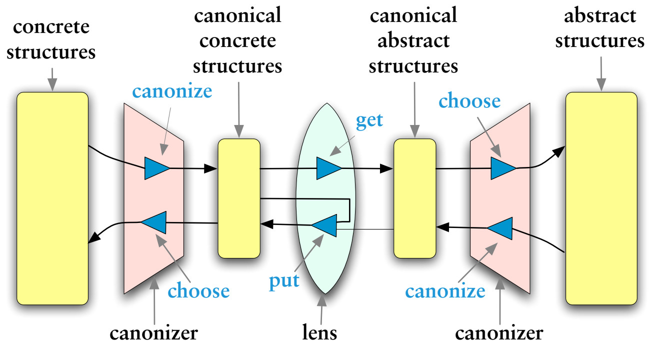


Figure 3.1: Lens architecture with "canonizers at the edges"

This chapter develops a more refined account of the lens framework that allows us to state, precisely and truthfully, that a bidirectional transformation is a well-behaved lens *modulo a particular equivalence relation*. The main advantages of this framework over the approach using viewers, as we will see, are that it allows us to compose lenses that are only well behaved modulo equivalences and to define and use canonizers anywhere in a lens program and not only at the perimeters.

The rest of this chapter is organized as follows. Section 3.1 presents the relaxed semantic space of quotient lenses. Section 3.2 describes a number of generic combinators—coercions from basic lenses to quotient lenses and from quotient lenses to canonizers, operators for quotienting a lens by a canonizer, and sequential composition. Section 3.2 defines quotient lens versions of the regular operators—concatenation, union, and Kleene star. Section 3.2 introduces some new primitives that are possible in the relaxed space of quotient lenses. Section 3.4 discusses typechecking for quotient lenses. Section 3.5 illustrates some uses of quotient lenses on an example—a lens for converting between XML and ASCII versions of a large genomic database. We conclude in Section 3.6.

3.1 Semantics

At the semantic level, the definition of quotient lenses is a straightforward refinement of basic lenses. Recall the definition of an equivalence relation:

3.1.1 Definition [Equivalence Relation]: A binary relation $R \subseteq S \times S$ is an *equivalence relation* if and only if it is reflexive, symmetric, and transitive.

In a quotient lens, we enrich the types of the source and view with equivalence relations—instead of $S \iff V$, we write $S/{\sim_S} \iff V/{\sim_V}$, where $\sim_S$ is an equivalence relation on S and $\sim_V$ is an equivalence relation on V—and we relax the lens laws by requiring that they only hold up to $\sim_S$ and $\sim_V$.

3.1.2 Definition [Quotient Lens]: Let $S \subseteq \mathcal{U}$ and $V \subseteq \mathcal{U}$ be sets of sources and views and let $\sim_S$ and $\sim_V$ be equivalence relations on S and V. A *quotient lens* l comprises three total functions

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

that obey the basic lens modulo $\sim_S$ and $\sim_V$:

$$l.\mathit{get}\ (l.\mathit{put}\ v\ s) \sim_V v \tag{\textsc{PutGet}}$$

$$l.\mathit{get}\ (l.\mathit{create}\ v) \sim_V v \tag{\textsc{CreateGet}}$$

$$l.\mathit{put}\ (l.\mathit{get}\ s)\ s \sim_S s \tag{\textsc{GetPut}}$$

Additionally, the components of every quotient lens must respect $\sim_S$ and $\sim_V$:

$$\frac{s \sim_S s'}{l.\mathit{get}\ s \sim_V l.\mathit{get}\ s'} \tag{\textsc{GetEquiv}}$$

$$\frac{v \sim_V v' \qquad s \sim_S s'}{l.\mathit{put}\ v\ s \sim_S l.\mathit{put}\ v'\ s'} \tag{\textsc{PutEquiv}}$$

$$\frac{v \sim_V v'}{l.\mathit{create}\ v \sim_S l.\mathit{create}\ v'} \tag{\textsc{CreateEquiv}}$$

We write $S/{\sim_S} \Longleftrightarrow V/{\sim_V}$ for the set of all quotient lenses between S (modulo $\sim_S$) and V (modulo $\sim_V$).

The relaxed round-tripping laws are just the basic lens laws on the equivalence classes $S/{\sim_S}$ and $V/{\sim_V}$. In fact, if we pick $\sim_S$ and $\sim_V$ to be equality—the finest equivalence relation—they are equivalent to the basic laws precisely. Note, however, that although we reason about the behavior of quotient lenses as if they operated on equivalence classes, the component functions actually operate on members of the underlying sets of sources and views—i.e, the type of *get* is $S \rightarrow V$ not $S/{\sim_S} \rightarrow V/{\sim_V}$. The second group of laws ensure that the components of a quotient lens treat equivalent structures equivalently. They play a critical role in (among other things) the proof that the typing rule for the composition operator is sound.

3.2 Syntax

So much for semantics. The story in this chapter is much more interesting on the syntactic side.

Generic Operators

Let us begin by developing some generic operators for building quotient lenses out of basic lenses.

Lift Intuitively, it should be clear that quotient lenses are a generalization of basic lenses. The *lift* operator, which converts a basic lens to a quotient lens, witnesses this fact.

$$\boxed{\frac{l \in S \Longleftrightarrow V}{\mathit{lift}\ l \in S/{=} \Longleftrightarrow V/{=}}}$$

$$\boxed{\begin{array}{ll} \mathit{get}\ s & = l.\mathit{get}\ s \\ \mathit{put}\ v\ s & = l.\mathit{put}\ v\ s \\ \mathit{create}\ v & = l.\mathit{create}\ v \end{array}}$$

3.2.1 Lemma: Let $l \in S \Longleftrightarrow V$ be a basic lens. Then *lift* l is a quotient lens in $S/{=} \Longleftrightarrow V/{=}$.

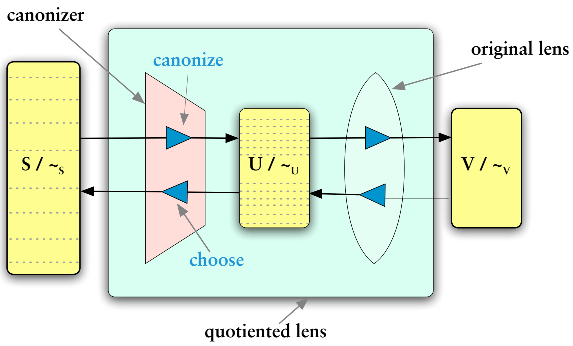


Figure 3.2: Adding a canonizer to a quotient lens (on the left)

The *get*, *put*, and *create* components of *lift* l are identical to the components of l. The equivalence relations on the source and view are both equality. The proof that *lift* l is a well-behaved quotient lens follows immediately from the basic lens laws for l.

Left and Right Quotient The next two operators provide a way to loosen up a lens, widening the set of structures that it can handle and coarsening the equivalence on the source and view accordingly. These operators are our primary means of building quotient lenses with non-trivial equivalence relations in their types.

To illustrate how quotienting works, consider a simple example. Suppose that l is a quotient lens from $U/{\sim_U}$ to $V/{\sim_V}$ where $\sim_U$ is a relatively fine equivalence—e.g., l could be a lifted basic lens with U a set of "canonical strings" containing no extraneous whitespace and $\sim_U$ could be equality. We want to build a new quotient lens whose source type is some larger set S—e.g., the same set of strings, except with arbitrary whitespace in various places—and whose equivalence $\sim_S$ relates strings that only differ in whitespace. To get back and forth between S and U, we need two functions. The first, called *canonize*, maps elements of S to their "canonical representatives" in U—e.g., by discarding extra whitespace. The other, called *choose*, maps canonical representatives in U to an element in its inverse image under *canonize* in S—e.g., the canonical string itself, or perhaps a pretty printed version with whitespace inserted according to a layout convention. Together, the *canonize* and *choose* functions form a *canonizer*—see Figure 3.2.

Clearly, a canonizer is a bit like a lens (minus the *put* component). However, canonizers only need to obey one round-tripping law.

3.2.2 Definition [Canonizer]: Let $S \subseteq \mathcal{U}$ and $U \subseteq \mathcal{U}$ be sets of objects and let $\sim_U$ be an equivalence relation on U.[3] A canonizer q from S to $U/{\sim_U}$ comprises two functions

$$\begin{array}{rcl} q.\mathit{canonize} & \in & S \to U \\ q.\mathit{choose} & \in & U \to S \end{array}$$

obeying

$$q.\mathit{canonize}\ (q.\mathit{choose}\ u) \sim_U u \qquad \text{(ReCanonize)}$$

for every $u \in U$. That is, *canonize* is a left inverse of *choose* modulo $\sim_U$. The set of all canonizers from S to $U/{\sim_U}$ is written $S \leftrightarrow\!\!\!\!\!\!\!+ U/{\sim_U}$.

[3] We keep track of the equivalence on U because when we quotient a lens by a canonizer we need the equivalences to match up. However, we do not keep track of the equivalence on S because it will be calculated in the typing rule.

A lens can be quotiented in two ways—on the left, which treats part of the source as ignorable, or on the right, which treats part of view as ignorable. The left quotient operator, *lquot*, takes a canonizer q and a quotient lens l as arguments and yields a quotient lens whose source type is coarsened using q.

$$\frac{\begin{array}{c} q \in S \mathrel{\leftarrow\!\!\!\boxminus\!\!\!\twoheadrightarrow} U/{\sim_U} \\ l \in U/{\sim_U} \Longleftrightarrow V/{\sim_V} \\ {\sim_S} \triangleq \{(s,s') \in S \times S \mid q.\mathit{canonize}\ s \sim_U q.\mathit{canonize}\ s'\} \end{array}}{\mathit{lquot}\ q\ l \in S/{\sim_S} \Longleftrightarrow V/{\sim_V}}$$

$$\begin{array}{lcl} \mathit{get}\ s & = & l.\mathit{get}\ (q.\mathit{canonize}\ s) \\ \mathit{put}\ v\ s & = & q.\mathit{choose}\ (l.\mathit{put}\ v\ (q.\mathit{canonize}\ s)) \\ \mathit{create}\ v & = & q.\mathit{choose}\ (l.\mathit{create}\ v) \end{array}$$

3.2.3 Lemma: Let q be a canonizer $S \mathrel{\leftarrow\!\!\!\boxminus\!\!\!\twoheadrightarrow} U/{\sim_U}$ and let l be a quotient lens in $U/{\sim_U} \Longleftrightarrow V/{\sim_V}$. Then *lquot q l* is a quotient lens in $S/{\sim_S} \Longleftrightarrow V/{\sim_V}$ where $s \sim_S s'$ if and only if $q.\mathit{canonize}\ s \sim_U q.\mathit{canonize}\ s'$.

The *get* component of *lquot* canonizes the source string using $q.\mathit{canonize}$ and then maps the canonical version to a view using $l.\mathit{get}$. The *put* function canonizes its source argument using $q.\mathit{canonize}$ and puts the view into the resulting U using $l.\mathit{put}$. To produce the final source it uses $q.\mathit{choose}$. The *create* function is similar. The equivalence relation $\sim_S$ on the source is the relation induced by $q.\mathit{canonize}$ and $\sim_U$—i.e., two sources are equivalent if $q.\mathit{canonize}$ maps them to elements of U related by $\sim_U$.

To illustrate left quotienting, recall the *del* E lens, which deletes a string in the *get* direction and restores it in the *put* direction. In many situations, this is the behavior we want. However, if the data being deleted is "unimportant"—e.g., whitespace—we might prefer to have the *put* function produce a particular source instead. This would not be a well-behaved basic lens because it would violate GetPut, but it is easy to have as a quotient lens.

```
let qdel (E:regexp) (e:string in E) : lens =
  lquot
    (canonizer_of_lens (del_default E e))
    (copy "")
```

The $\mathit{canonizer\_of\_lens}$ combinator is defined later in this section. It converts a quotient lens to a canonizer, using its *get* function for *canonize* and its *create* function for *choose*. The *get* component of *qdel* E e first maps the source to the empty string using the canonizer and then copies the empty string to the view while *put* function copies the empty string and then invokes the canonizer's *choose* function, which produces e as the final result. As an example, here is a lens that deletes an upper-case letter in the *get* direction and produces "Z" in the *put* direction:

```
test (qdel [A-Z] "Z").get "A" = ""
test (qdel [A-Z] "Z").put "" into "A" = "Z"
```

The type of *qdel* E e is $[\![E]\!]/\mathsf{Tot}([\![E]\!]) \Longleftrightarrow \{\epsilon\}/{=}$, which records the fact that the lens treats every source string as equivalent. The equivalence $\mathsf{Tot}([\![E]\!])$ is $[\![E]\!] \times [\![E]\!]$, the total relation on $[\![E]\!]$.

We often use `qdel` to standardize whitespace. For example, here is a variant of the `xml_elt` helper function we defined previously that produces pretty-printed XML in the *put* direction. It takes two extra arguments, `ws1` and `ws2`. The string `ws1` controls the amount of whitespace before opening tags and `ws2` controls the whitespace before closing tags.

```
let full_xml_elt
  (t:string) (ws1:string) (ws2:string) (l:lens) : lens =
    qdel WHITESPACE* ws1
  . del ("<" . t . ">")
  . l
  . qdel WHITESPACE* ws2
  . del ("</" . t . ">")
```

We can instantiate full_xml_elt to obtain several other helper functions:

```
let xml_elt (t:string) (ws:string) (l:lens) : lens =
  full_xml_elt t ws ws l
let xml_pcdata_elt (t:string) (ws:string) (l:lens) : lens =
  full_xml_elt t ws "" l
let xml_outer_elt (t:string) (ws:string) (l:lens) : lens =
  full_xml_elt t "" ws l
```

The xml_elt helper takes just one extra argument that controls the whitespace before both kinds of tags. The xml_pcdata_elt helper handles elements containing PCDATA. It adds the specified whitespace before the opening tag but adds nothing before the closing tag. The xml_outer_elt helper handles the outer-most element in a document. It adds no whitespace before the opening tag and the specified whitespace before the closing tag. Using these helpers, we can rewrite the composer lens so that it produces pretty-printed XML in the *put* direction:

```
let composer : lens =
  xml_elt "composer" INDENT1
    ( xml_pcdata_elt "name" INDENT2
        (copy (ALPHA . " " . ALPHA) )
    . ins ", "
    . xml_pcdata_elt "dates" INDENT2
        (copy (YEAR . "-" . YEAR) )
    . xml_pcdata_elt "nationality" INDENT2
        ( del_default ALPHA "Unknown" ) )
let composers : lens =
  xml_outer_elt "composers" INDENT0
    ( copy EPSILON | composer . (ins NEWLINE . composer)* )
```

The strings INDENT0, INDENT1, and INDENT2, etc. contain increasing amounts of whitespace. In the *get* direction this lens discards the whitespace between XML elements, just like the original version. However, in the *put* direction, it produces pretty-printed XML—e.g., putting

```
Jean Sibelius, 1865-1957
Aaron Copland, 1910-1990
Benjamin Britten, 1913-1976
```

into a source with no indentation

```
<composers>
<composer>
<name>Jean Sibelius</name>
<dates>1865-1957</dates>
```

```
<nationality>Finnish</nationality>
</composer>
<composer>
<name>Aaron Copland</name>
<dates>1910-1990</dates>
<nationality>American</nationality>
</composer>
<composer>
<name>Benjamin Briten</name>
<dates>1913-1976</dates>
<nationality>English</nationality>
</composer>
</composers>
```

yields a pretty-printed result

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <dates>1865-1957</dates>
    <nationality>Finnish</nationality>
  </composer>
  <composer>
    <name>Aaron Copland</name>
    <dates>1910-1990</dates>
    <nationality>American</nationality>
  </composer>
  <composer>
    <name>Benjamin Britten</name>
    <dates>1913-1976</dates>
    <nationality>English</nationality>
  </composer>
</composers>
```

that also reflects the change made to the view. The original version of the lens would have restored the whitespace from the source.

Right Quotient The right quotient operator, *rquot*, is symmetric to *lquot*. It quotients a lens l in $S/{\sim_S} \Longleftrightarrow U/{\sim_U}$ on the right using a canonizer q from V to $U/{\sim_U}$. Note that compared to *lquot*, the canonizer is applied in the opposite direction. If we think of a canonizer as a weaker form of a lens, then *lquot* is essentially just lens composition, while *rquot* is a kind of "head to head" composition—i.e., composing a function that discards information in the forward direction with a function that discards information in the reverse direction—that would not make sense with basic lenses.

$$\frac{\begin{array}{c} l \in S/{\sim_S} \Longleftrightarrow U/{\sim_U} \\ q \in V \mathrel{\leftarrow\!\!\boxminus\!\!\rightarrow} U/{\sim_U} \\ {\sim_V} \triangleq \{(v, v') \in V \times V \mid q.\mathit{canonize}\ v \sim_U q.\mathit{canonize}\ v'\} \end{array}}{\mathit{rquot}\ l\ q \in S/{\sim_S} \Longleftrightarrow V/{\sim_V}}$$

$$\begin{aligned} \mathit{get}\ s &= q.\mathit{choose}\ (l.\mathit{get}\ s) \\ \mathit{put}\ v\ s &= l.\mathit{put}\ (q.\mathit{canonize}\ v)\ s \\ \mathit{create}\ v &= l.\mathit{create}\ (q.\mathit{canonize}\ v) \end{aligned}$$

3.2.4 Lemma: Let l be a quotient lens in $S/{\sim_S} \Longleftrightarrow U/{\sim_U}$ and q a canonizer in $V \mathrel{\leftarrow\!\!\boxminus\!\!\rightarrow} U/{\sim_U}$. Then *rquot* l q is a quotient lens in $S/{\sim_S} \Longleftrightarrow V/{\sim_V}$ where $v{\sim_V}v'$ if and only if $q.\mathit{canonize}\, v \sim_U q.\mathit{canonize}\, v'$.

The *get* function first transforms the source to an intermediate view using $l.\mathit{get}$ and then selects a representative using $q.\mathit{choose}$. The *put* and *create* functions canonize the view using $q.\mathit{canonize}$ and then use $l.\mathit{put}$ or $l.\mathit{create}$ to compute the updated source.

Recall the basic lens *ins* e, which inserts a fixed string e into the view. The quotient lens, *qins* E e behaves like *ins* e in the *get* direction but accepts any string in $[\![E]\!]$ in the *put* direction. We often use *qins* to insert formatting into the view—e.g., if we flipped the composers lens around so that the source was ASCII and the view was XML, we would use *qins* to insert the XML formatting. It is straightforward to define *qins* using *rquot*:

```
let qins (E:regexp) (e:string) : lens =
  rquot
    (ins e)
    (canonizer_of_lens (E <-> e))
```

The type of *qins* E e is $\{\epsilon\}/{=} \Longleftrightarrow [\![E]\!]/\mathsf{Tot}([\![E]\!])$. Although it does not obey the strict version of PutGet, it does obey it modulo the equivalence relation specified in its type. To illustrate this point, consider the following examples

```
let l : lens = copy ALPHA . qins " "* "" . copy (" " . ALPHA)
let s : string = "Aaron Copland"
test l.put "Aaron Copland" into s = s
test l.put "Aaron  Copland" into s = s
test l.put "Aaron   Copland" into s = s
```

and observe that the *put* function maps views that differ only in the amount of whitespace between names to the same source.

Using *qdel* and *qins*, is straightforward to define a quotient lens version of the constant lens:

```
let qconst
  (u:string) (E:regexp) (D:regexp) (v:string) : lens =
    qdel E u . qins D v
```

In the *get* direction, *qconst* takes an E and maps it to v. In the *put* direction, it takes a D and maps it to u. Its type, $[\![E]\!]/\mathsf{Tot}([\![E]\!]) \Longleftrightarrow [\![D]\!]/\mathsf{Tot}([\![D]\!])$, records the fact that it treats all sources and all views as equivalent.

Subsumption The *lquot* and *rquot* operators allow us to quotient a quotient lens repeatedly on either side, which has the effect of composing canonizers. We do this often in quotient lens programs—stacking up several canonizers, each of which canonizes a distinct aspect of the concrete or abstract structures. The following rule of subsumption is often useful:

$$\boxed{\frac{\begin{array}{c}\sim_U \text{ is a refinement of } \sim_{U'} \\ q \in V \mathrel{\leftarrow\!\!\!\!\triangleright\!\!\!\!\rightarrow} U/\!\sim_U\end{array}}{q \in V \mathrel{\leftarrow\!\!\!\!\triangleright\!\!\!\!\rightarrow} U/\!\sim_{U'}}}$$

3.2.5 Lemma: Let $q \in V \mathrel{\leftarrow\!\!\!\!\triangleright\!\!\!\!\rightarrow} U/\!\sim_U$ be a canonizer and let $\sim_{U'}$ be an equivalence relation on U such that $\sim_{U'}$ is a refinement of $\sim_U$. Then q is also a canonizer in $V \mathrel{\leftarrow\!\!\!\!\triangleright\!\!\!\!\rightarrow} U/\!\sim_{U'}$.

This rule allows the equivalence relation component of a canonizer's type to be coarsened. For example, if we want to quotient a lens $l \in U/\!\sim_U \iff V/\!\sim_V$ on the left using a canonizer $q \in S \mathrel{\leftarrow\!\!\!\!\triangleright\!\!\!\!\rightarrow} U/ =$, we can use this typing rule to promote q to $S \mathrel{\leftarrow\!\!\!\!\triangleright\!\!\!\!\rightarrow} U/\!\sim_U$.

Composition The next operator puts two quotient lenses in sequence.

$$\boxed{\frac{\begin{array}{c}l_1 \in S/\!\sim_S \iff U/\!\sim_U \\ l_2 \in U/\!\sim_U \iff V/\!\sim_V\end{array}}{l_1;l_2 \in S/\!\sim_S \iff V/\!\sim_V}}$$

$$\boxed{\begin{array}{lcl}\mathit{get}\ s & = & l_2.\mathit{get}\ (l_1.\mathit{get}\ s) \\ \mathit{put}\ v\ s & = & l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s \\ \mathit{create}\ v & = & l_1.\mathit{create}\ (l_2.\mathit{create}\ v)\end{array}}$$

3.2.6 Lemma: Let $l_1 \in S/\!\sim_S \iff U/\!\sim_U$ and $l_2 \in U/\!\sim_U \iff V/\!\sim_V$ be quotient lenses. Then $l_1;l_2$ is a quotient lens in $S/\!\sim_S \iff V/\!\sim_V$.

The components of $l_1;l_2$ are identical to the ones in the basic lens composition operator described in the last chapter. However, they typing rule demands that the view type of l_1 and the source type of l_2 have the same equivalence relation $\sim_U$. This raises an interesting implementation issue: to statically typecheck the composition operator, we need to be able to decide whether two equivalence relations are identical—see Section 3.4. To see what goes wrong if this condition is dropped, consider

$$\begin{array}{lcllcl} l_1 & = & \mathit{copy}\ \{a\} & \in & \{a\}/\!= \iff \{a\}/\!= \\ l_2 & = & \mathit{copy}\ (\{a\} \mid \{b\}) & \in & \{a,b\}/\!= \iff \{a,b\}/\!= \end{array}$$

and $q \in \{a,b\} \mathrel{\leftarrow\!\!\!\!\triangleright\!\!\!\!\rightarrow} \{a\}/\!=$ defined by

$$\begin{array}{lcl} q.\mathit{canonize}\ \_ & = & a \\ q.\mathit{choose}\ a & = & a \end{array}$$

If we let $l = (\mathit{rquot}\ l_1\ q);\ l_2$, where the equivalence on the left is the total relation on $\{a,b\}$ while the relation on the right is equality, then PutGet fails:

$$\begin{array}{ll} l.\mathit{get}\ (l.\mathit{put}\ b\ b) & = l.\mathit{get}\ a \\ & = l_2.\mathit{get}\ (q.\mathit{choose}\ (l_1.\mathit{get}\ a)) \\ & = a \\ & \neq b \end{array}$$

Conversely, if we let $l = l_2;\ (\mathit{lquot}\ q\ l_1)$, where the left equivalence is equality and the right equivalence is the total relation on $\{a, b\}$, then GetPut fails. First, note that

$$\begin{aligned}
l.\mathit{get}\ b\ &= (\mathit{lquot}\ q\ l_2).\mathit{get}(l_2.\mathit{get}\ b)\\
&= (l_2.\mathit{get}\ (q.\mathit{canonize}\ (l_2.\mathit{get}\ b))\\
&= (l_2.\mathit{get}\ (q.\mathit{canonize}\ b)\\
&= (l_2.\mathit{get}\ a)\\
&= a
\end{aligned}$$

Then, using this equality, calculate as follows:

$$\begin{aligned}
l.\mathit{put}\ (l.\mathit{get}\ b)\ b\ &= l.\mathit{put}\ a\ b\\
&= l_2.\mathit{put}\ (q.\mathit{choose}\ (l_1.\mathit{put}\ a\ (q.\mathit{canonize}\ (l_2.\mathit{get}\ b))))\ b\\
&= l_2.\mathit{put}\ a\ b\\
&= a\\
&\neq b
\end{aligned}$$

Intuitively, these failures make sense. It would be surprising if composition somehow managed to respect the equivalences on the source and view even though l_1 and l_2 disagreed about the equivalence relation in the middle.

Canonizer So far, we have seen how to lift basic lenses to quotient lenses, how to coarsen the equivalence relations in the types of quotient lenses using canonizers, and how to compose them. We have not, however, discussed where canonizers come from! Of course, we can always define them as primitives—this is essentially the approach used in previous "canonizers at the perimeters" proposals, where the set of viewers (i.e., parsers and pretty printers) is fixed. But here we can do something much better: we can construct a canonizer using the *get* and *create* components of an arbitrary lens—indeed, an arbitrary quotient lens!

$$\frac{l \in S/{\sim_S} \Longleftrightarrow U/{\sim_U}}{\mathit{canonizer\_of\_lens}\ l \in S \mathrel{\leftarrow\!\!\!\!\mapsto} U/{\sim_U}}$$

$$\begin{aligned}
\mathit{canonize}\ s &= l.\mathit{get}\ s\\
\mathit{choose}\ u &= \ \ l.\mathit{create}\ u
\end{aligned}$$

3.2.7 Lemma: Let l be a quotient lens in $S/{\sim_S} \Longleftrightarrow U/{\sim_U}$. Then the canonizer $\mathit{canonizer\_of\_lens}\ l$ is in $S \mathrel{\leftarrow\!\!\!\!\mapsto} U/{\sim_U}$.

Building canonizers out of lenses gives us a pleasingly parsimonious design, making it possible to define canonizers using whatever generic or domain-specific operators are already available for lenses. For example, a composition operator on canonizers can be derived from the quotienting operators. The type follows directly from the types of the *copy*, *lquot*, and *canonizer_of_lens* operators.

$$\frac{q_1 \in S \mathrel{\leftarrow\!\!\!\!\mapsto} U/{=} \qquad q_2 \in U \mathrel{\leftarrow\!\!\!\!\mapsto} V/{=}}{(q_1; q_2) \in S \mathrel{\leftarrow\!\!\!\!\mapsto} V/{=}}$$

$$\begin{aligned}
(q_1; q_2) \triangleq\ &\mathit{canonizer\_of\_lens}\\
&\quad (\mathit{lquot}\ q_1\ (\mathit{lquot}\ q_2\ (\mathit{copy}\ (\mathit{canonized\_type}\ q_2))))
\end{aligned}$$

In general, the equivalence on U does not need to be the identity, but it must refine the equivalence induced by q_2. Of course, it is also interesting to design primitive canonizers from scratch. The single canonizer law imposes fewer restrictions than the lens laws, so we have considerable latitude for developing canonizing transformations that would not be valid as lenses.

Regular Operators on Quotient Lenses

Having defined the semantic space of quotient lenses and several generic operators, we now focus our attention on quotient lenses for strings. The combinators presented in this section are direct generalizations of the corresponding basic lens combinators defined in the previous chapter. In particular, the *get*, *put*, and *create* components of each operator are identical to the basic lens versions. To save space, we do not repeat their definitions. The typing rules, however, are different as they define equivalence relations on the source and view.

Concatenation As in the previous chapter, let us start with concatenation, which is the simplest regular operator. Before we can define the lens, we need to lift concatenation to relations:

3.2.8 Definition [Relation Concatenation]: Let L_1 and L_2 be languages and let R_1 and R_2 be binary relations on L_1 and L_2 respectively. The concatenation of R_1 and R_2 is the relation defined as follows: $w\ (R_1{\cdot}R_2)\ w'$ if and only if there exist strings $w_1, w_1' \in L_1$ and $w_2, w_2' \in L_2$ with $w = w_1{\cdot}w_2$ and $w' = w_1'{\cdot}w_2'$ such that $w_1\ R_1\ w_1'$ and $w_2\ R_2\ w_2'$.

We need to be careful when we concatenate equivalence relations because, in general, the concatenation operator does not preserve transitivity. Thus, the concatenation of two equivalence relations $\sim_1$ and $\sim_2$ may not be an equivalence. However, it will be an equivalence in two important cases

1. if the concatenation of L_1 and L_2 is unambiguous or
2. if $\sim_1$ and $\sim_2$ are both the identity relation.

Since the typing rule of the concatenation lens ensures that the concatenations of the underlying languages are unambiguous, the concatenation of the equivalence relations on the source and view are also equivalences.

With this definition in place, the typing rule for concatenation is simply:

$$\boxed{\dfrac{\begin{array}{ll} l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}} & S_1{\cdot^{!}}S_2 \\ l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}} & V_1{\cdot^{!}}V_2 \end{array}}{l_1{\cdot}l_2 \in (S_1{\cdot}S_2)/({\sim_{S_1}}{\cdot}{\sim_{S_2}}) \Longleftrightarrow (V_1{\cdot}V_2)/({\sim_{V_1}}{\cdot}{\sim_{V_2}})}}$$

3.2.9 Lemma: Let $l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}}$ and $l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}}$ be quotient lenses such that $S_1{\cdot^{!}}S_2$ and $V_1{\cdot^{!}}V_2$. Then $l_1{\cdot}l_2$ is a quotient lens in $(S_1{\cdot}S_2)/({\sim_{S_1}}{\cdot}{\sim_{S_2}}) \Longleftrightarrow (V_1{\cdot}V_2)/({\sim_{V_1}}{\cdot}{\sim_{V_2}})$.

Concatenation raises an interesting point: suppose that we have canonizers q_1 and q_2 and quotient lenses l_1 and l_2 that we want to—in some order—concatenate and quotient on the left. There are two ways we could do this: quotient l_1 and l_2 first using q_1 and q_2 and combine the results by concatenating the resulting quotient lenses, or concatenate the quotient lenses and the canonizers first and then quotient the results. Both constructions are possible in our framework and they yield equivalent quotient lenses (when they are well-typed)[4]. We define the concatenation operator on canonizers later in this chapter and prove this fact—see Lemma 3.2.15.

[4] Quotienting first is slightly more flexible, since the concatenation of the original quotient lenses need not be unambiguous.

Kleene Star As with concatenation, before we can define the quotient version of Kleene star, we need to lift iteration to relations:

3.2.10 Definition [Relation Iteration]: Let L be a regular language, and let R be a binary relation on L. The iteration of R is defined as follows: $w \; R^{*} \; w'$ if and only if there exist strings w_1 to w_n and w'_1 to w'_n in L with $w = (w_1 \cdots w_n)$ and $w' = (w'_1 \cdots w'_n)$ such that $w_i \; R \; w'_i$ for every i from 1 to n.

The iteration of an equivalence relation is not an equivalence in general, but it is when the underlying language is unambiguously iterable or when the relation being iterated is the identity.

The typing rule for the Kleene star operator is as follows:

$$\boxed{\frac{\begin{array}{c} S^{!*} \qquad V^{!*} \\ l \in S/{\sim_S} \iff V/{\sim_V} \end{array}}{l^* \in S^*/{\sim_S}^* \iff V^*/{\sim_V}^*}}$$

3.2.11 Lemma: Let $l \in S/{\sim_S} \iff V/{\sim_V}$ be a quotient lens such that $S^{!*}$ and $V^{!*}$. Then l^* is a quotient lens in $S^*/{\sim_S}^* \iff V^*/{\sim_V}^*$.

Union Some care is needed in designing a sound typing rule for the union operator. In particular, because the view types may overlap, we need to be sure that the equivalence relations of the sublenses relate the same strings in the intersection.

$$\boxed{\frac{\begin{array}{c} S_1 \cap S_2 = \emptyset \\ l_1 \in S_1/{\sim_{S_1}} \iff V_1/{\sim_{V_1}} \\ l_2 \in S_2/{\sim_{S_2}} \iff V_2/{\sim_{V_2}} \\ \forall v, v' \in V_1 \cap V_2.\ v \sim_{V_1} v' \text{ if and only if } v \sim_{V_2} v' \end{array}}{l_1 \,|\, l_2 \in (S_1 \cup S_2)/({\sim_{S_1}} \cup {\sim_{S_2}}) \iff (V_1 \cup V_2)/({\sim_{V_1}} \cup {\sim_{V_2}})}}$$

3.2.12 Lemma: Let $l_1 \in S_1/{\sim_{S_1}} \iff V_1/{\sim_{V_1}}$ and $l_2 \in S_2/{\sim_{S_2}} \iff V_2/{\sim_{V_2}}$ be quotient lenses such that $S_1 \cap S_2 = \emptyset$ and for every v and v' in $V_1 \cap V_2$ we have $v \sim_{V_1} v'$ if and only if $v \sim_{V_2} v'$. Then the quotient lens $l_1 \,|\, l_2$ is in $(S_1 \cup S_2)/({\sim_{S_1}} \cup {\sim_{S_2}}) \iff (V_1 \cup V_2)/({\sim_{V_1}} \cup {\sim_{V_2}})$.

The equivalence relations $\sim_S$ and $\sim_V$ are defined as the union of the corresponding equivalences from l_1 and l_2. The side conditions in the typing rule ensure that these are equivalences. Additionally, the side condition on $\sim_V$ is essential for ensuring soundness. To see what would go wrong if we did not include it, let v be a string in $V_1 \cap V_2$ and let v' be a string in $V_2 - V_1$ with $v \sim_V v'$ and let s be a string in S_1 with $(l_1 \,|\, l_2).\mathit{get}\ s \sim_V v$. By the definition of $l_1 \,|\, l_2$, we have the following equalities:

$$\begin{array}{rcll} (l_1 \,|\, l_2).\mathit{put}\ v\ s & = & l_1.\mathit{put}\ v\ c & \quad \text{as } v \in V_1 \cap V_2 \text{ and } s \in S_1 \\ (l_1 \,|\, l_2).\mathit{put}\ v'\ s & = & l_2.\mathit{create}\ v'\ c & \quad \text{as } v \in V_2 \cap V_1 \text{ and } s \in S_1 \end{array}$$

By the PutEquiv law we also have:

$$(l_1 \,|\, l_2).\mathit{put}\ v\ s \sim_S (l_1 \,|\, l_2).\mathit{put}\ v'\ s.$$

But this is a contradiction—the codomain of $l_1.\mathit{put}$ is S_1 and the codomain of $l_2.\mathit{create}$ is S_2 and the typing rule stipulates that these sets must be disjoint. Hence, the two strings cannot be related by ${\sim_{S_1}} \cup {\sim_{S_2}}$.

Swap The typing rule for the swap operator is a straightforward generalization of the typing rule for concatenation:

$$\frac{\begin{array}{cc} l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}} & S_1 \cdot^{!} S_2 \\ l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}} & V_2 \cdot^{!} V_1 \end{array}}{l_1 \sim l_2 \in (S_1 \cdot S_2)/({\sim_{S_1}} \cdot {\sim_{S_2}}) \Longleftrightarrow (V_2 \cdot V_1)/({\sim_{V_2}} \cdot {\sim_{V_1}})}$$

3.2.13 Lemma: Let $l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}}$ and $l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}}$ be quotient lenses such that $S_1 \cdot^{!} S_2$ and $V_2 \cdot^{!} V_1$. Then $l_1 \sim l_2$ is a quotient lens in $(S_1 \cdot S_2)/({\sim_{S_1}} \cdot {\sim_{S_2}}) \Longleftrightarrow (V_2 \cdot V_1)/({\sim_{V_2}} \cdot {\sim_{V_1}})$.

Regular Operators on Canonizers

Now we define canonizer versions of the regular operators. Since canonizers only have to satisfy the ReCanonize law, we have some additional flexibility compared to the corresponding lens operators.

Concatenation Recall that the concatenation operator on quotient lenses requires that the concatenations of the source and view types each be unambiguous. With canonizers, we only need the concatenation on the source side to be unambiguous. The notation $\mathsf{TrClose}(R)$ used below denotes the smallest transitive relation that contains R:

$$\frac{\begin{array}{c} S_1 \cdot^{!} S_2 \qquad p \in \Pi u : (U_1 \cdot U_2).\ \{(u_1, u_2) \in U_1 \times U_2 \mid u_1 \cdot u_2 = u\} \\ q_1 \in S_1 \mathrel{\leftrightarrow\!\!\!\!\!\square} U_1/{\sim_{U_1}} \\ q_2 \in S_2 \mathrel{\leftrightarrow\!\!\!\!\!\square} U_2/{\sim_{U_2}} \end{array}}{q_1 \cdot_p q_2 \in (S_1 \cdot S_2) \mathrel{\leftrightarrow\!\!\!\!\!\square} (U_1 \cdot U_2)/\mathsf{TrClose}({\sim_{U_1}} \cdot {\sim_{U_2}})}$$

$$\begin{array}{lll} \mathit{canonize}\ (s_1 \cdot s_2) & = & (q_1.\mathit{canonize}\ s_1) \cdot (q_2.\mathit{canonize}\ s_2) \\ \mathit{choose}\ u & = & (q_1.\mathit{choose}\ u_1) \cdot (q_2.\mathit{choose}\ u_2) \\ & & \quad \text{where } p\ u = (u_1, u_2) \end{array}$$

3.2.14 Lemma: Let $q_1 \in S_1 \mathrel{\leftrightarrow\!\!\!\!\!\square} U_1/{\sim_{U_1}}$ and $q_2 \in S_2 \mathrel{\leftrightarrow\!\!\!\!\!\square} U_2/{\sim_{U_2}}$ be canonizers such that $S_1 \cdot^{!} S_2$ and let p be a function in:

$$\Pi u : (U_1 \cdot U_2).\ \{(u_1, u_2) \in U_1 \times U_2 \mid u_1 \cdot u_2 = u\}$$

Then $q_1 \cdot_p q_2$ is a canonizer in $(S_1 \cdot S_2) \mathrel{\leftrightarrow\!\!\!\!\!\square} (U_1 \cdot U_2)/\mathsf{TrClose}({\sim_{U_1}} \cdot {\sim_{U_2}})$.

The function p determines how strings in the concatenation $U_1 \cdot U_2$, which may be ambiguous, are split. The dependent type for p ensures that it produces substrings that belong to U_1 and U_2. In examples, we will elide p when the concatenation of U_1 and U_2 is unambiguous. We take the transitive closure of ${\sim_{U_1}} \cdot {\sim_{U_2}}$ to ensure that it is an equivalence relation.

Using this definition, we can now prove the result we discussed earlier: canonizing and quotienting (on the left) in either order yields equivalent quotient lenses

3.2.15 Lemma: Let

$$\begin{array}{lcl@{\qquad\qquad}lcl} l_1 & \in & U_1/{\sim_{U_1}} \Longleftrightarrow V_1/{\sim_{V_1}} & q_1 & \in & S_1 \mathrel{\leftrightarrow\!\!\!\!\!\square} U_1/{\sim_{U_1}} \\ l_2 & \in & U_2/{\sim_{U_2}} \Longleftrightarrow V_2/{\sim_{V_2}} & q_2 & \in & S_2 \mathrel{\leftrightarrow\!\!\!\!\!\square} U_2/{\sim_{U_2}} \end{array}$$

be quotient lenses and canonizers and suppose that

$$\begin{array}{lcl} l & \triangleq & (\mathit{lquot}\ q_1\ l_1) \cdot (\mathit{lquot}\ q_2\ l_2) \\ l' & \triangleq & \mathit{lquot}\ (q_1 \cdot q_2)\ (l_1 \cdot l_2) \end{array}$$

are well formed according to the typing rules given in this section. Then l and l' are equivalent quotient lenses.

Proof: Let l_1 and l_2 be quotient lenses and q_1 and q_2 be canonizers and define quotient lenses l and l' as stated above and suppose that l and l' are well typed. By the typing derivations for l and l' we have that $V_1 \cdot^! V_2$ and $U_1 \cdot^! U_2$. Using these facts, we prove that the components of l and l' are equivalent:

▶ ***get***: Let $s = s_1 \cdot s_2 \in S_1 \cdot S_2$ be a source. We calculate as follows

$$\begin{array}{ll}
\quad l.\mathit{get}\ (s_1 \cdot s_2) & \\
= ((\mathit{lquot}\ q_1\ l_1) \cdot (\mathit{lquot}\ q_2\ l_2)).\mathit{get}\ (s_1 \cdot s_2) & \text{by definition } l \\
= ((\mathit{lquot}\ q_1\ l_1).\mathit{get}\ s_1) \cdot ((\mathit{lquot}\ q_2\ l_2).\mathit{get}\ s_2) & \text{by definition } (\cdot) \\
= (l_1.\mathit{get}\ (q_1.\mathit{canonize}\ s_1)) \cdot (l_2.\mathit{get}\ (q_2.\mathit{canonize}\ s_2)) & \text{by definition } \mathit{lquot} \\
= (l_1 \cdot l_2).\mathit{get}\ ((q_1 \cdot q_2).\mathit{canonize}\ (s_1 \cdot s_2)) & \text{as } U_1 \cdot^! U_2 \\
= (\mathit{lquot}\ (q_1 \cdot q_2)\ (l_1 \cdot l_2)).\mathit{get}\ (s_1 \cdot s_2) & \text{by definition } \mathit{lquot} \\
= l'.\mathit{get}\ (s_1 \cdot s_2) & \text{by definition } l'
\end{array}$$

and obtain the required equality.

▶ ***put***: Let $v = v_1 \cdot v_2 \in V_1 \cdot V_2$ be a view and $s = s_1 \cdot s_2 \in S_1 \cdot S_2$ a source. We calculate as follows

$$\begin{array}{ll}
\quad l.\mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) & \\
= ((\mathit{lquot}\ q_1\ l_1) \cdot (\mathit{lquot}\ q_2\ l_2)).\mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) & \text{by definition } l \\
= ((\mathit{lquot}\ q_1\ l_1).\mathit{put}\ v_1\ s_1) \cdot ((\mathit{lquot}\ q_1\ l_1).\mathit{put}\ v_2\ s_2) & \text{by definition } (\cdot) \\
= (q_1.\mathit{choose}\ (l_1.\mathit{put}\ v_1\ (q_1.\mathit{canonize}\ s_1))) \cdot & \\
\quad (q_2.\mathit{choose}\ (l_2.\mathit{put}\ v_2\ (q_2.\mathit{canonize}\ s_2))) & \text{by definition } \mathit{lquot} \\
= (q_1 \cdot q_2).\mathit{choose} & \\
\qquad ((l_1 \cdot l_2).\mathit{put}\ (v_1 \cdot v_2)\ ((q_1 \cdot q_2).\mathit{canonize}\ (s_1 \cdot s_2))) & \text{as } U_1 \cdot^! U_2 \\
= (\mathit{lquot}\ (q_1 \cdot q_2)\ (l_1 \cdot l_2)).\mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) & \text{by definition } \mathit{lquot} \\
= l'.\mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) & \text{by definition } l'
\end{array}$$

and obtain the required equality.

▶ ***create***: Similar to the proof for *put*. □

Kleene Star The iteration operator on canonizers is similar:

$$\frac{S^{!*} \qquad p \in \Pi u : U^*.\ \{[u_1, \ldots, u_n] \in U\ \mathit{list} \mid u_1 \cdots u_n = u\} \qquad q \in S \stackrel{\Box}{\longleftrightarrow} U/{\sim_U}}{q^{*p} \in S^* \stackrel{\Box}{\longleftrightarrow} U^*/\mathsf{TrClose}({\sim_U}^*)}$$

$$\begin{array}{lll}
\mathit{canonize}\ s_1 \cdots s_n & = & (q.\mathit{canonize}\ s_1) \cdots (q.\mathit{canonize}\ s_n) \\
\mathit{choose}\ u & = & (q.\mathit{choose}\ u_1) \cdots (q.\mathit{choose}\ u_n) \\
 & & \quad \text{where } p\ u = [u_1, \ldots, u_n]
\end{array}$$

3.2.16 Lemma: Let $q \in S \stackrel{\Box}{\longleftrightarrow} U/{\sim_U}$ be a canonizer such that $S^{!*}$. Also let p be a function in:

$$\Pi u : U^*.\ \{[u_1, \ldots, u_n] \in U\ \mathit{list} \mid u_1 \cdots u_n = u\}$$

Then q^{*p} is a canonizer in $S^* \stackrel{\Box}{\longleftrightarrow} U^*/\mathsf{TrClose}({\sim_U}^*)$.

Note that U need not be unambiguously iterable. The p function takes a string belonging to U^* and splits it into a list of strings belonging to U.

Union The final regular operator forms the union of two canonizers.

$$\begin{array}{c} S_1 \cap S_2 = \emptyset \\ q_1 \in S_1 \leftrightarrow\!\!\!\!\!\mapsto U_1/{\sim_{U_1}} \\ q_2 \in S_2 \leftrightarrow\!\!\!\!\!\mapsto U_2/{\sim_{U_2}} \\ \hline q_1 \,|\, q_2 \in (S_1 \cup S_2) \leftrightarrow\!\!\!\!\!\mapsto (U_1 \cup U_2)/\mathsf{TrClose}({\sim_{U_1}} \cup {\sim_{U_2}}) \end{array}$$

$$\begin{array}{ll} \textit{canonize } s = \begin{cases} q_1.\textit{canonize } s & \text{if } s \in S_1 \\ q_2.\textit{canonize } s & \text{otherwise} \end{cases} \\[2ex] \textit{choose } u \quad = \begin{cases} q_1.\textit{choose } u & \text{if } u \in U_1 \\ q_2.\textit{choose } u & \text{otherwise} \end{cases} \end{array}$$

3.2.17 Lemma: Let $q_1 \in S_1 \leftrightarrow\!\!\!\!\!\mapsto U_1/{\sim_{U_1}}$ and $q_2 \in S_2 \leftrightarrow\!\!\!\!\!\mapsto U_2/{\sim_{U_2}}$ be canonizers such that the intersection $S_1 \cap S_2$ of the source types is empty. Then $q_1 \,|\, q_2$ is a canonizer in $S_1 \cup S_2 \leftrightarrow\!\!\!\!\!\mapsto (U_1 \cup U_2)/$ $\mathsf{TrClose}({\sim_{U_1}} \cup {\sim_{U_2}})$.

The typing rule closes ${\sim_{U_1}} \cup {\sim_{U_2}}$ transitively to ensure that it is an equivalence.

Other Primitives

So far, we have focused on quotient lenses and canonizers that can be constructed from existing basic lenses using the lifting and quotienting operators. Of course, we can also define new primitives directly—the semantic laws governing the behavior of quotient lenses and canonizers allow many transformations that are not valid as basic lenses. This section defines some quotient lenses and canonizers that cannot be obtained by lifting and quotienting.

Duplication In many applications, it is useful to duplicate part of the source in the view. For example, consider augmenting a document with a table of contents generated from section headings. Unfortunately, as discussed at the end of Chapter 2, it is impossible to have a well-behaved basic lens that duplicates the source at the type $S \Longleftrightarrow (S{\cdot}S)$ because if we edit just one copy of the duplicated data the PutGet law will not be satisfied—no matter what the *put* function does, *get* will return a view where the two copies are equal. We can have a duplication operator as a basic lens if we assign it the more complicated type $S \Longleftrightarrow \{s{\cdot}s' \in S{\cdot}S \mid s = s'\}$, but the types used in our implementation are based on regular languages, which cannot express arbitrary equality constraints. Fortunately, it is easy to define a duplication operator with a regular type as a quotient lens:

$$\begin{array}{c} V_1{\cdot^!}V_2 \\ l \in S/{\sim_S} \Longleftrightarrow V_1/{\sim_{V_1}} \\ f \in S \to V_2 \\ \hline \textit{dup}_1\ l\ f \in S/{\sim_S} \Longleftrightarrow (V_1{\cdot}V_2)/({\sim_{V_1}}{\cdot}\mathsf{Tot}(V_2)) \end{array}$$

$$\begin{array}{ll} \textit{get } s & = (l.\textit{get } s){\cdot}(f\ s) \\ \textit{put } (v_1{\cdot}v_2)\ s & = (l.\textit{put } v_1\ s) \\ \textit{create } (v_1{\cdot}v_2) & = (l.\textit{create } v_1) \end{array}$$

3.2.18 Lemma: Let l be a quotient lens in $S/{\sim_S} \Longleftrightarrow V_1/{\sim_{V_1}}$ and let f be a function from S to V_2 such that $V_1{\cdot^!}V_2$. Then $\textit{dup}_1\ l\ f$ is a quotient lens in $S/{\sim_S} \Longleftrightarrow (V_1{\cdot}V_2)/({\sim_{V_1}}{\cdot}\mathsf{Tot}(V_2))$.

The dup_1 lens takes a quotient lens l and a function f as arguments (f is often the *get* component of a lens). In the *get* direction it duplicates the source string and passes one copy to l's *get* function and the other copy to f. The *put* function discards the portion of the view generated by f and invokes l's *put* function on the rest of the view. For example, the following lens duplicates a letter in the *get* direction:

```
let l : lens = copy [A-Z]
test (dup1 l (get l)).get "A" = "AA"
```

The *put* function simply discards the second copy:

```
test (dup1 l (get l)).put "BC" into "A" = "B"
```

The type of dup_1 records the fact that it treats all strings in the part of the view generated by f as equivalent.

A symmetric operator $dup_2\ f\ l$ discards the first substring of the view in the *put* direction instead. In both of these lenses, the handling of duplicated data is admittedly quite simple. In particular, unlike the duplication operators proposed and extensively studied by [HMT08], *put* and *create* do not merge the changes made to the duplicated data. Nevertheless, they suffice for many examples that arise in practice. For example, when f is an aggregation operator such as *count* E, which takes a string u in $[\![E]\!]^*$ and returns the number of substrings in E that u can be split into, discarding the aggregate value while propagating the modifications made to the other copy makes sense.

Normalize The next combinator is a generic operator builds a canonizer from a function that maps a set of strings onto a "normalized" subset of itself.

$$\frac{S_0 \subseteq S \qquad \forall s \in S_0.\ f\ s = s \qquad f \in S \to S_0}{\textit{normalize}\ f \in S \mathrel{\leftrightarrow\!\!\!\!\!\!\!\square} S_0/{=}}$$

$$\begin{aligned} \textit{canonize}\ s &= f\ s \\ \textit{choose}\ s &= s \end{aligned}$$

3.2.19 Lemma: Let S and S_0 be sets such that $S_0 \subseteq S$. Also let $f \in S \to S_0$ be a function from S to S_0 such that $f\ s = s$ for every s in S_0. Then *normalize* f is a canonizer in $S \mathrel{\leftrightarrow\!\!\!\!\!\!\!\square} S_0/{=}$.

The *canonize* function is the function f, and the *choose* component is the identity function on S_0. The constraint on f and the condition that $S_0 \subseteq S$ guarantee that the ReCanonize law holds.

As an example, consider a canonizer that puts the substrings of a longer string in sorted order. To keep the notation simple, we will describe the binary version; generalizing to an n-ary sorting canonizer is straightforward. Let S_1 and S_2 be regular languages such that $S_1 \cdot^{!} S_2$ and $S_2 \cdot^{!} S_1$. Let f be the function from $(S_1 \cdot S_2) \cup (S_2 \cdot S_1)$ to $S_1 \cdot S_2$ that takes the concatenation—in either order—of a string belonging to S_1 and a string belonging to S_2, and reorders the substrings so that the S_1 substring comes before the S_2 substring. Formally, f is defined by the following equations: $f\ (s_1 \cdot s_2) = (s_1 \cdot s_2)$ and $f\ (s_2 \cdot s_1) = (s_1 \cdot s_2)$, where $s_1 \in S_1$ and $s_2 \in S_2$. Observe that f satisfies the side condition mentioned in the typing rule for *normalize* because it behaves like the identity function on already-sorted strings. Thus, the canonizer *sort* $S_1\ S_2 \triangleq$ *normalize* f is in $(S_1 \cdot S_2 \cup S_2 \cdot S_1) \mathrel{\leftrightarrow\!\!\!\!\!\!\!\square} (S_1 \cdot S_2)/{=}$. We often use the n-ary version of *sort* to standardize the representation of semantically-unordered data—e.g., XML attributes, BibTeX fields, etc.

Unwrap Many ad hoc textual formats require that long lines be broken by replacing a space character with a newline followed by several spaces. For example, the UniProt genomic database (described in Section 3.5) does not allow lines longer than 75 characters. The *unwrap* canonizer maps between wrapped and unwrapped lines of text:

$$\frac{n \in \mathbb{N} \qquad \mathit{sp} \in \Sigma^* \qquad \mathit{nl} \in \Sigma^* \\ (\Sigma^* {\cdot} \mathit{nl} {\cdot} \Sigma^*) \cap S_0 = \emptyset}{\mathit{unwrap}\ n\ S_0\ \mathit{sp}\ \mathit{nl} \in ([(\mathit{sp} \cup \mathit{nl})/\mathit{sp}]S_0) \mathrel{\leftarrow\!\!\!+\!\!\!\twoheadrightarrow} S_0/{=}}$$

canonize s : *replace* nl *with* sp *in* s
choose s : *replace* sp *with* nl *in* s *as needed to break lines longer than* n

3.2.20 Lemma: Let n be a natural number, S_0 a language, and *sp* and *nl* strings such that for every string u in S_0 the string *nl* does not occur in u. Then *unwrap* n S_0 *sp* *nl* is a canonizer in $([(\mathit{sp} \cup \mathit{nl})/\mathit{sp}]S_0) \mathrel{\leftarrow\!\!\!+\!\!\!\twoheadrightarrow} S_0/{=}$.

Formally, the *unwrap* canonizer takes a number n, a set of strings S_0, a "space" string *sp*, and a "newline" string *nl* as arguments. The *canonize* function unwraps blocks of text by replacing every occurrence of the newline string with the space string. The *choose* function attempts to create well-wrapped lines of text. It breaks lines longer than n characters by replacing the space string with the newline string as needed. The typing rule for *unwrap* stipulates that *nl* must not appear in any strings in S_0. The type of uncanonized strings is obtained by widening S_0 so that *nl* may appear anywhere that *sp* may.

3.3 Loosening Lens Types

We were originally motivated to study quotient lenses by the need to work "modulo insignificant details" when writing lenses to transform real-world data formats. However, as we began using quotient lenses to develop larger examples we discovered a significant—and completely unexpected—side benefit: quotient lenses allow us to assign many lenses *coarser* types than the strict lens laws allow, which eases some serious tractability concerns.

Recall from Chapter 2 that because we require *put* to be a total function, the types of many lenses need to be extremely precise. Totality is attractive for users because it ensures that any view can be put back with any source. However, for exactly the same reason, totality makes it more difficult to design lens primitives—the *put* function must do something reasonable with every valid view and valid source, so the only way that a lens can avoid having to handle certain structures is by excluding them from its type. Thus, in practice, a lens language with a sufficiently rich collection of primitives needs to be equipped with a correspondingly rich collection of types. There are some advantages to working with very precise types—e.g., the Boomerang typechecker often finds subtle source of ambiguity. But it also imposes burdens because programmers must write programs that satisfy a very picky typechecker and implementations must use algorithms that are computationally expensive. Fortunately, the increased flexibility of quotient lenses and canonizers can be exploited to loosen types and alleviate both of these burdens. This section describes three examples of this phenomenon.

The first example involves the *unwrap* transformation. The functions that map between unbroken lines of text and blocks of well-wrapped lines are a bijection, so they satisfy the basic lens laws trivially. Thus, we could define *unwrap* as a lens—either as a primitive, or using combinators (although the combinator program would have to keep track of the number of characters on the current line so it would be quite tedious to write). However, the type of this lens on the view side would be the set of minimally-split, well-wrapped blocks of text (i.e., sequences of lines that must be broken exactly at the margin column, or ones that must be broken at the column just before the margin because the next two characters are not spaces, or lines that must be broken at the second-to-last column... and so on). This type is complicated and cumbersome—both for programmers who must use it and for the implementation, which must represent it. We could loosen the type to match the one we assigned to the *unwrap* canonizer—i.e., arbitrary blocks of text, including ones with "extra" newlines—but changing the type in this way also requires changing the *put* function in order to avoid violating the GetPut law.[5] Formulating *unwrap* as a canonizer rather than a lens, avoids all of these issues and results in a primitive whose type and behavior are both simple.

The second example of a transformation whose type can be simplified using canonizers is *sort*. As with *unwrap*, it is possible to define a basic lens version of *sort*. To sort S_1 to S_k, we take the union of the lenses that recognize every permutation of the S_is and use the *permute* lens to put them in sorted order. This lens has the behavior we want, but its type on the source side is the set of all permutations of the S_is—a type whose size grows as the factorial of k! Representing this type in the implementation would rapidly become impractical. Fortunately, this combinatorial blowup can be avoided by widening the concrete type to $(S_1 \mid \ldots \mid S_k)^*$. This approximates the set of strings that we actually want to sort, but has an *enormously* more compact representation—one that grows linearly with k. Of course, having widened the type in this way, we also need to extend the canonizer's functional components to handle this larger set of strings. In particular, we must extend *canonize* to handle the case where several or no substrings belong to a given R_i. A simple choice that works well for many examples is to discard the extras and fill in any missing ones with defaults.

The final example involves the duplication operator. As discussed in Chapter 2, it is possible to have duplication as a basic lens the view type must include an equality constraint to satisfy PutGet. By defining dup_1 and dup_2 as quotient lenses, we obtain a primitive with a much simpler—in fact, regular—type.

3.4 Typechecking

The examples in the previous section showed how quotient lenses ease certain aspects of typechecking. However, quotient lenses complicate other aspects because the typechecker needs to keep track of equivalence relations on the source and view. Also, the typing rules for left and right quotienting, sequential composition, and union all place constraints on the equivalence relations mentioned in the types of sublenses. For example, to check that the composition $l;k$ is well typed, it needs to verify that the equivalence on l's view is identical to the equivalence on k's source.

This section describes two different strategies for implementing these rules. The first uses a coarse analysis, simply classifying equivalences according to whether they are or are not the equality relation. Surprisingly, this very simple analysis captures our most common programming idioms and turns out to be sufficient for all of the applications we have built. The second approach is more refined: it represents equivalence relations by rational functions that induce them. This works, in principle, for a large class of equivalence relations including most of our canonizers (except for those that do reordering). Unfortunately, it requires representing and deciding equivalences for finite state transducers, which appears prohibitively expensive.

[5] If we take a block of text containing extra newlines, map it to a single line of text by *get*, and immediately map it back to a block by *put*, then the GetPut law stipulates that the extra newlines must be restored exactly. Thus, the *put* function cannot just insert the minimum number of newlines needed to avoid spilling over into the margin; it must restore the extra newlines from the original source.

The first type system is based on two simple observations: first, most quotient lenses originate as lifted basic lenses, and therefore have types whose equivalence relations are both equality; second, equality is preserved by many of our combinators including all of the regular operators, *swap*, sequential composition, and even (on the non-quotiented side) the left and right quotient operators. These observations suggest a coarse classification of equivalence relations into two sorts:

$$\tau \ ::= \ \mathit{Identity} \ \mid \ \mathit{Any}$$

We can now restrict the typing rules for our combinators to only allow sequential composition, quotienting, and union of types whose equivalence relation type is *Identity*. Although this restriction is draconian (it disallows many quotient lenses that are valid according to the typing rules presented in earlier sections), it turns out to be surprisingly successful in practice—we have not needed anything more in many thousands of lines of lens programs. There are two reasons for this. First, it allows two quotient lenses to be composed, whenever the uses of *lquot* are all in the lens on the left and the uses of *rquot* on the right, a very common case. And second, it allows arbitrary quotient lenses (with any equivalences) to be concatenated as long as the result is not further composed, quotiented, or unioned—another very natural idiom. This is the typechecking algorithm implemented in our Boomerang language.

In theory, we can go further by replacing the *Identity* sort with a tag carrying an arbitrary rational function f—i.e., a function computable by a finite state transducer[Ber79]:

$$\tau \ ::= \ \mathit{Fst} \text{ of } f \ \mid \ \mathit{Any}$$

Equivalence relations induced by rational functions are a large class that includes nearly all of the equivalence relations that can be formed using our combinators—everything except quotient lenses constructed from canonizers based on *sort* and *swap*. Moreover, we can decide equivalence for these relations.

3.4.1 Notation [Induced Equivalence Relation]: Let $f \in A \to B$ be a rational function. Denote by $\sim_f$ the relation $\{(x, y) \in A \times A \mid f(x) = f(y)\}$.

3.4.2 Lemma: Let $f \in A \to B$ and $g \in A \to C$ be rational and surjective functions. Define a rational relation $h \subseteq C \times B$ as $f \circ g^{-1}$. Then $\sim_g \subseteq \sim_f$ if and only if h is functional.

Proof: Let us expand the definition of h

$$h(c) = \{f(a) \mid a \in A \text{ and } g(a) = c\}$$

Observe that, by the surjectivity of g we have $h(c) \neq \emptyset$.

($\Rightarrow$) Suppose that $\sim_g \subseteq \sim_f$.

Let $b, b' \in h(c)$. Then by the definition of h, there exist $a, a' \in A$ with $b = f(a)$ and $b' = f(a')$ and $g(a) = c = g(a')$. We have that $a \sim_g a'$, which implies that $a \sim_f a'$, and so $b = b'$. Since b and b' were arbitrary elements of $h(c)$, we conclude that h is functional.

($\Leftarrow$) Suppose that h is functional.

Let $a, a' \in A$ with $a \sim_g a'$. Then there exists $c \in C$ such that $g(a) = g(a') = c$. By the definition of h, and our assumption that h is functional, we have that $f(a) = h(c) = f(a')$ and so $a \sim_f a'$. Since a and a' were arbitrary, we conclude that $\sim_g \subseteq \sim_f$. □

3.4.3 Corollary: Let f and g be rational functions. It is decidable whether $\sim_f = \sim_g$.

Proof: Recall that rational relations are closed under composition and inverse. Observe that $\sim_f = \sim_g$ if and only if both $f \circ g^{-1}$ and $g \circ f^{-1}$ are functional. Since these are both rational relations, the result follows using the decidability of functionality for rational relations [Bla77]. □

The condition mentioned in union can also be decided using an elementary construction on rational functions. Thus, this finer system gives decidable typechecking for a much larger set of quotient lenses. Unfortunately, the constructions involved seem prohibitively expensive to implement.

3.5 Examples

Most of the examples discussed in this chapter have focused on fairly simple transformations—e.g., handling whitespace. In this last section, we illustrate the use of quotient lenses in a larger transformation that maps between XML and ASCII versions of the UniProtKB/Swiss-Prot protein sequence database. We originally implemented this transformation as a basic lens, but found that although the lens handled the essential data correctly, it did not handle the full complexity of either format. On the XML side, the databases had to be a certain canonical form—e.g., with attributes in a particular order—while on the ASCII side, it did not conform to the UniProt conventions for wrapping long lines and did not handle fields with duplicated data. We initially considered implementing custom viewers to handle these complexities, but this turned out to be almost as difficult as writing the lens itself, due to the slightly different formatting details used to represent lines for various kinds of data. Re-engineering the program as a quotient lens was a big improvement.

To get a taste of programming with quotient lenses, let us start with a simple example illustrating canonization of XML trees. In the XML presentation of UniProt databases, patent citations are represented as XML elements with three attributes:

```
<citation type="patent" date="1990-09-20"
          number="WO9010703"/>
```

In ASCII, they appear as `RL` lines:

```
RL  Patent number WO9010703, 20-SEP-1990.
```

The bidirectional transformation between these formats is essentially bijective—the patent number can be copied verbatim to the line, and the date can be transformed from `YYYY-MM-DD` to `DD-MMM-YYYY`—but because the formatting of the element may include extra whitespace and the attributes may appear in any order, building a lens that maps between all valid representations of patent citations in XML and ASCII formats is more complicated than it might seem at first.

A bad choice (and the only choice available with basic lenses) would be to treat the whitespace and the order of attributes as data that should be explicitly discarded by the *get* function and restored by the *put*. This complicates the lens, which then has to explicitly manage all this irrelevant data. Slightly better would be to write a canonizer that standardizes the representation of the XML tree and compose this with a lens that operates on the canonized data to produce the ASCII form. But we can do even better by combining the functionality of the canonizer and the lens into a single quotient lens. It uses some helper functions and library code described below.

```
let patent_xml : lens =
  ins "RL  " .
  Xml.attr3_elt_no_kids NL2 "citation"
    "type" ("patent" <-> "Patent number" . space)
```

```
        "number" (escaped_pcdata . comma . space)
        "date" date .
      dot
```

This lens transforms concrete XML to abstract ASCII in a single pass. The first line inserts the RL tag and spaces into the ASCII format. The second line is a library function from the Xml module that encapsulates details related to the processing of XML elements. The first argument, a string NL2, is a constant representing the second level of indentation. The attr3_elt_no_kids function passes this argument to the *qdel* lens, which uses it to construct the whitespace before the open tag for the XML element in the *put* direction. The second argument, citation, is the name of the element. The remaining arguments are the names of the attributes and the lenses used for processing their corresponding values. These are given in canonical order. Internally attr3_elt_no_kids sorts the attributes to put them in this order. The space, comma, and dot lenses insert the obvious characters; escaped_pcdata handles unescaping of PCDATA; the date lens performs the bijective transformation on dates illustrated above.

The next example illustrates quotienting on the ASCII side. In the XML format, taxonomic lineages of source organisms are represented like this

```
<lineage>
  <taxon>Eukaryota</taxon>
  <taxon>Lobosea</taxon>
  <taxon>Euamoebida</taxon>
  <taxon>Amoebidae</taxon>
  <taxon>Amoeba</taxon>
</lineage>
```

while in ASCII, they are flattened onto lines tagged with OC:

```
OC  Eukaryota; Lobosea; Euamoebida; Amoebidae; Amoeba.
```

The code that converts between these formats is:

```
let oc_taxon : lens =
  Xml.pcdata_elt NL3 "taxon" esc_pcdata
let oc_xml : lens =
  ins "OC  " .
  Xml.elt NL2 "lineage"
    (iter_with_sep oc_taxon (semi . space)) .
  dot
```

The first lens, oc_taxon, processes a single taxon element using a library function pcdata_elt that extracts encapsulated PCDATA from an element. As in the previous example, the NL3 argument is a constant representing canonical whitespace. The second lens, oc_xml, processes a lineage element. It inserts the OC tag into the ASCII line and then processes the children of the lineage element using a generic library function iter_with_sep that iterates its first argument using Kleene-star, and inserts its second argument between iterations. The dot lens terminates the line. The lineage for amoeba shown above is compact enough to fit onto a single OC line, but most lineages are not:

```
OC  Eukaryota; Metazoa; Chordata; Craniata; Vertebrata;
OC  Euteleostomi; Mammalia; Eutheria; Euarchontoglires;
OC  Primates; Haplorrhini; Catarrhini; Hominidae; Homo.
```

The quotient lens that maps between single-line `OC` strings produced by `oc_xml` and the final line-wrapped format:[6]

```
let oc_q : canonizer =
  unwrap 60 (vtype oc_xml) " " "\nOC  "
let oc_line : lens =
  rquot oc_xml oc_q
```

The `vtype` primitive extracts the view part of the type of a quotient lens.

Lastly, let us look at two instances where data is duplicated. In a few places in the UniProt database, there is data that is represented just once on the XML side but several times on the ASCII side. For example, the count of the number of amino acids in the actual protein sequence for an entry is listed as an attribute in XML

```
<sequence length="262" ...>
```

but appears twice in ASCII, in the `ID` line

```
ID  GRAA_HUMAN Reviewed; 262 AA.
```

and again in the `SQ` line:

```
SQ  SEQUENCE   262 AA;  28969 MW;
```

Using dup_2, we can write a lens that copies the data from the XML attribute and onto both lines in the ASCII format. The backwards direction of dup_2 discards the copy on the `ID` line, a reasonable policy for this application, since it is generated information.

Another place where duplication is needed is when data is aggregated. The ASCII format of the information about alternative splicings of the gene is

```
CC  -!- ALTERNATIVE PRODUCTS:
CC      Event=Alternative initiation; Named isoforms=2;
CC      Name=Long; Synonyms=Cell surface;
CC        IsoId=P08037-1; Sequence=Displayed;
CC      Name=Short; Synonyms=Golgi complex;
CC        IsoId=P08037-2; Sequence=VSP_018801;
```

where the `Named isoforms` field in the second line is the count of the number of `Name` blocks that follow below. The quotient lens that produces these lines uses dup_2 and *count* to generate the appropriate integer in the *get* direction and simply discards the integer in the *put* direction.

3.6 Summary

Quotient lenses generalize basic lenses, allowing their forward and backward transformations to treat certain specified portions of the source and view as "inessential". This extension, while simple at the semantic level, turns out have an elegant syntactic story based on canonizers and quotienting operators. The resulting system is parsimonious—the same primitives can be used as lenses and as canonizers—and compositional—unlike previous approaches, where canonization is kept at the edges of transformations, canonizers can be freely interleaved with the processing of data. Moreover, the flexibility offered

[6] In this example, we have split lines at the 60th column to make them fit onto the page. In a real UniProt instance, the lines would be split at the 75th column instead.

by quotient lenses make it possible to define additional primitives such as duplication and sorting operators and simplifies the typing rules for several operators, which addresses some serious engineering concerns. Our experience indicates that canonizers and quotient lenses are essential for building lenses for real-world data formats.

Chapter 4

Resourceful Lenses

> *"The art of progress is to preserve order amid change*
> *and to preserve change amid order."*
> —Alfred North Whitehead

Alignment is a fundamental issue in bidirectional languages. Intuitively, to correctly propagate an update to a view, the *put* component of a lens needs to be able to match up the pieces of the modified view with the corresponding pieces of the underlying source. Unfortunately, the basic and quotient lenses we have seen so far have extremely limited capabilities with respect to alignment—they match up data by position. When the update to the view preserves the positional association between pieces of the source and pieces of the view this simple strategy works well, but when the update breaks the association it does not—the *put* function restores information extracted from pieces of the source to completely unrelated pieces of the view.

To illustrate the problem, consider the composers example again. Suppose that the source is a string representing the same XML document as before

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <dates>1865-1956</dates>
    <nationality>Finnish</nationality>
  </composer>
  <composer>
    <name>Aaron Copland</name>
    <dates>1910-1990</dates>
    <nationality>American</nationality>
  </composer>
  <composer>
    <name>Benjamin Briten</name>
    <dates>1913-1976</dates>
    <nationality>English</nationality>
  </composer>
</composers>
```

and the view is the corresponding block of ASCII text:

```
Jean Sibelius, 1865-1956
```

```
Aaron Copland, 1910-1990
Benjamin Briten, 1913-1976
```

If the update only modifies composers in place and perhaps adds new entries to the end of the view, the simple positional alignment strategy used in basic lenses works well. For example, if we change Sibelius's death date from "`1956`" to "`1957`", correct the spelling of Britten's name from "`Briten`" to "`Britten`", and add a new line for Tansman, then the *put* function combines

```
Jean Sibelius, 1865-1957
Aaron Copland, 1910-1990
Benjamin Britten, 1913-1976
Alexandre Tansman, 1897-1986
```

with the original source and yields an updated XML tree that reflects the changes made to the view:

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <dates>1865-1957</dates>
    <nationality>Finnish</nationality>
  </composer>
  <composer>
    <name>Aaron Copland</name>
    <dates>1910-1990</dates>
    <nationality>American</nationality>
  </composer>
  <composer>
    <name>Benjamin Britten</name>
    <dates>1913-1976</dates>
    <nationality>English</nationality>
  </composer>
  <composer>
    <name>Alexandre Tansman</name>
    <dates>1897-1986</dates>
    <nationality>Unknown</nationality>
  </composer>
</composers>
```

However, if the update breaks the positional association between pieces of the source and view, the behavior of the *put* function is highly unsatisfactory. For example, if we correct Sibelius's death date and Britten's name as above and *swap* the order of Britten and Copland, then the *put* function combines

```
Jean Sibelius, 1865-1957
Benjamin Britten, 1913-1976
Aaron Copland, 1910-1990
```

with the original source and produces a mangled source

```
<composers>
  <composer>
```

```
    <name>Jean Sibelius</name>
    <dates>1865-1957</dates>
    <nationality>Finnish</nationality>
  </composer>
  <composer>
    <name>Benjamin Britten</name>
    <dates>1913-1976</dates>
    <nationality>American</nationality>
  </composer>
  <composer>
    <name>Aaron Copland</name>
    <dates>1910-1990</dates>
    <nationality>English</nationality>
  </composer>
</composers>
```

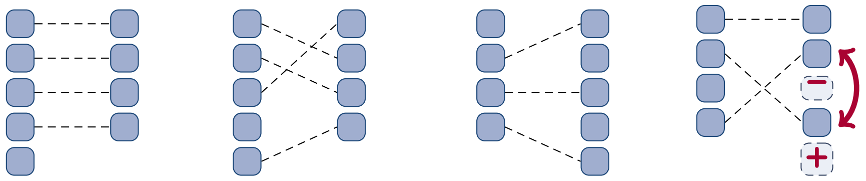
where the nationality has been taken from Copland's element and spliced into Britten's element, and vice versa.

This is a serious problem, and a pervasive one: it is triggered whenever the *get* function discards information and the update to the view does not preserve the positional correspondence between pieces of the source and view. It is a show-stopper for many of the applications we want to write using lenses. Of course, what we would like is for the *put* function to align the composers in the source and view using some criteria other than their absolute position. For example, it could match up composers with the same name. On the same inputs as above, a lens that aligned composers in this way would produce a source in which each nationality is restored to the appropriate composer:

```
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <dates>1865-1957</dates>
    <nationality>Finnish</nationality>
  </composer>
  <composer>
    <name>Benjamin Britten</name>
    <dates>1913-1976</dates>
    <nationality>English</nationality>
  </composer>
  <composer>
    <name>Aaron Copland</name>
    <dates>1910-1990</dates>
    <nationality>American</nationality>
  </composer>
</composers>
```

Unfortunately, neither basic lenses nor quotient lenses provide the means to achieve this effect. Developing mechanisms that lens programmers can use to specify and use strategies for aligning information in the source and view is the goal of this chapter.

Our solution is to provide programmers with a way to identify the locations of *chunks* in the source and view as well as a way to specify the strategy to use to align chunks in the *put* direction. The



(a) positional (b) min. edit distance (c) non-crossing (d) operation-based

Figure 4.1: Alignment strategies

idea is for the *put* function to use the specified policy to compute an association—formally, a partial injection—between chunks in the original view and the updated view. It then combines this association with the association between chunks in the source and view realized by the *get* function to obtain an end-to-end association between chunks in the original source with chunks in the updated view and uses this *alignment* to *put* back corresponding chunks with each other.

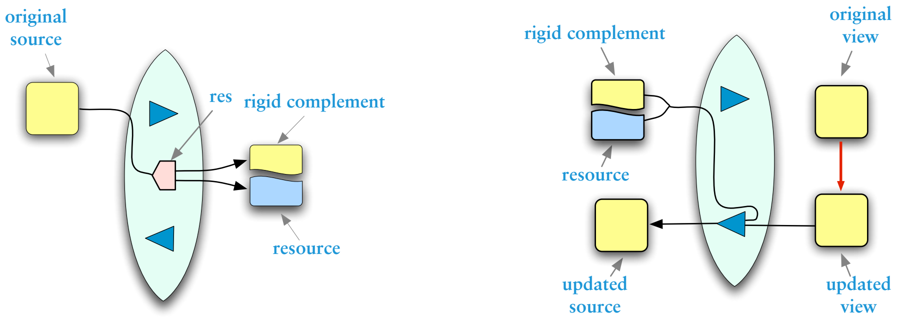
Here is a lens written using the extensions described in this chapter that has the desired behavior for the composers example:

```
let composer : lens =
  xml_elt "composer"
    ( xml_elt "name"
        (key (copy (ALPHA . " " . ALPHA) ) )
    . ins ", "
    . xml_elt "dates" (copy (YEAR . "-" . YEAR) )
    . xml_elt "nationality"
        (del_default ALPHA "Unknown" ) )

let composers : lens =
  xml_elt "composers"
    ( copy "" | <composer> . (ins "\n" . <composer>)* )
```

Compared to the original version of the lens we have made two changes. First, we have enclosed both occurrences of the `composer` lens in angle brackets, indicating that each composer should be treated as a reorderable chunk. Second, we have wrapped the lens that copies the name of each composer to view with a special primitive `key`. This indicates that the lens should use the name to align the enclosing chunk. We do not actually demand that keys be unique—i.e., these are not keys in the strict database sense so when several chunks have the same key the relative alignment of those chunks is positional. The net effect of these changes is that the *put* function aligns composers by name, as desired.

To make these features work as expected, we extend the framework of basic lenses in several ways. Operationally, we decouple the handling of rigidly ordered and reorderable information by dividing the representation of source information provided to the *put* function into two pieces: a *rigid complement* that records the source information that should be handled positionally and a *resource* that stores the information extracted from the reorderable chunks in the source. The resource provides a means of supplying a lens with explicit alignment information since its elements can be rearranged to yield a pre-aligned structure in which each element matches up with the appropriate chunk in the view. To generate these structures, we augment lenses with a new component called *res*. The architecture of these *resourceful lenses* is depicted graphically in Figure 4.2.



(a) *res* function (b) *put* function

Figure 4.2: Resourceful Lens Architecture

Semantically, we enrich the types of resourceful lenses with explicit notions of what constitutes a reorderable "chunk," and we add new behavioral laws that capture essential constraints on the handling of chunks. These laws stipulate that lenses must carry chunks in the source through to chunks in the view, and vice versa. They can be used to derive other natural properties—e.g., that resourceful lenses translate reorderings to reorderings.

Finally, at the level of syntax, we develop a collection of primitives for building lenses for strings. We define coercions that convert between basic lenses and resourceful lenses, reinterpret the familiar regular operators (union, concatenation, and Kleene star) as resourceful lenses, and show that resourceful lenses are closed under composition. We also present primitives for specifying, combining, and tuning alignment functions in terms of "species", "tags", "keys", and "thresholds."

One of the hallmarks of our design for resourceful lens is its flexibility. In a resourceful lens, the handling of rigidly ordered and reorderable information are completely decoupled from each other. This explicit separation of concerns provides a clean interface for supplying a lens with clear directives about how the source and view should be aligned and yields an extremely flexible framework that can be instantiated with many different policies for computing alignments. Figure 4.1 depicts several common strategies we have found useful in examples: (a) positional, (b) minimizing the total edit distance between matched chunks, (c) minimizing total edit distance but only considering "non-crossing" alignments (i.e., similar to longest common sequence), and (d) extracting an alignment from the actual update operation applied to the view. The framework of resourceful lenses accommodates each of these strategies.

The contributions of this chapter can be summarized as follows:

1. We define a refined semantic space of *resourceful lenses* that enriches the types of basic lenses with chunks and adds new behavioral laws ensuring that chunks are handled correctly.

2. We develop syntax for constructing resourceful lenses for strings, showing how to convert between basic and resourceful lenses, and reinterpreting each of the regular operators as a resourceful lens.

3. We describe a number of alignment "species" and show how they can be tuned using notions of "tags", "keys", and "thresholds."

In outline, the rest of this chapter is organized as follows. Section 4.1 defines the semantic space of resourceful lenses and develops some essential properties. Section 4.2 defines a core language of resourceful lens primitives for strings. Section 4.3 presents primitives for describing and tuning alignment strategies. Section 4.4 describes extensions to the framework. We conclude the chapter in Section 4.5.

4.1 Semantics

Resourceful lenses are organized around a simple two-level architecture in which a top-level lens handles the alignment of chunks and the processing of information outside of chunks and a lower-level basic lens handles the processing of information contained in chunks. For now, we will assume that chunks only appear at the top level and the same basic lens is used to process every chunk. We will see how to generalize this design to accommodate multiple basic lenses and nested chunks in Sections 4.3 and 4.4.

Notation

Before we can define resourceful lenses precisely, we need to fix a few pieces of notation. We assume that the sets of sources and views come equipped with notions of what constitutes a reorderable chunk of information. When u is a structure containing chunks we write

- $|u|$ for the number of chunks in u,
- $\mathit{locs}(u)$ for the set of locations of chunks in u, where a location is a natural number and we number the chunks of u from 1 to $|u|$ in some canonical way,
- $u[n]$ for the chunk located at n in u, where $n \in \mathit{locs}(u)$,
- $u[n{:=}v]$ for the structure obtained from u by setting the chunk at n to v, where n in $\mathit{locs}(u)$ and v is a structure,
- and $\mathit{skel}(u)$ for the residual structure consisting of the parts of u that are not contained in any chunk.

We assume that structures are completely characterized by their skeleton and chunks.

To ensure that chunks can be freely reordered, we will require the sets of sources and views be closed under the operation of replacing chunks by other chunks. Formally, when U is a set of structures with chunks (e.g., the set of sources or views) and U' is a set of ordinary structures, we say that U is *chunk compatible* with U' if and only if

- the chunks of every structure in U belong to U'—i.e., for every $u \in U$ we have $u[n] \in U'$,
- and membership in U is preserved when we replace arbitrary chunks with elements of U'—i.e., for every $u \in U$ and $n \in \mathit{locs}(u)$ and $u' \in U'$ we have that $u[n{:=}u'] \in U'$.

To represent resources we will use finite maps from locations to chunks. This representation makes it easy to apply alignment information to resources—we can match up chunks in the source and view as specified in the alignment by permuting the order of certain chunks and discarding others. When r is a finite map we write:

- $\{||\}$ for the totally undefined finite map,
- $\{|n \mapsto c|\}$ for the singleton finite map that associates the location n to the complement c and is otherwise undefined,
- $r(n)$ for the complement that r associates to n,

- $\mathrm{dom}(r)$ for the domain of r,
- $|r|$ for the largest element of $\mathrm{dom}(r)$,
- $(r_1 \mathbin{++} r_2)$ for the finite map that behaves like the finite map r_1 on locations in $\mathrm{dom}(r_1)$ and like r_2 on other locations,
$$(r_1 \mathbin{++} r_2)(n) \triangleq \begin{cases} r_1(n) & \text{if } n \leq |r_1| \\ r_2(n - |r_1|) & \text{otherwise,} \end{cases}$$
- and $\{|\mathbb{N} \mapsto C|\}$ for the set of all finite maps from locations to complements in a set C.

To illustrate how these finite maps are used in a resourceful lens, consider a simple abstract example. Suppose that we start with a source s that the *get* function maps to a view v and the *res* function maps to a rigid complement c and a finite map r, called a resource (recall that resourceful lenses split the representation of rigidly ordered and reorderable source information). Also suppose that the chunks in s, v and r are in exact correspondence—i.e., the lens does not reorder chunks, so for every location n, the source chunk $s[n]$ at n maps to $v[n]$ in the view and $r(n)$ in the resource. Now suppose that we change the view to v' and compute—in some way—a correspondence g between v' and v, represented formally as a partial injective function on the locations of chunks in v'. Composing g and r as functions yields a new resource in which the complement for each chunk in s is lined up with the specified chunk in v'. Thus, to propagate the modification make to the view we simply need to *put* back v' with the rigid complement c and the pre-aligned resource $r \circ g$ using a lens that accesses source information for chunks through the resource.

Definitions

We are now ready to define the semantic space of resourceful lenses precisely. To make the definition easier to follow, we first revise the definition of basic lenses to make the representation of source information provided to the *put* function—the *complement*—explicit.

4.1.1 Definition [Basic Lens with Complement]: Let S and V be sets of sources and views and let C be a set of "complements". A *basic lens* l mapping between S and V with complement C comprises four total functions

$$\begin{array}{rcl} l.\mathit{get} & \in & S \to V \\ l.\mathit{res} & \in & S \to C \\ l.\mathit{put} & \in & V \to C \to S \\ l.\mathit{create} & \in & V \to S \end{array}$$

obeying the following "round-tripping" laws for every source s in S, complement c in C, and view v in V:

$$l.\mathit{get}\ (l.\mathit{put}\ v\ c) = v \qquad (\textsc{PutGet})$$

$$l.\mathit{get}\ (l.\mathit{create}\ v) = v \qquad (\textsc{CreateGet})$$

$$l.\mathit{put}\ (l.\mathit{get}\ s)\ (l.\mathit{res}\ s) = s \qquad (\textsc{GetPut})$$

We write $S \overset{C}{\Longleftrightarrow} V$ for the set of all basic lenses between S, C, and V.

Now we are ready for the main definition:

4.1.2 Definition [Resourceful Lens]: Let S and V be sets of structures with chunks, C a set of rigid complements, and k a basic lens with S chunk compatible with $k.S$ (i.e., the source type of k) and V chunk compatible with $k.V$ (i.e., the view type of k). A *resourceful lens* l on S, C, k, and V comprises four functions

$$\begin{array}{rcl} l.\mathit{get} & \in & S \to V \\ l.\mathit{res} & \in & S \to C \times \{\!|\mathbb{N} \mapsto k.C|\!\} \\ l.\mathit{put} & \in & V \to C \times \{\!|\mathbb{N} \mapsto k.C|\!\} \to S \\ l.\mathit{create} & \in & V \to \{\!|\mathbb{N} \mapsto k.C|\!\} \to S \end{array}$$

obeying the following laws for every source $s \in S$, views $v \in V$ and $v' \in V$, rigid complement $c \in C$, and resource $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ (i.e., finite map from locations to appropriately-typed complements for k):

$$l.\mathit{get}\ (l.\mathit{put}\ v\ (c, r)) = v \qquad (\textsc{PutGet})$$

$$l.\mathit{get}\ (l.\mathit{create}\ v\ r) = v \qquad (\textsc{CreateGet})$$

$$l.\mathit{put}\ (l.\mathit{get}\ s)\ (l.\mathit{res}\ s) = s \qquad (\textsc{GetPut})$$

$$\mathit{locs}(s) = \mathit{locs}(l.\mathit{get}\ s) \qquad (\textsc{GetChunks})$$

$$\frac{c, r = l.\mathit{res}\ s}{\mathit{locs}(s) = \mathrm{dom}(r)} \qquad (\textsc{ResChunks})$$

$$\frac{n \in (\mathit{locs}(v) \cap \mathrm{dom}(r))}{(l.\mathit{put}\ v\ (c, r))[n] = k.\mathit{put}\ v[n]\ (r(n))} \qquad (\textsc{ChunkPut})$$

$$\frac{n \in (\mathit{locs}(v) \cap \mathrm{dom}(r))}{(l.\mathit{create}\ v\ r)[n] = k.\mathit{put}\ v[n]\ (r(n))} \qquad (\textsc{ChunkCreate})$$

$$\frac{n \in (\mathit{locs}(v) - \mathrm{dom}(r))}{(l.\mathit{put}\ v\ (c, r))[n] = k.\mathit{create}\ v[n]} \qquad (\textsc{NoChunkPut})$$

$$\frac{n \in (\mathit{locs}(v) - \mathrm{dom}(r))}{(l.\mathit{create}\ v\ r)[n] = k.\mathit{create}\ v[n]} \qquad (\textsc{NoChunkCreate})$$

$$\frac{\mathit{skel}(v) = \mathit{skel}(v')}{\mathit{skel}(l.\mathit{put}\ v\ (c, r)) = \mathit{skel}(l.\mathit{put}\ v'\ (c, r'))} \qquad (\textsc{SkelPut})$$

$$\frac{\mathit{skel}(v) = \mathit{skel}(v')}{\mathit{skel}(l.\mathit{create}\ v\ r) = \mathit{skel}(l.\mathit{create}\ v'\ r')} \qquad (\textsc{SkelCreate})$$

We write $S \overset{C,k}{\Longleftrightarrow} V$ for the set of all resourceful lenses on S, C, k and V.

Note that we build k, the basic lens that processes chunks, into the semantics of resourceful lenses because the ChunkPut, NoChunkPut, ChunkCreate, and NoChunkCreate laws all mention it. For technical reasons, it is important that the same basic lens be used for every chunk. Among other things, it ensures that a resourceful lens translates reorderings on the view to reorderings on the source.

The *get* function has the same type as in basic lenses. The *put* function, however, has a different type: it takes a rigid complement and a resource rather than a complement. The *res* extracts these structures from a source. The *create* function also has a different type—along with the view, it takes a resource as an argument. This makes it possible for resourceful lenses to restore source information to chunks that have been newly created. To create a source from a view "from scratch", we invoke *create* with the empty resource.

The PUTGET, CREATEGET, and GETPUT laws express the same constraints as the basic lens laws.

The GETCHUNKS and RESCHUNKS law constrain the handling of chunks. They force resourceful lenses to maintain a one-to-one correspondence between the chunks in the source and view and the complements in the resource. Specifically, the GETCHUNKS law stipulates that each chunk in the source must be carried through to a chunk in the view. This rules out lenses that advertise the presence of chunks in the source but not in the view and vice versa. The RESCHUNKS law requires an analogous property for the resource generated by the *res* function from the source. Lenses that violate these laws would cause problems with the protocol for using alignments information with a resourceful lens described previously—rearranging the resource using an alignment computed for the view would not make sense if the underlying source had different chunks than the view. We do not state PUTCHUNKS and CREATECHUNKS laws because they can be derived from the other laws:

4.1.3 Lemma [PutChunks]: Let l be a resourceful lens in $S \overset{C,k}{\Longleftrightarrow} V$, let $v \in V$ be a view, let $c \in C$ be a rigid complement, and let $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ be a resource. Then $locs(l.\mathit{put}\ v\ (c, r)) = locs(v)$.

4.1.4 Lemma [CreateChunks]: Let l be a resourceful lens in $S \overset{C,k}{\Longleftrightarrow} V$, let $v \in V$ be a view, and let $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ be a resource. Then $locs(l.\mathit{create}\ v\ r) = locs(v)$.

The next four laws are the fundamental resourceful lens laws—they ensure that the *put* and *create* functions use their resource arguments and the basic lens k correctly. The PUTCHUNKS law stipulates that the nth chunk in the source produced by *put* must be identical to the structure produced by applying k.*put* to the nth chunk in the view and the complement associated to n in the resource. The CHUNKCREATE law is similar. The NOCHUNKPUT and NOCHUNKCREATE laws stipulate that the resourceful lens must use k.*create* to produce the nth source when the resource does not contain an entry for n.

The last two laws, SKELPUT and SKELCREATE, state that the skeleton of the sources produced by *put* and *create* must not depend on any of the chunks in the view or complements in the resource. This law is critical for ensuring that resourceful lenses translate reorderings on chunks in the view to reorderings on source chunks.

Properties

Compared to the basic lens laws, these laws have a low-level and operational feel—they spell out the precise handling of chunks and resources in detail. However, we can use them to derive higher-level, more declarative properties. For instance, we can use them to show that the *put* and *create* components of every resourceful lens translate reorderings on the chunks in the view to corresponding reorderings on the chunks in the source. We write $\mathsf{Perms}(u)$ for the set of all permutations of the chunks in u and $(\textcircled{q}\ u)$ for the structure obtained by reordering the chunks of u according to a permutation q. The next two lemmas follow directly from the resourceful lens laws:

4.1.5 Lemma [ReorderPut]: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens, let $v \in V$ be a view, let $c \in C$ be a rigid complement, let $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ be a resource, and let $q \in \mathsf{Perms}(v)$ be a permutation on the chunks of v. Then we have $\textcircled{q}\ (l.\mathit{put}\ v\ (c, r)) = l.\mathit{put}\ (\textcircled{q}\ v)\ (c, r \circ q^{-1})$.

4.1.6 Lemma [ReorderCreate]: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens, let $v \in V$ be a view, let $r \in \{|\mathbb{N} \mapsto k.C|\}$ be a resource, and let $q \in \mathsf{Perms}(v)$ be a permutation on the chunks of v. Then we have $\textcircled{\scriptsize q}\ (l.\mathit{create}\ v\ r) = l.\mathit{create}\ (\textcircled{\scriptsize q}\ v)\ (r \circ q^{-1})$.

Another way to understand the semantics of resourceful lenses is by the coercion $\lfloor\cdot\rfloor$ (pronounced "lower"), which takes a resourceful lens l in $S \overset{C,k}{\Longleftrightarrow} V$ and packages it up with the interface of a basic lens in $S \overset{S}{\Longleftrightarrow} V$. This coercion realizes the procedure for using a resourceful lens described above, where we compute a correspondence between the chunks in the new and old views and use it to pre-align the resource before invoking the *put* function (it computes the alignment using the function *align*, which is described below):

$$\frac{l \in S \overset{C,k}{\Longleftrightarrow} V}{\lfloor l\rfloor \in S \overset{S}{\Longleftrightarrow} V}$$

$$\begin{array}{ll}
\mathit{get}\ s & = l.\mathit{get}\ s \\
\mathit{res}\ s & = s \\
\mathit{put}\ v\ s & = l.\mathit{put}\ v\ (c, r \circ g) \\
 & \quad \text{where } (c, r) = l.\mathit{res}\ s \\
 & \quad\ \ \text{and } g = \mathit{align}(v, l.\mathit{get}\ s) \\
\mathit{create}\ v & = l.\mathit{create}\ v\ \{||\}
\end{array}$$

4.1.7 Lemma: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens. Then $\lfloor l\rfloor$ is a basic lens in $S \overset{S}{\Longleftrightarrow} V$.

The *get* function is identical to $l.\mathit{get}$. The *res* function simply uses the entire source as the complement. The *put* function takes a view v and a complement s as arguments. It first uses $l.\mathit{res}$ to calculate a rigid complement c and resource r from s and then calculates a correspondence g between the locations of chunks in v and chunks in $l.\mathit{get}\ s$ using *align*. For now, we simply assume that *align* is a fixed function that takes two views and computes a correspondence between their chunks—formally, a partial injective function on their locations. We will describe mechanisms for specifying *align* in Section 4.3. Next, it composes r and g as functions, which has the effect of pre-aligning the complements in r with the chunks in v as specified by g. To finish the job, the *put* function passes v, c and $r \circ g$ to $l.\mathit{put}$, which produces the new source. The basic *create* function invokes $l.\mathit{create}$ with the view and the empty resource. Note that $\lfloor\cdot\rfloor$ does not assume anything about the *align* function except that it returns the identity alignment when its arguments are identical views—this property is needed to prove that $\lfloor l\rfloor$ obeys GetPut.

4.2 Syntax

Having defined the semantic space of resourceful lenses and developed some of their main properties, we now turn our attention to syntax and develop a collection of resourceful string lens combinators.

Notation

First, let us fix some notation for strings with chunks. We will describe the types of our resourceful lens primitives using regular expressions decorated with annotations indicating the locations of chunks. Let '$\langle$' and '$\rangle$' be fresh symbols not occurring in Σ. The set of chunk-annotated regular expressions is generated by the following grammar

$$\mathcal{A} ::= \mathcal{R} \mid \langle\mathcal{R}\rangle \mid \mathcal{A}|\mathcal{A} \mid \mathcal{A}{\cdot}\mathcal{A} \mid \mathcal{A}^*$$

where $\mathcal{R}$ ranges over ordinary regular expressions. Observe that every ordinary regular expression is also a chunk-annotated regular expression and that chunks only appear at the top level. The denotation $[\![A]\!]$ of a chunk-annotated regular expression A is a language of chunk-annotated strings—i.e., strings over the extended alphabet $(\Sigma \cup \{\text{‘}\langle\text{’}, \text{‘}\rangle\text{’}\})^*$ where occurrences of ‘$\langle$’ and ‘$\rangle$’ are balanced and non-nested. We write $\lfloor\cdot\rfloor$ for the erasure function that maps chunk-annotated strings to ordinary strings (by removing ’$\langle$’ and ’$\rangle$’ characters and mapping every other character to itself) and we lift $\lfloor\cdot\rfloor$ to regular expressions and languages in the obvious way.

We will use languages of chunk-annotated strings to “read off” the locations of chunks in ordinary strings. Given a language of chunk-annotated strings L and an ordinary string u in the erasure of L, we calculate the number $|u|$ of chunks in u, the chunk $u[n]$ at n in u, and so on, by first “parsing” u into a chunk-annotated string using L, and then using the explicit chunks in the result to give meaning to each of the concepts involving chunks. For example, if L is the language of chunk-annotated strings described by the chunk-annotated regular expression $\langle(\text{‘A’} \mid \ldots \mid \text{‘Z’})\cdot(\text{‘1’} \mid \ldots \mid \text{‘9’})\rangle^*$ and u is “A1B2C3”, then u parses into “$\langle$A1$\rangle\langle$B1$\rangle\langle$C1$\rangle$”, so the number $|u|$ of chunks in u is 3, the second chunk $u[2]$ in u is “B2”, and the string $u[2{:=}\text{“Z9”}]$ obtained by setting the second chunk in u to “Z9” is “A1Z9C3”.

Obviously for this way of identifying chunks in ordinary strings to make sense, we need to be sure that every string has a unique parse into a chunk-annotated string using L. Not every language has this property—e.g., “ab” has two different parses using the language $\{\text{“}\langle a\rangle b\text{”}, \text{“}a\langle b\rangle\text{”}\}$. To rule out such chunk ambiguous languages, we will be careful to ensure that every language of chunk-annotated strings under discussion uniquely determines the chunks of strings in its erasure—i.e., whenever we introduce a chunk-annotated regular language L as the source or view type of a resourceful lens, we will make sure that $\lfloor\cdot\rfloor$ is bijective on L. To lighten the notation, we will sometimes conflate L and $\lfloor L\rfloor$ when it is clear from context—e.g., we will write $u \in L$ instead of $u \in \lfloor L\rfloor$.

Primitives

With this notation fixed, we can now define some resourceful lens primitives.

Lift The first primitive lifts a basic lens to a resourceful lens. This makes it possible to use basic lenses such as copy and (<->) as resourceful lenses. As basic lenses have sets of ordinary strings in their types, the lifted lens does not have chunks so it satisfies the new resourceful lens laws vacuously.

$$\frac{\begin{array}{c} k' \in S' \overset{C'}{\Longleftrightarrow} V' \\ k \in S \overset{C}{\Longleftrightarrow} V \end{array}}{\widehat{k} \in S \overset{C,k'}{\Longleftrightarrow} V}$$

$$\begin{array}{ll}
\mathit{get}\ s & = k.\mathit{get}\ s \\
\mathit{res}\ s & = k.\mathit{res}\ s, \{|\,|\} \\
\mathit{put}\ v\ (c,r) & = k.\mathit{put}\ v\ c \\
\mathit{create}\ v\ r & = k.\mathit{create}\ v
\end{array}$$

4.2.1 Lemma: Let $k \in S \overset{C}{\Longleftrightarrow} V$ and $k' \in S' \overset{C'}{\Longleftrightarrow} V'$ be basic lenses. Then $\widehat{k}$ is a resourceful lens in $S \overset{C,k'}{\Longleftrightarrow} V$.

Note that the lens k' mentioned in the type of $\widehat{k}$ can be an arbitrary basic lens.

Match Another way to convert a basic lens to a resourceful lens is to place it in a chunk.

$$\frac{k \in S \overset{C}{\Longleftrightarrow} V}{\langle k \rangle \in \langle S \rangle \overset{\{\square\},k}{\Longleftrightarrow} \langle V \rangle}$$

$$\begin{array}{ll}
\mathit{get}\ s & = k.\mathit{get}\ s \\
\mathit{res}\ s & = \square, \{\!|1 \mapsto k.\mathit{res}\ s|\!\} \\
\mathit{put}\ v\ (\square, r) & = \begin{cases} k.\mathit{put}\ v\ (r(1)) & \text{if } 1 \in \text{dom}(r) \\ k.\mathit{create}\ v & \text{otherwise} \end{cases} \\
\mathit{create}\ v\ r & = \begin{cases} k.\mathit{put}\ v\ (r(1)) & \text{if } 1 \in \text{dom}(r) \\ k.\mathit{create}\ v & \text{otherwise} \end{cases}
\end{array}$$

4.2.2 Lemma: Let $k \in S \overset{C}{\Longleftrightarrow} V$ be a basic lens. Then $\langle k \rangle$ is a resourceful lens in $\langle S \rangle \overset{\{\square\},k}{\Longleftrightarrow} \langle V \rangle$.

The $\langle k \rangle$ lens (pronounced "match k") is the essential resourceful lens. It uses k to process strings in both directions, treating the whole source or view as a reorderable chunk. The *get* component of $\langle k \rangle$ simply passes off control to the basic lens k. The *res* function takes the source s and yields $\square$ as the rigid complement and $\{1 \mapsto k.\mathit{res}\ s\}$ as the resource. The *put* and *create* functions invoke $k.\mathit{put}$ on the view and $r(1)$ if r is defined on 1 and $k.\mathit{create}$ otherwise.

Concatenation Next, let us define resourceful versions on the regular operators, starting with concatenation:

$$\frac{\begin{array}{ll} l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 & \lfloor S_1 \rfloor \cdot^! \lfloor S_2 \rfloor \\ l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 & \lfloor V_1 \rfloor \cdot^! \lfloor V_2 \rfloor \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2) \overset{(C_1 \times C_2),k}{\Longleftrightarrow} (V_1 \cdot V_2)}$$

$$\begin{array}{ll}
\mathit{get}\ (s_1 \cdot s_2) & = (l_1.\mathit{get}\ s_1) \cdot (l_2.\mathit{get}\ s_2) \\
\mathit{res}\ (s_1 \cdot s_2) & = (c_1, c_2), (r_1 +\!\!+\ r_2) \\
\qquad \text{where } c_1, r_1 = l_1.\mathit{res}\ s_1 & \\
\qquad \quad \text{and } c_2, r_2 = l_2.\mathit{res}\ s_2 & \\
\mathit{put}\ (v_1 \cdot v_2)\ (c, r) & = (l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\ v_2\ (c_2, r_2)) \\
\qquad \text{where } c_1, c_2 = c & \\
\qquad \quad \text{and } r_1, r_2 = \mathit{split}(|v_1|, r) & \\
\mathit{create}\ (v_1 \cdot v_2)\ r & = (l_1.\mathit{create}\ v_1\ r_1) \cdot (l_2.\mathit{create}\ v_2\ r_2) \\
\qquad \text{where } r_1, r_2 = \mathit{split}(|v_1|, r) &
\end{array}$$

4.2.3 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be resourceful lenses such that $\lfloor S_1 \rfloor \cdot^! \lfloor S_2 \rfloor$ and $\lfloor V_1 \rfloor \cdot^! \lfloor V_2 \rfloor$. Then $l_1 \cdot l_2$ is a resourceful lens in $(S_1 \cdot S_2) \overset{(C_1 \times C_2),k}{\Longleftrightarrow} (V_1 \cdot V_2)$.

The *get* function splits the source string into two smaller strings s_1 and s_2, applies the *get* functions of l_1 and l_2 to these strings, and concatenates the resulting strings. The *res* function also splits the source into smaller strings s_1 and s_2 and applies the *res* functions of l_1 and l_2 to these strings. This yields rigid complements c_1 and c_2 and resources r_1 and r_2. It merges the complements into a pair (c_1, c_2) and combines the resources into a single finite map $(r_1 +\!\!+\ r_2)$. Because the same basic lens k is mentioned in the types of both l_1 and l_2, the resources r_1, r_2, and $(r_1 +\!\!+\ r_2)$ are all finite maps belonging to $\{\!|\mathbb{N} \mapsto k.C|\!\}$. This ensures that we can freely reorder the resource and pass portions of it to l_1 and l_2.

The *put* function splits each of the view, rigid complement, and resource in two, applies the *put* functions of l_1 and l_2 to the corresponding pieces of each, and concatenates the results. The *create* function is similar. Both split the resource r using $\mathit{split}(|v_1|, r)$ (where $|v_1|$ is the number of chunks of the first substring of the view). This yields two resources: one that behaves like r restricted to locations less than or equal to $|v_1|$ and another resource that behaves like r shifted down by $|v_1|$. Splitting the resource in this way ensures that every complement that is aligned with a chunk in the view remains aligned with the same chunk in the corresponding portion of the resource and substring of the view. Formally, the *split* operator is defined as follows:

$$
\begin{aligned}
(\pi_1(\mathit{split}(n,r)))(m) &= \begin{cases} r(m) & \text{if } m \leq n \text{ and } m \in \mathrm{dom}(r) \\ \text{undefined} & \text{otherwise} \end{cases} \\
(\pi_2(\mathit{split}(n,r)))(m) &= \begin{cases} r(m+n) & \text{if } (m+n) \in \mathrm{dom}(r) \\ \text{undefined} & \text{otherwise.} \end{cases}
\end{aligned}
$$

Note that $\mathit{split}(|r_1|, r_1 \mathbin{++} r_2) = (r_1, r_2)$. This property is essential for ensuring the GetPut law.

The requirement that l_1 and l_2 be defined over the same basic lens ensures that the resource has a uniform type. We might be tempted to relax the condition and allow l_1 and l_2 to be defined over different basic lenses, as long as those lenses had compatible complement types. However, this would lead to lenses with weaker properties. Conside $(\langle k_1 \rangle \cdot \langle k_2 \rangle)$ where k_1 and k_2 are defined as follows:

$$
\begin{aligned}
k_1 &\triangleq (\mathtt{a} \leftrightarrow \mathtt{a} \,|\, \mathtt{b} \leftrightarrow \mathtt{b}) \in \{\mathtt{a},\mathtt{b}\} \overset{\{\mathtt{a},\mathtt{b}\}}{\Longleftrightarrow} \{\mathtt{a},\mathtt{b}\} \\
k_2 &\triangleq (\mathtt{a} \leftrightarrow \mathtt{b} \,|\, \mathtt{b} \leftrightarrow \mathtt{a}) \in \{\mathtt{a},\mathtt{b}\} \overset{\{\mathtt{a},\mathtt{b}\}}{\Longleftrightarrow} \{\mathtt{a},\mathtt{b}\}
\end{aligned}
$$

Invoking *put* on “`aa`” yields “`ab`” as a result (as k_1 and k_2 are “bijective” lenses, only the view affects the evaluation of *put*). Now suppose that we swap the chunks of “`aa`”. By Lemma 4.1.5, the *put* function should produce “`ba`”—i.e., the string obtained by swapping the chunks of “`ab`”. But this is not what happens. Swapping the chunks of “`aa`” is a no-op, so *put* produces the same result as before. Thus, although it is tempting to allow resourceful lenses that use different lenses to process chunks, we don’t do this, because it would require sacrificing properties such as Lemma 4.1.5.

Kleene Star The Kleene star operator iterates a resourceful lens:

$$
\frac{\lfloor S \rfloor^{!*} \qquad \lfloor V \rfloor^{!*} \qquad l \in S \overset{C,k}{\Longleftrightarrow} V}{l* \in S* \overset{(C\ \text{list}),k}{\Longleftrightarrow} V\ *}
$$

$$
\begin{aligned}
&\mathit{get}\,(s_1 \cdots s_n) &&= (l.\mathit{get}\, s_1) \cdots (l.\mathit{get}\, s_n) \\
&\mathit{res}\,(s_1 \cdots s_n) &&= [c_1, \ldots, c_n], (r_1 \mathbin{++} \ldots \mathbin{++} r_n) \\
&\quad \text{where } c_i, r_i = l.\mathit{res}\ s_i \text{ for } i \in \{1, \ldots, n\} \\
&\mathit{put}\,(v_1 \cdots v_n)\,(c,r) = s'_1 \cdots s'_n \\
&\quad \text{where } s'_i = \begin{cases} l.\mathit{put}\, v_i\,(c_i, r_i) & i \in \{1, \ldots, \min(n,m)\} \\ l.\mathit{create}\, v_i\, r_i & i \in \{m+1, \ldots, n\} \end{cases} \\
&\qquad \text{and } [c_1, \cdots, c_m] = c \\
&\qquad \text{and } r'_0 = r \\
&\qquad \text{and } r_i, r'_i = \mathit{split}(|v_i|, r'_{(i-1)}) \text{ for } i \in \{1, \ldots, n\} \\
&\mathit{create}\,(v_1 \cdots v_n)\, r &&= (l.\mathit{create}\, v_1\, r_1) \cdots (l.\mathit{create}\, v_n\, r_n) \\
&\quad \text{where } r'_0 = r \\
&\qquad \text{and } r_i, r'_i = \mathit{split}(|v_i|, r'_{(i-1)}) \text{ for } i \in \{1, \ldots, n\}
\end{aligned}
$$

4.2.4 Lemma: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens such that $\lfloor S \rfloor^{!*}$ and $\lfloor V \rfloor^{!*}$. Then l^* is a resourceful lens in $S^* \overset{K\ list,R}{\Longleftrightarrow} V^*$.

The *get* and *res* components of the Kleene star lens are straightforward generalizations of the corresponding components of the concatenation lens. The *put* function, however is different. Because it is a total function, it needs to handle situations where the number of substrings of the view is different than the number of rigid complements. When there are more rigid complements than substrings of the view, the lens simply discards the extra complements. When there are more substrings than rigid complements, it processes the extra substrings using *l.create*. This is the reason that *create* takes a resource as an argument—the resource may contain complements for chunks in the extra substrings of the view.

To illustrate the last few definitions, let us consider a simple example:

```
let k : lens = key [A-Z] . del [a-z]
let l : lens = <k> . (copy "," . <k>)*
```

The lens `k` copies an upper-case letter from source to view and deletes a lower-case letter while `l` uses the match, concatenation, and Kleene-star lenses to iterate `k` over a non-empty list of comma-separated chunks (the Boomerang implementation automatically inserts coercions to lift basic lenses to resourceful lenses using $(\widehat{\cdot})$ and to convert the top-level resourceful lens to a basic lens using $\lfloor\cdot\rfloor$ when we invoke its `get` or `put` component with string arguments). The behavior of `l`.*get* is straightforward—e.g., it maps “`Xx,Yy,Zz`” to “`X,Y,Z`”. However, `l`.*put* is more sophisticated—it restores the lower-case letters from source chunks by resourceful up upper-case letters in the old and new views. For instance, if we insert “`W`” into the middle of the view, *put* behaves as follows:

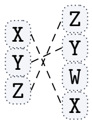
```
l.put "Z,Y,W,X" into "Xx,Yy,Zz" = "Zz,Yy,Wa,Xx"
```

Let us trace the evaluation of this example in detail. First, the $\lfloor$`l`$\rfloor$.*put* lens uses `l`.*res* to calculate a rigid complement c and resource r from the source string:

$$c = (\square, [(\text{“}\texttt{,}\text{”}, \square), (\text{“}\texttt{,}\text{”}, \square)]) \qquad r = \left\{\left|\begin{array}{l} 1 \mapsto \text{“}\texttt{Xx}\text{”} \\ 2 \mapsto \text{“}\texttt{Yy}\text{”} \\ 3 \mapsto \text{“}\texttt{Zz}\text{”} \end{array}\right|\right\}$$

Next, it calculates a correspondence g between the chunks in the old view and the new view and composes this correspondence with r to obtain the pre-aligned resource (for the moment we are ignoring how the lens computes g—see Section 4.3):

$$g =$$



$$= \left\{\left|\begin{array}{l} 4 \mapsto 1 \\ 2 \mapsto 2 \\ 1 \mapsto 3 \end{array}\right|\right\} \qquad (r \circ g) = \left\{\left|\begin{array}{l} 4 \mapsto \text{“}\texttt{Xx}\text{”} \\ 2 \mapsto \text{“}\texttt{Yy}\text{”} \\ 1 \mapsto \text{“}\texttt{Zz}\text{”} \end{array}\right|\right\}$$

Finally, it invokes `l`.*put* on the new view c and pre-aligned resource $r \circ g$. The effect is that the lower-case letters are restored to the chunk containing the corresponding upper-case letter. Note that the third chunk, `W` is created fresh because the resource $r \circ g$ is undefined on 3.

Union The final regular operator forms the union of two resourceful lenses:

$$\frac{\begin{array}{c}\lfloor S_1\rfloor \cap \lfloor S_2\rfloor = \emptyset \qquad \lfloor V_1\rfloor \cap \lfloor V_2\rfloor \subseteq \lfloor V_1 \cap V_2\rfloor \\ l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 \\ l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2\end{array}}{l_1 \,|\, l_2 \in (S_1 \cup S_2) \overset{(C_1+C_2),k}{\Longleftrightarrow} (V_1 \cup V_2)}$$

$$\begin{aligned}
\mathit{get}\; s &= \begin{cases} l_1.\mathit{get}\; s \text{ if } s \in \lfloor S_1\rfloor \\ l_2.\mathit{get}\; s \text{ if } s \in \lfloor S_2\rfloor \end{cases} \\
\mathit{res}\; s &= \begin{cases} \mathit{Inl}(l_1.\mathit{res}\; s) \text{ if } s \in \lfloor S_1\rfloor \\ \mathit{Inr}(l_2.\mathit{res}\; s) \text{ if } s \in \lfloor S_2\rfloor \end{cases} \\
\mathit{put}\; v\; (c, r) &= \begin{cases} l_1.\mathit{put}\; v\; (c_1, r) \text{ if } v \in \lfloor V_1\rfloor \wedge c = \mathit{Inl}(c_1) \\ l_2.\mathit{put}\; v\; (c_2, r) \text{ if } v \in \lfloor V_2\rfloor \wedge c = \mathit{Inr}(c_2) \\ l_1.\mathit{create}\; v\; r \quad \text{ if } v \notin \lfloor V_2\rfloor \wedge c = \mathit{Inl}(c_2) \\ l_2.\mathit{create}\; v\; r \quad \text{ if } v \notin \lfloor V_1\rfloor \wedge c = \mathit{Inr}(c_1) \end{cases} \\
\mathit{create}\; v\; r &= \begin{cases} l_1.\mathit{create}\; v\; r \text{ if } v \in \lfloor V_1\rfloor \\ l_2.\mathit{create}\; v\; r \text{ if } v \notin \lfloor V_1\rfloor \end{cases}
\end{aligned}$$

4.2.5 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be resourceful lenses such that $\lfloor S_1\rfloor \cap \lfloor S_2\rfloor = \emptyset$ and $\lfloor V_1\rfloor \cap \lfloor V_2\rfloor \subseteq \lfloor V_1 \cap V_2\rfloor$. Then $l_1 \,|\, l_2$ is a resourceful lens in $(S_1 \cup S_2) \overset{(C_1+C_2),k}{\Longleftrightarrow} (V_1 \cup V_2)$.

The *get* function selects l_1.*get* or l_2.*get* by testing whether the source string belongs to $\lfloor S_1\rfloor$ or $\lfloor S_2\rfloor$. Similarly, *res* selects one of l_1.*res* or l_2.*res* by testing the source string. It places the resulting rigid complement in a tagged sum, producing $\mathit{Inl}(c)$ if the source belongs to $\lfloor S_1\rfloor$ and $\mathit{Inr}(c)$ if it belongs to $\lfloor S_2\rfloor$. It does not tag the resource—because l_1 and l_2 are defined over the same basic lens k for chunks, we can safely pass a resource computed by l_1.*res* to l_2.*put* and vice versa.

The *put* function is slightly more complicated, because the typing rule allows the view types to overlap. It tries to select one of l_1.*put* or l_2.*put* using the view and uses the rigid complement disambiguate cases where the view belongs to both $\lfloor V_1\rfloor$ and $\lfloor V_2\rfloor$. The *create* function is similar. Note that because *put* is a total function, it needs to handle cases where the view belongs to $(\lfloor V_1\rfloor - V_2)$ but the complement is of the form $\mathit{Inl}(c)$. To satisfy the PutGet law, it must invoke one of l_1's component functions, but it cannot invoke l_1.*put* because the rigid complement c does not necessarily belong to C_1. It discards c and uses l_1.*create* instead.

The side condition $(\lfloor V_1\rfloor \cap \lfloor V_2\rfloor) \subseteq \lfloor V_1 \cap V_2\rfloor$ in the typing rule for union ensures that $(V_1 \,|\, V_2)$ is chunk unambiguous—i.e., that strings in the intersection $(V_1 \cap V_2)$ have unique parses. It rules out languages of chunk-annotated strings such as $(a\cdot\langle b\rangle \,|\, \langle a\rangle\cdot b)$.

Interestingly, the resourceful version of the union lens can pass source information between branches in the *put* direction. This recovers some of the functionality of "fixup functions" described in Chapter 2. To illustrate, consider the following example:

```
let k : lens = del [0-9]
let l1 : lens = copy [A-Z] . <k>
let l2 : lens = <k>
let l : lens = (l1 | l2)
```

The lens l is similar to a basic lens we discussed in Chapter 3, except that the lens `k`, which deletes the number, occurs within a chunk. This means that when we put "`A`" back into "`3`", the "`3`" will be restored from the source even though the source and view come from different sides of the union:

```
test l.put "A" into "3" = "A3"
```

Thus, using resources, we can realize some of the benefits of fixup functions without having to describe them explicitly.

Composition The composition operator puts two resourceful lenses in sequence:

$$\frac{\begin{array}{c} l_1 \in S \overset{C_1,k_1}{\Longleftrightarrow} U \\ l_2 \in U \overset{C_2,k_2}{\Longleftrightarrow} V \end{array}}{l_1;l_2 \in S \overset{(C_1 \otimes C_2),(k_1;k_2)}{\Longleftrightarrow} V}$$

$$\begin{array}{ll}
\textit{get } s & = l_2.\textit{get } (l_1.\textit{get } s) \\
\textit{res } s & = \langle c_1, c_2 \rangle, \textit{zip } r_1 \; r_2 \\
\quad \text{where } c_1, r_1 = l_1.\textit{res } s & \\
\qquad \text{and } c_2, r_2 = l_2.\textit{res } (l_1.\textit{get } s) & \\
\textit{put } v \; (\langle c_1, c_2 \rangle, r) & = l_1.\textit{put } (l_2.\textit{put } v \; (c_2, r_2)) \; (c_1, r_1) \\
\quad \text{where } r_1, r_2 = \textit{unzip } r & \\
\textit{create } v \; r & = l_1.\textit{create } (l_2.\textit{create } v \; r_2) \; r_1 \\
\quad \text{where } r_1, r_2 = \textit{unzip } r &
\end{array}$$

4.2.6 Lemma: Let $l_1 \in S \overset{C_1,k_1}{\Longleftrightarrow} U$ and $l_2 \in U \overset{C_2,k_2}{\Longleftrightarrow} V$ be resourceful lenses. Then $(l_1;l_2)$ is a resourceful lens in $S \overset{(C_1 \otimes C_2),(k_1;k_2)}{\Longleftrightarrow} V$.

Composition is especially interesting as a resourceful lens because it propagates alignment information through two phases of computation. The *get* function applies l_1.*get* and l_2.*get* in sequence. The *res* function applies l_1.*res* to the source s, yielding a rigid complement c_1 and resource r_1, and l_2.*res* to (l_1.*get* s), yielding c_2 and r_2. It merges the rigid complements into a pair $\langle c_1, c_2 \rangle$ and combines the resources by zipping them together, where the *zip* function is defined as follows:[1]

$$(\textit{zip } r_1 \; r_2)(m) = \begin{cases} \langle r_1(m), r_2(m) \rangle & \text{if } m \in \text{dom}(r_1) \cap \text{dom}(r_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that we have the following equalities

$$\begin{array}{rcll}
\text{dom}(r_1) & = & \textit{locs}(s) & \text{by } \textsc{ResChunks} \text{ for } l_1 \\
 & = & \textit{locs}(l.\textit{get } s) & \text{by } \textsc{GetChunks} \text{ for } l_1 \\
 & = & \text{dom}(r_2) & \text{by } \textsc{ResChunks} \text{ for } l_2
\end{array}$$

so *zip* r_1 r_2 is defined on the same locations as $\text{dom}(r_1)$ and $\text{dom}(r_2)$.

The *put* function unzips the resource and applies l_2.*put* and l_1.*put* in that order. The *unzip* function on finite maps is defined as follows

$$(\pi_i(\textit{unzip } r))(m) = \begin{cases} c_i & \text{if } r(m) = \langle c_1, c_2 \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $i \in \{1, 2\}$. Because the zipped resource represents the resources generated by l_1 and l_2 together, rearranging the resource has the effect of pre-aligning the resources for both phases of computation.

To illustrate the behavior of the composition lens, consider the following example:

[1] The angle brackets distinguish these pairs from the pairs generated as rigid complements in the concatenation lens.

```
let k1 : lens = del [0-9] . copy [A-Z] . copy [a-z]
let k2 : lens = del [A-Z] . key (copy [a-z])
let l : lens =
  <k1> . (copy "," . <k1>)* ;
  <k2> . (copy "," . <k2>)*
```

The *get* function takes a non-empty list of comma-separated chunks containing a number, an upper-case letter, and a lower-case letter, and deletes the number in the first phase and the upper-case letter in the second phase:

```
l.get "1Aa,2Bb,3Cc" = "a,b,c"
```

The resource produced by *res* represents the upper-case letter and number together, so the *put* function restores both to the appropriate chunk:

```
l.put "b,a" into "1Aa,2Bb,3Cc" = "2Bb,1Aa"
```

The typing rule for the composition lens requires that the view type of l_1 be identical to the source type of l_2. In particular, the chunks in these types must be identical. Intuitively, this makes sense—the only way that the *put* function can reasonably translate alignments on the view back through both phases of computation to the source is if the chunks in the types of each lens agree. However, in some situations, it is useful to compose lenses that have identical erased types but different notions of chunks—e.g., one lens does not have any chunks, while the other lens does have chunks. To do this "asymmetric" form of composition, we can convert both lenses to basic lenses using $\lfloor\cdot\rfloor$, which forgets the chunks in the source and view, compose them as basic lenses, and then lift the result back to a resourceful lens.

4.3 Alignments

So far, our discussion has focused exclusively on the mechanisms of resourceful lenses, which extend basic lenses with a notion of chunks and provide an interface for supplying a lens with explicit directives about how source chunks should be aligned against the view. But we have not yet said where these alignments come from!

In this section, we describe the strategies for computing alignments that we have implemented in the Boomerang language [FP08]. We describe three different alignment "species" and we present mechanisms for tuning alignment strategies using notions of "keys" and "thresholds". Because alignment is a fundamentally heuristic operation, the choice of a proper alignment function depends intimately on the details of the application at hand. One of the main strengths of our framework is that it can be instantiated with arbitrary alignment functions—the well-behavedness of our resourceful lens combinators does not hinge on any special properties of the alignment function. The only property we require is that it return the identity alignment when its arguments are identical. Thus, the mechanisms described in this section should not be taken as exhaustive; it would be easy to extend them with additional mechanisms if needed.

Species Boomerang currently supports three alignment "species", depicted in Figure 4.1 (a-c):

- **Positional:** The alignment matches up chunks by position. If one of the lists has more chunks than the other, the extra chunks at the end of the longer list do not match any chunk in the shorter list.
- **Set-like:** The alignment minimizes the sum of the total edit distances between pairs of matched chunks and the lengths of unmatched chunks.

- **Diff-like:** The alignment minimizes the same function as the set-like strategy, but only considers alignments without "crossing" edges. This heuristic can be computed efficiently using a variant of the standard algorithm for computing the longest common subsequence of two lists.

These species are illustrated in the following simple examples:

```
let l = key [A-Z] . del [0-9]
<pos:l>*.put "BCA" into "A1B2C3" = "B1C2A3"
<set:l>*.put "BCA" into "A1B2C3" = "B2C3A1"
<dif:l>*.put "BCA" into "A1B2C3" = "B2C3A0"
```

These examples also illustrate how Boomerang programmers indicate a species to use with chunks. The match combinator implemented in Boomerang actually takes two arguments: an annotation that specifies the alignment species and a basic lens for chunks. The shorthand <*l*> desugars to <set:*l*>. When we coerce a resourceful lens to a basic lens using $\lfloor \cdot \rfloor$, it instantiates the *align* function using the species indicated in the annotation (recall that this coercion is automatically implemented by the Boomerang typechecker when we invoke the get or put component of a resourceful lens). Boomerang's typechecker checks that the same annotation is used on every instance of the match combinator—e.g., it disallows (<pos:l > . <dif:l>), which specifies two different species for chunks.

Keys Typically, we do not want to consider the entire contents of chunks when we compute alignments. Boomerang includes two primitives, key and nokey, that allow programmers to control the portions of each chunk that are used to compute alignments. Both of these combinators take a resourceful lens as an argument, but they do not change the *get*/*put* behavior of the lens they enclose. Instead, they add extra annotations to the view type that we use to "read off" a key for each chunks (just like we use annotations to "read off" the locations of chunks). When the *align* function computes an alignment for two lists of chunks, it uses the view type to extract the regions of each chunk that are marked as keys and ignores the rest of each chunk.

To illustrate the use of keys, consider a simple example:

```
let k = del [0-9] . copy [A-Z] . copy [a-z]
let l = <set:k> . (copy "," . <set:k>)*
l.put "Cc,Bb,Aa" into "1Aa,2Bb,3Cc" = "1Cc,2Bb,3Aa"
```

Although this program aligns chunks using the set-like species, it behaves positionally because the view type does not contain any key annotations—i.e., the key of every chunk is the empty string. The following revised version of the lens has a key annotation

```
let k = del [0-9] . key (copy [A-Z]) . copy [a-z]
let l = <k> . (copy "," . <k>)*
```

so its *put* function matches up chunks using the upper-case letters in the view:

```
l.put "Cc,Bb,Aa" into "1Aa,2Bb,3Cc" = "3Cc,2Bb,1Aa"
```

Note that lower-case letters, which are not marked as a part of the key, do not affect alignment:

```
l.put "Ca,Bb,Ac" into "1Aa,2Bb,3Cc" = "3Ca,2Bb,1Ac"
```

The nokey primitive is dual to key—it removes the key annotation on the view type of the lens it encloses. We can use nokey to write an equivalent version of the previous lens:

```
let k = key (del [0-9] . copy [A-Z] . nokey (copy [a-z]))
let l = <k> . (copy "," . <k>)*
```

The simple mechanisms for indicating keys provided by the key and nokey primitives suffice for many practical examples, but there are many ways that it could be extended. For example, we could provide programmers with mechanisms for generating unique keys or for building keys structured as tuples or records (rather than simply flattening the regions of each chunk marked as a key into a string). We plan to explore these ideas in future work.

Thresholds The set-like and diff-like species compute alignments by minimizing the sum of the total edit distances between matched chunks and the lengths of unmatched chunks. In some applications, it is important to *not* match up chunks that are "too different", even if aligning those chunks would result in a minimal cost alignment. For instance, in the following program, where keys are three characters long

```
let k : lens = key [A-Z]{3} . del [0-9]
let l : lens = (<set:k> . copy ";")*
l.put "DBD;CCC;AAA;" into "AAA1;BBB2;CCC3;" =
  "DBD2;CCC3;AAA1;"
```

we might like the DBD and BBB2 chunks to not be aligned with each other. However, the set-like species aligns them because the cost of a two-character edit is less than the six-character edit of deleting BBB from the view and adding DBD. To achieve the behavior we want, we can add a threshold, as shown in the following example:

```
let l : lens = (<sim 50:k> . copy ";")*
l.put "DBD;CCC;AAA;" into "AAA1;BBB2;CCC3;" =
  "DBD0;CCC3;AAA1;"
```

The sim species is similar to set, but takes an integer n as an argument. It minimizes the total edit distances between aligned chunks, like set, but it only aligns chunks whose longest common subsequence is at least n% of the lengths of their keys. The set species desugars to (sim 0) and "dictionary" alignment can be simulated using (sim 100) [BFP$^+$08]. The revised version of the l lens does not align DBD with BBB2 because the longest common subsequence computed from their keys does not meet the threshold. The diff species also supports thresholds. We often use diff with a threshold to align chunks containing of unstructured text.

4.4 Extensions

Our design for resourceful lenses is based on three assumptions:

1. the source and view only contain chunks at the top level,
2. the same lens is used to process every chunk, and
3. the lens does not reorder chunks.

However, it is often important to be able to use different lenses to process multiple kinds of chunks, to nest chunks within other chunks, and to reorder chunks in going from source to view. This section describes how we can extend the resourceful lens framework to accommodate each of these features.

Nested Chunks

Some sources contain reorderable information at several different levels of structure. For example, suppose that the source is a Wiki with three levels of structure: sections, subsections, and paragraphs,

```
=Grand Tours=
The grand tours are major cycling races...
==Giro d'Italia==
The Giro is usually held in May and June...
==Tour de France==
The Tour is usually held in July...
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
Milan-San Remo is held in March...
==Paris-Roubaix==
Paris-Roubaix is held in mid-April...
```

and the view is a simplified list of section and subsection headings:

```
Grand Tours:
  Giro d'Italia
  Tour de France
Classics:
  Milan-San Remo
  Paris-Roubaix
```

If we update the view by reordering the sections and adding some new subsections to each,

```
Classics:
  Milan-San Remo
  Ronde van Vlaanderen
  Paris-Roubaix
Grand Tours:
  Giro d'Italia
  Tour de France
  Vuelta a Espana
```

we would the paragraphs to be restored to the appropriate section or subsection:

```
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
Milan-San Remo is held in March...
==Ronde van Vlaanderen==
==Paris-Roubaix==
Paris-Roubaix is held in mid-April...
=Grand Tours=
The grand tours are major cycling races...
==Giro d'Italia==
The Giro is usually held in May and June...
```

```
==Tour de France==
The Tour is usually held in July...
==Vuelta a Espana==
```

To do this, we need a lens that aligns chunks at several levels of structure, not just at the top-level. Using the lower combinator we can convert a resourceful lens for the nested chunks to a basic lens and use it to process the nested chunks as in the following program:

```
let HEADING : regexp = ([^=\n]* - ( " " . [^]* ))
let TEXT : regexp = (([^=\n] . [^\n]* )? . "\n")*
let paragraphs : lens = del (TEXT . ("\n" . TEXT)* )
let subsection : lens =
  ins "  " . del "==" . key HEADING . del "==" .
  copy "\n" .
  paragraphs
let section : lens =
  del "=" . key HEADING . del "=" . ins ":" .
  copy "\n" .
  paragraphs . lower < set : subsection >*
let wiki : lens =
  < set : section >*
```

The paragraph lens deletes blocks of text separated by double newline characters. The subsection lens inserts two space characters as indentation, copies the subsection heading, and deletes the paragraphs that follow. The section lens copies the heading, inserts a colon character, deletes the paragraphs that follow, and then processes a list of subsections. The top-level wiki lens processes a list of sections.

The main thing to notice about this program is that we can use lower to build resourceful lenses that process chunks using other resourceful lenses even though the match combinator takes a basic lens as an argument. Lenses constructed in this way align chunks in strict nested fashion—e.g., in this example, the top-level chunks for sections are first aligned against other sections and the nested chunks for subsections within each section are aligned each other.

Tags

In other applications, we need to use several different basic lenses to process chunks. For example, suppose that we wanted to build a version of the wiki lens that aligns subsections and sections separately. Why would we want this? Observe that the nested alignments computed by the wiki lens just described never align subsections in different sections. Thus, if we update the view by moving the heading for the "Paris-Roubaix" subsection from the classics section to the grand tours section

```
Classics:
  Milan-San Remo
Grand Tours:
  Paris-Roubaix
  Giro d'Italia
  Tour de France
```

the paragraph under the Paris-Roubaix subsection will be lost when we invoke the *put* function

```
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
Milan-San Remo is held in March...
=Grand Tours=
The grand tours are major cycling races...
==Paris-Roubaix==
==Giro d'Italia==
The Giro is usually held in May and June...
==Tour de France==
The Tour is usually held in July...
```

because the alignment follows the nesting structure of the document.

In this example, it would be better to align section and subsections separately instead of following the structure of the document. To do this, we need to generalize resourceful lenses to allow multiple kinds of chunks in the same program. Here is a revised version of the wiki lens written using “tags” that has the behavior we want:

```
let section : lens =
  del "=" . key HEADING . del "=" . ins ":" .
  copy "\n" .
  paragraphs
let wiki : lens =
  ( < tag "section" set : section > .
    < tag "subsection" set : subsection >* )*
```

Rather than having nested chunks, this lens has two differently-tagged top-level chunks—one for sections and another for subsections. The tag primitive gives distinct names to these chunks and indicates that the two kinds of chunks should be handled separately by the lens. On the same inputs as above, the *put* function produces a new source

```
=Classics=
The classics are one-day cycling races...
==Milan-San Remo==
Milan-San Remo is held in March...
=Grand Tours=
The grand tours are major cycling races...
==Paris-Roubaix==
Paris-Roubaix is held in mid-April...
==Giro d'Italia==
The Giro is usually held in May and June...
==Tour de France==
The Tour is usually held in July...
```

where the paragraph under the Paris-Roubaix subsection is restored from the source. Extending resourceful lenses with tags is simple—we generalize each of our structures with an extra level of indirection for tags. For example, we change the type of resources to finite maps from tags to locations to complements and we compute alignments by tag.

Swap

All of the resourceful lenses we have seen so far map chunks in the source through to the same chunks in the view and vice versa, but in some applications we need resourceful lenses that reorder chunks. The swap operator, written $l_1 \sim l_2$, behaves like the concatenation lens, but swaps the order of strings in the view. Adding swap as a resourceful lens complicates the story significantly because it makes it possible to construct lenses that reorder chunks. Lenses that reorder chunks break the protocol for using resourceful lenses where we pre-align the resource using a correspondence computed for the view. They also cause problems with the sequential composition operator—in general, the two lenses will reorder the chunks in different ways, so it will not make sense to simply zip the resources generated by each lens together and align them against the view.

To recover the behavior we want in the presence of primitives that reorder chunks, we need to keep track of the permutation on chunks that is computed by the lens. Therefore, we add a new component to every resourceful lens

$$l.\mathit{perm} \in \Pi\ s : \lfloor S \rfloor.\ \mathsf{Perms}(\mathit{locs}(s))$$

that computes the permutation on chunks realized by the *get* function.

It is straightforward to add *perm* to each of the lenses we have seen so far—e.g., the lift primitive returns the empty permutation, match returns the identity permutation on its only chunk, and the concatenation operator merges the permutations returned by its sublenses in the obvious way.

We also need to generalize the ChunkPut, ChunkCreate, NoChunkPut, and NoChunkCreate laws using *perm*—the old versions do not hold for lenses that permute the order of chunks in going from source to view:

$$\frac{\begin{array}{c} n \in (\mathit{locs}(v) \cap \mathrm{dom}(r)) \\ (l.\mathit{perm}\,(l.\mathit{put}\,v\,(c,r)))(m) = n \end{array}}{(l.\mathit{put}\,v\,(c,r))[m] = k.\mathit{put}\,v[n]\,(r(n))} \qquad (\textsc{ChunkPut})$$

$$\frac{\begin{array}{c} n \in (\mathit{locs}(v) \cap \mathrm{dom}(r)) \\ (l.\mathit{perm}\,(l.\mathit{create}\,v\,r))(m) = n \end{array}}{(l.\mathit{create}\,v\,r)[m] = k.\mathit{put}\,v[n]\,(r(n))} \qquad (\textsc{ChunkCreate})$$

$$\frac{\begin{array}{c} n \in (\mathit{locs}(v) - \mathrm{dom}(r)) \\ (l.\mathit{perm}\,(l.\mathit{put}\,v\,(c,r)))(m) = n \end{array}}{(l.\mathit{put}\,v\,(c,r))[m] = k.\mathit{create}\,v[n]} \qquad (\textsc{NoChunkPut})$$

$$\frac{\begin{array}{c} n \in (\mathit{locs}(v) - \mathrm{dom}(r)) \\ (l.\mathit{perm}\,(l.\mathit{create}\,v\,r))(m) = n \end{array}}{(l.\mathit{create}\,v\,r)[m] = k.\mathit{create}\,v[n]} \qquad (\textsc{NoChunkCreate})$$

These laws generalize the laws given in Section 4.1. The ChunkPut law stipulates that the mth chunk in the source produced by *put* must be identical to the structure produced by applying $k.\mathit{put}$ to the nth chunk in the view and the element $r(n)$ in the resource, where the permutation computed by the *perm* function on the source maps m to n. The other laws are similar generalizations of the previous versions.

Composition Using *perm*, we can define a better version of the composition lens that uses the permutation on chunks computed in each phase:

$$\frac{\begin{array}{c} l_1 \in S \overset{C_1,k_1}{\Longleftrightarrow} U \\ l_2 \in U \overset{C_2,k_2}{\Longleftrightarrow} V \end{array}}{(l_1;l_2) \in S \overset{(C_1 \otimes C_2),(k_1;k_2)}{\Longleftrightarrow} V}$$

$$\begin{array}{ll}
\textit{get } s & = l_2.\textit{get } (l_1.\textit{get } s) \\
\textit{res } s & = \langle c_1, c_2 \rangle, \textit{zip } (r_1 \circ p_2^{-1}) \; r_2 \\
\quad \text{where } c_1, r_1 = l_1.\textit{res } s & \\
\quad\quad \text{and } c_2, r_2 = l_2.\textit{res } (l_1.\textit{get } s) & \\
\quad\quad \text{and } p_2 = l_2.\textit{perm } (l_1.\textit{get } s) & \\
\textit{perm } s & = (l_2.\textit{perm } (l_1.\textit{get } s)) \circ (l_1.\textit{perm } s) \\
\textit{put } v \; (\langle c_1, c_2 \rangle, r) & = l_1.\textit{put } (l_2.\textit{put } v \; (c_2, r_2)) \; (c_1, r_1 \circ p_2^{-1}) \\
\quad \text{where } r_1, r_2 = \textit{unzip } r & \\
\quad\quad \text{and } p_2 = l_2.\textit{perm } (l_2.\textit{put } v \; (c_2, r_2)) & \\
\textit{create } v \; r & = l_1.\textit{create } (l_2.\textit{create } v \; r_2) \; (r_1 \circ p_2^{-1}) \\
\quad \text{where } r_1, r_2 = \textit{unzip } r & \\
\quad\quad \text{and } p_2 = l_2.\textit{perm } (l_2.\textit{create } v \; r_2) &
\end{array}$$

The *res* function applies the inverse of the permutation computed by l_2 on the intermediate view to the resource computed by l_1, which puts it into the "view order" of l_2. Similarly, the *put* function puts r_1 back into the view order of l_1.

Swap The swap lens is defined as follows:

$$\frac{\begin{array}{cc} l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 & \lfloor S_1 \rfloor \cdot^! \lfloor S_2 \rfloor \\ l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 & \lfloor V_2 \rfloor \cdot^! \lfloor V_1 \rfloor \end{array}}{l_1 \sim l_2 \in (S_1 \cdot S_2) \overset{(C_2 \times C_1),k}{\Longleftrightarrow} (V_2 \cdot V_1)}$$

$$\begin{array}{ll}
\textit{get} \, (s_1 \cdot s_2) & = (l_2.\textit{get} \, s_2) \cdot (l_1.\textit{get} \, s_1) \\
\textit{res} \, (s_1 \cdot s_2) & = (c_2, c_1), (r_2 \mathbin{++} r_1) \\
\quad \text{where } c_1, r_1 = l_1.\textit{res} \, s_1 & \\
\quad\quad \text{and } c_2, r_2 = l_2.\textit{res} \, s_2 & \\
\textit{perm} \, (s_1 \cdot s_2) & = (l_2.\textit{perm} \, s_2) ** (l_1.\textit{perm} \, s_1) \\
\textit{put} \, (v_2 \cdot v_1) \, (c, r) & = (l_1.\textit{put} \, v_1 \, (c_1, r_1)) \cdot (l_2.\textit{put} \, v_2 \, (c_2, r_2)) \\
\quad \text{where } c_2, c_1 = c & \\
\quad\quad \text{and } r_2, r_1 = \textit{split}(|v_2|, r) & \\
\textit{create} \, (v_2 \cdot v_1) \, r & = (l_1.\textit{create} \, v_1 \, r_1) \cdot (l_2.\textit{create} \, v_2 \, r_2) \\
\quad \text{where } r_2, r_1 = \textit{split}(|v_2|, r) &
\end{array}$$

4.4.1 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be resourceful lenses such that $\lfloor S_1 \rfloor \cdot^! \lfloor S_2 \rfloor$ and $\lfloor V_1 \rfloor \cdot^! \lfloor V_2 \rfloor$. Then $l_1 \sim l_2$ is a resourceful lens in $(S_1 \cdot S_2) \overset{(C_2 \times C_1),k}{\Longleftrightarrow} (V_2 \cdot V_1)$.

Like the concatenation lens, the *get* component of swap splits the source string in two and applies $l_1.\textit{get}$ and $l_2.\textit{get}$ to the resulting substrings. However, before it concatenates the results, it swaps their order.

The *res*, *put*, and *create* functions are similar. The *perm* component of swap combines permutations using the $(**)$ operator

$$(q_2 ** q_1)(m) = \begin{cases} q_1(m) + |q_2| & \text{if } m \leq |q_1| \\ q_2(m - |q_1|) & \text{otherwise} \end{cases}$$

which is similar to the $(++)$ operator for resources.

4.5 Summary

Resourceful lenses extend the mechanisms of basic lenses with new constructs for handling ordered data. These features makes it possible to handle situations where the update to the view involves a reordering. Semantically, we revise the architecture of lenses to separate the handling of rigidly ordered and reorderable data from each other and we add new laws ensuring that the components of lenses use alignment information correctly. The resulting architecture can be instantiated with arbitrary functions for computing alignments. Syntactically, we add a new combinator for specifying the reorderable chunks in the source and view, we reinterpret each of our core lens combinators as resourceful lenses. We also add several new primitives for specifying and tuning alignment strategies.

Chapter 5

Secure Lenses

"Whoever wishes to keep a secret must hide the fact that he possesses one."

—Johannes Wolfgang von Goethe

In databases, views are often used as a means for controlling access to sensitive information. By forcing users to access data via a *security view* that only exposes public information, data administrators ensure that secrets will not be leaked, even if the users mishandle the data or are malicious. Security views are robust, making it impossible for users to expose the data hidden by the view,[1] and they are flexible: since they are implemented as arbitrary programs, they can be used to enforce fine-grained access control policies. However, they are not usually updatable—and for good reason! Propagating updates to views made by untrusted users can, in general, alter the source, including the parts that are hidden by the view.

This is a shame, since there are many applications in which having a mechanism for reliably updating security views would be extremely useful. As an example, consider the Intellipedia system [And04], a collaborative data sharing system based on a Wiki that is used by members of the United States intelligence community. The data stored in Intellipedia is classified at the granularity of whole documents, but many documents actually contain a mixture of highly classified and less-classified data. In order to give users with low clearances access to the portions of documents they have sufficient clearance to see, documents often have to be regraded: i.e., the highly classified parts need to be erased or redacted, leaving behind a residual document—a security view—that can be reclassified at a lower level of clearance. Of course (since it is a Wiki), users of these security views would to be able to make changes—e.g., to correct errors or add new information—and have their modifications be propagated back to the original document.

Unfortunately, the lenses we have described so far do not deal adequately with security issues. The critical issue that they fail to address is that many of the natural ways of propagating view updates back to sources alter the source data in ways that violate expectations about its integrity. For example, in the Intellipedia application, the natural way to propagate the deletion of a section of a regraded document is to delete the corresponding section of the original document. But while doing this faithfully reflects the edit made to the view—formally, it satisfies the PutGet law—it is not necessarily what we want: if the section in the original document contains additional classified data in nested subsections, then

[1] Strictly speaking, the user of the view may still be able to gain some knowledge of the hidden parts of the source using the view [MS07]—i.e., security views do not provide privacy—but they do prevent users from directly accessing the data hidden by the view.

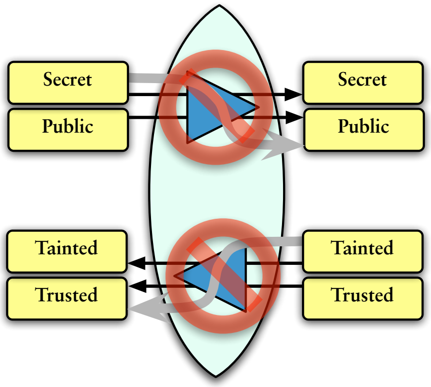


Figure 5.1: Non-interference properties of secure lenses

deleting the section is almost surely unacceptable—users probably should not be able to delete data they do not even have sufficient clearance to see!

It is tempting to require that propagating updates to the view must not lose *any* hidden source data—i.e., require that every lens used to define a security view be very well behaved. But, as discussed in Chapter 2, forcing every lens to obey PutPut is a draconian restriction that rules out transformations such as union and Kleene star that are needed in many applications.

So, because we want to allow untrusted users to modify hidden source data through the view, under certain circumstances, we need a simple, declarative way to specify which parts of the source can be affected by view updates and which parts cannot. Developing a framework in which it is possible to formulate integrity policies like "these sections in the source can be deleted" or "these sections in the view must not be altered (because doing so would have an unacceptable effect on the source)," and verify that lenses obey them, is the goal of this chapter.

Our solution is to define a new semantic space of *secure lenses*, in which types not only describe the sets of structures manipulated by the components of the lens, but also capture the notion that certain parts of the source and view represent trusted data while other parts may be tainted. Semantically, we model these types as sets of structures together with equivalence relations identifying structures that agree on trusted data. Syntactically, we describe them using *security-annotated regular types*—regular expressions decorated with annotations drawn from a set of labels representing static levels of integrity. We formulate a condition ensuring the integrity of source data by stipulating that the *put* function must be *non-interfering*. This ensures that if the update to the view does not change high-integrity data in the view, then the *put* function will not modify high-integrity data in the source.

We then develop security-enhanced versions of our string lens combinators. The typing rules for these combinators use an information-flow analysis to track dependencies between data in the source and view and ensure the new non-interference properties. There are some interesting details compared to information-flow type systems for general-purpose languages, because regular languages describe data schemas at a high level of precision.

Of course, confidentiality is also interesting in the context of security views. Typically the whole reason for defining the view is to hide certain parts of the source. None of the previous work on security views has provided a way to formally and statically verify that the information hidden by the view adheres to a declarative confidentiality policy—the query itself *is* the policy. But, having developed the technical machinery for tracking integrity, it is easy to extend it to track confidentiality as well, and we do so in our information-flow type system. The type system tracks flows of information in two directions, ensuring confidentiality in the forward direction and integrity in the reverse direction—see Figure 5.1.

Tracking information flow using a static type system yields an analysis that is effective but conservative. For example, if the *put* component of a lens ever produces a tainted result, then the type system must classify the source as tainted to ensure the secure lens properties. However, very often there are many inputs that the *put* function can propagate without tainting the source. In the last part of this chapter, we extend secure lenses with mechanisms for detecting these situations. These lenses use a combination of static types and dynamic checks to establish the same essential security properties.

The contributions of this chapter can be summarized as follows:

1. We develop a new semantic space of *secure lenses* that extends our previous work on lenses with a type system ensuring the confidentiality and integrity of data in the source. This provides a reliable framework for building updatable security views.

2. We design the syntax and semantics of *security-annotated regular expressions*, which describe sets of strings as well as equivalence relations that encode confidentiality and integrity policies.

3. We reinterpret our *string lens combinators* as secure lenses.

4. We present an extension to secure lenses that ensures the integrity of source data but replaces some of the static constraints on lens types with dynamic tests.

The rest of this chapter is organized as follows. Section 5.1 introduces an example that illustrates the main challenges that arise when lenses are used to define security views. Section 5.2 defines the semantic space of secure lenses. Section 5.3 introduces security-annotated regular expressions. Section 5.4 presents syntax for secure lenses, with a type system based on an information-flow analysis involving security-annotated regular expressions. Section 5.5 describes an extension to this type system that replaces static checks with dynamic tests. We conclude the chapter in Section 5.6.

5.1 Example

To warm up, let us consider a very small example—simpler than the Intellipedia application discussed in the introduction, but still rich enough to raise the same essential issues. Suppose that the source is an electronic calendar in which certain appointments, indicated by "*", are intended to be private.

```
*08:30 Coffee with Sara (Beauty Shop Cafe)
 10:00 Meeting with Brett (My office)
 12:00 PLClub Seminar (Seminar room)
*15:00 Run (Fairmount Park)
```

Next, suppose that we want to compute a security view where some of the private data is hidden—e.g., perhaps we want to redact the descriptions of the private appointments by rewriting them to `BUSY` and, at the same time, we also want to erase the location of every appointment.

```
08:30 BUSY
10:00 Meeting with Brett
12:00 PLClub Seminar
15:00 BUSY
```

Or, perhaps, we want to go a step further and erase private appointments completely.

```
10:00 Meeting with Brett
12:00 PLClub Seminar
```

In either case, having computed a security view, we might like to allow colleagues make changes to the public version of our calendar to correct errors and make amendments. For example, here the user of the view has corrected a misspelling by replacing "Brett" with "Brent" and added a meeting with Michael at four o'clock.

```
08:30 BUSY
10:00 Meeting with Brent
12:00 PLClub
15:00 BUSY
16:00 Meeting with Michael
```

The *put* function of the redacting lens combines this new view with the original source and produces an updated source that reflects both changes:

```
*08:30 Coffee with Sara (Beauty Shop Cafe)
 10:00 Meeting with Brent (My office)
 12:00 PLClub (Seminar room)
*15:00 Run (Fairmount Park)
 16:00 Meeting with Michael
```

Although this particular update was handled in a reasonable way, in general, propagating view updates can violate expectations about the handling of hidden data in the source. For example, if the user of the view deletes some appointments,

```
08:30 BUSY
10:00 Meeting with Brent
```

then the source will also be truncated (as it must, to satisfy the PutGet law):

```
*08:30 Coffee with Sara (Beauty Shop Cafe)
 10:00 Meeting with Brent (My office)
```

From a certain perspective, this is correct—the updated view was obtained by deleting appointments, and the new source is obtained by deleting the corresponding appointments. But if the owner of the source expects the lens to both hide the private data and maintain the integrity of the hidden data, then it is unacceptable for the user of the view to cause some of the hidden data—the description and location of the three o'clock appointment and the location of the noon appointment—to be discarded.

A similar problem arises when the user of the view replaces a private entry with a public one. Consider a private appointment in the source

```
*15:00 Run (Fairmount Park)
```

which maps via *get* to a view:

```
15:00 BUSY
```

If user of the view replaces it with a public appointment (here, they have insisted that an important event has precedence)

```
15:00 Distinguished Lecture
```

then the description (Run) and location (Fairmount Park) associated with the entry in the original source are both lost.

```
15:00 Distinguished Lecture ()
```

As these examples demonstrate, managing updatable security views reliably requires mechanisms for tracking the integrity of source data.

Let us consider an attractive—but impossible—collection of guarantees we might like to have. Ideally, the *get* function of the lens would hide the descriptions of private appointments as well as the location of every appointment, and the *put* function would take *any* updated view and produce an updated source where all of this hidden data is preserved. Sadly, this is not possible: we either need to allow the possibility that certain updates will cause hidden data to be lost, or, if we insist that this cannot happen, then we need to prevent the user of the view from making those updates—e.g., deleting entries and replacing private entries with public ones—in the first place.

Both alternatives can be expressed using the secure lens framework developed in this chapter. To illustrate these choices precisely, we need a few definitions. The source and view types of the redacting and erasing lenses are formed out of regular expressions that describe timestamps, descriptions, and locations (along with a few predefined regular expressions, NUMBER, COLON, SPACE, etc.) defined as follows:

```
let TIME : regexp =
  NUMBER{2} . COLON . NUMBER{2} . SPACE
let DESC : regexp =
  [^\n()]* - (ANY . BUSY . ANY)
let LOCATION : regexp =
  (SPACE . LPAREN . [^()]* . RPAREN)?
```

To specify the policy that prevents the user from applying updates to the view that would cause hidden data to be lost, we pick a type that marks some of the data as trusted by decorating the bare regular expressions with annotations. Here is a type in which the private appointments are trusted, as indicated by annotations of the form (R : Trusted), but the public appointments are tainted, as indicated by annotations of the form (R: Tainted):

$$
\begin{array}{rl}
 & (\,(\mathtt{SPACE}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE})\colon\mathsf{Tainted}\\
 & \mid(\mathtt{ASTERISK}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE})\colon\mathsf{Trusted})^{*}\\
\overset{🔒}{\Longleftrightarrow} & (\,(\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{NEWLINE})\colon\mathsf{Tainted}\\
 & \mid(\mathtt{TIME}\cdot\mathtt{BUSY}\cdot\mathtt{NEWLINE})\colon\mathsf{Trusted})^{*}
\end{array}
$$

Before the owner of the source data allows the user of the view to propagate their updates back to the source using the *put* function, they check that the original and updated views agree on trusted data. In this case, since the private appointments are trusted, they will refuse to propagate views where the private appointments have been modified. The public appointments, however, may be freely modified.

Alternatively, to specify the policy that provides weaker guarantees about the integrity of source data but allows more updates, we pick a type that labels both public and private appointments as tainted:

$$
\begin{array}{rl}
 & (\,(\mathtt{SPACE}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE})\colon\mathsf{Tainted}\\
 & \mid(\mathtt{ASTERISK}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE})\colon\mathsf{Tainted})*\\
\overset{🔒}{\Longleftrightarrow} & (\,(\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{NEWLINE})\colon\mathsf{Tainted}\\
 & \mid(\mathtt{TIME}\cdot\mathtt{BUSY}\cdot\mathtt{NEWLINE})\colon\mathsf{Tainted})*
\end{array}
$$

The user of the view may update the view however they like—the whole view is tainted—but the lens does not guarantee the integrity of any appointments in the source. The fact that the entire source may be tainted is reflected explicitly in its type.

Here is the Boomerang code that implements these lenses.

```
let public : lens =
  del SPACE .
  copy ( TIME . DESC ) .
  del LOCATION .
  copy NEWLINE

let private : lens =
  del ASTERISK .
  copy TIME .
  ( ( DESC . LOCATION ) <-> "BUSY" ) .
  copy NEWLINE

let redact : lens =
  public* . ( private . public* )*

let erase : lens =
  filter (stype public) (stype private);
  public*
```

Note that there are no security annotations in these programs—the current implementation only tracks basic lens types, leaving security annotations to be checked by hand.[2]

Here is an example of the sort of property we will be able to show using the secure lens framework developed in this chapter:

5.1.1 Lemma: The `redact` lens is a secure lens at the following type:

$$
\begin{array}{rl}
 & (\,(\mathtt{SPACE}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE})\!:\mathsf{Tainted}\\
 & \mid(\mathtt{ASTERISK}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE})\!:\mathsf{Trusted})^{*}\\
\overset{🔒}{\Longleftrightarrow} & (\,(\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{NEWLINE})\!:\mathsf{Tainted}\\
 & \mid(\mathtt{TIME}\cdot\mathtt{BUSY}\cdot\mathtt{NEWLINE})\!:\mathsf{Trusted})^{*}
\end{array}
$$

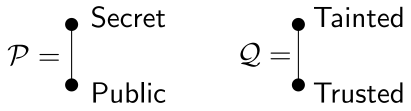
The proof of this property can be found in the appendix.

5.2 Semantics

The basic lens laws ensure some fundamental sanity conditions on the handling of data in the source and view, but, as we saw in the preceding section, to uses lenses reliably in security applications we need additional guarantees. In this section, we describe the refined semantic space of *secure lenses*. These lenses obey new behavioral laws, formulated as *non-interference* conditions, which stipulate that the *put* function must not taint trusted (high integrity) source data and the *get* function must not leak secret (high confidentiality) data—see Figure 5.1.

[2] Also, the *put* functions of the lenses we consider here operate *positionally*—e.g., the *put* function of l^* splits the source and view into substrings and applies *l.put* to pairs of these in order. We defer an investigation of secure resourceful lenses to future work.

Let $\mathcal{P}$ (for "privacy") and $\mathcal{Q}$ (for "quality") be lattices of security labels representing levels of confidentiality and integrity, respectively. To streamline the presentation, we will work with two-point lattices $\mathcal{P} = \{\mathsf{Public}, \mathsf{Secret}\}$ with $\mathsf{Public} \sqsubseteq \mathsf{Secret}$ and $\mathcal{Q} = \{\mathsf{Trusted}, \mathsf{Tainted}\}$ with $\mathsf{Trusted} \sqsubseteq \mathsf{Tainted}$.



Our results generalize to arbitrary finite lattices in a straightforward way. Note that although we think of trusted data as being "high integrity" informally, it is actually the least element in $\mathcal{Q}$. This is standard—intuitively, data that is higher in the lattice needs to be handled more carefully while data that is lower in the lattice can be used more flexibly.

Fix sets S (of sources) and V (of views). To formalize notions like "these two sources contain the same public information (but possibly differ on their private parts)," we will use equivalence relations on S and V indexed by both lattices of security labels. Formally, let $\sim_k^S \subseteq S \times S$ and $\sim_k^V \subseteq V \times V$ be families of equivalence relations indexed by security labels in $\mathcal{P}$, and let $\approx_k^S \subseteq S \times S$ and $\approx_k^V \subseteq V \times V$ be families of equivalence relations indexed by labels in $\mathcal{Q}$. In what follows, when S and V are clear from context, we will suppress the superscripts to lighten the notation. Typically, $\sim_{\mathsf{Secret}}$ and $\approx_{\mathsf{Tainted}}$ will be equality, while $\sim_{\mathsf{Public}}$ and $\approx_{\mathsf{Trusted}}$ will be coarser relations that identify sources and views containing the same public and trusted parts, respectively. These equivalences capture confidentiality and integrity policies for the data.

5.2.1 Definition [Secure Lens]: A *secure lens* l has the same components as a basic lens

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

obeying the following laws for every s in S, v in V, and k in $\mathcal{Q}$ or $\mathcal{P}$ as appropriate:

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

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

$$\frac{v \approx_k l.get\ s}{l.put\ v\ s \approx_k s} \qquad \text{(GetPut)}$$

$$\frac{s \sim_k s'}{l.get\ s \sim_k l.get\ s'} \qquad \text{(GetNoLeak)}$$

We write $S \overset{\text{🔒}}{\Longleftrightarrow} V$ for the set of all secure lenses between S and V.

The PutGet and CreateGet laws here are identical to the basic lens version that we saw in Chapter 2 and express the same fundamental constraints: the *put* and *create* functions must propagate updates to views exactly.

The GetPut law for secure lenses, however, is different. It uses a non-interference property for the *put* function to guarantee the integrity of source data. Formally, it requires that if the original and updated views are related by $\approx_k$, then the original source and the updated source computed by *put* must also be related by $\approx_k$. For example, if the original and updated views are related by $\approx_{\mathsf{Trusted}}$—i.e., they agree on the trusted data—then GETPUT guarantees that the new source will also agree with the original source on trusted data. Note that we recover the basic lens law GETPUT when $\approx_k$ is equality, as it typically is for $\approx_{\mathsf{Tainted}}$.

The GETPUT law suggests a protocol for using secure lenses: before the owner of the source allows the user of a view to invoke the *put* function, they check that the original and updated views are related by $\approx_k$ for every k that is lower in $\mathcal{Q}$ than the data the user is allowed to edit—e.g., in the two-point lattice, a user whose edits are considered tainted would have the checks performed using $\approx_{\mathsf{Trusted}}$. The owner of the source only performs the *put* if the test succeeds.

Secure lenses obey a variant of the PUTPUT law that captures a notion of lenses that are very well behaved on trusted data:

5.2.2 Lemma: Secure lenses admit the following inference rule:

$$\frac{v' \approx_k l.get\ s \approx_k v}{l.put\ v'\ (l.put\ v\ s) \approx_k l.put\ v'\ s} \qquad \text{(PUTPUTTRUSTED)}$$

When $\approx_k$ relates strings that only agree on trusted data (as it typically does for $\approx_{\mathsf{Trusted}}$) then PUTPUTTRUSTED implies that *put* must preserve the trusted hidden data in the source. This law allows operators such as conditional and iteration whose *put* functions do sometimes discard hidden source data in the reverse direction, and are therefore not very well behaved lenses in the strict sense, as long as they indicate that they do so in their type by marking the source data that might be discarded as potentially tainted.

Our main concern in this chapter is preserving integrity after updates, but it is worth noticing that we can also tell an improved story about confidentiality. In previous work on (non-updatable) security views, the confidentiality policy enforced by the view is not stated explicitly—the private information in the source is simply "whatever information is projected away in the view." Our security lenses, on the other hand, have an explicit representation of confidentiality policies, embodied in the choice of equivalence relations. Thus, we can add the GETNOLEAK law stipulating that the *get* function must not leak confidential source information source. This law is formulated as a non-interference condition stating that, if two sources are related by $\sim_k$, then the results computed by *get* must also be related by $\sim_k$. For example, when $\sim_{\mathsf{Public}}$ relates two sources, GETNOLEAK ensures that the views computed from those sources also agree on public data. Thus, secure lenses provide a confidentiality guarantee that can be understood without having to look at the lens program.[3] In the next section, we present a declarative language for security annotations that can be used to describe many such equivalences.

5.3 Security-Annotated Regular Expressions

We will describe the types of our secure string lens combinators using regular expressions decorated with labels drawn from the two lattices of security labels. In this section, we define the syntax and semantics of these security-annotated regular expressions.

Let $\mathcal{K} = (K, \sqsubseteq)$ be a finite lattice. To streamline the notation, we will describe annotations from just *one* lattice of labels. Later, when we use these annotated regular expressions to denote the types of secure string lenses, we will decorate them with labels from both $\mathcal{P}$ and $\mathcal{Q}$. When we calculate the semantics of a type—in particular, the equivalence relations it denotes—we will consider each lattice separately, ignoring the labels in the other lattice.

[3] We treat confidentiality and integrity as orthogonal—almost, see Section 5.5—so users can also choose $\sim^S_{\mathsf{Public}}$ to be equality and our laws place no constraints on confidentiality. This yields the same story as in previous systems, where "what the view hides" is read off from the view definition.

5.3.1 Definition [Security-Annotated Regular Expression]: The set of *security-annotated regular expressions* over Σ and $\mathcal{K}$ is the smallest set generated by the following grammar

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

where $u \in \Sigma^*$ and $k \in K$.

Every security-annotated expression can be interpreted in two ways:

- As a regular language $\mathcal{L}(R)$, defined in the usual way (after ignoring annotations).

- As a family of equivalence relations $\sim_k \subseteq \mathcal{L}(R) \times \mathcal{L}(R)$ capturing the intuitive notion that two strings differing only in high-security regions cannot be distinguished by a low-security observer.

To lighten the notation, when it is clear from context we will often conflate R and $\mathcal{L}(R)$—e.g., we will write $u \in R$ instead of $u \in \mathcal{L}(R)$.

In many languages with security-annotated types, the type structure of the language is relatively simple and so the definition of the "observability relations" is straightforward. However, annotated regular expressions have features like non-disjoint unions that make the intended semantics less obvious—indeed, there seem to be several reasonable alternatives. We describe here a simple semantics based on a notion of erasing inaccessible substrings that we find natural. We discuss alternatives toward the end of the section.

Formally, we define the equivalence relations using a function that erases substrings that are inaccessible to a k-observer, and we take a pair of strings to be equivalent if their erased versions are identical. For ease of exposition, we will describe the erasing function as the composition of two functions: one that marks the inaccessible regions of a string and another that erases marked regions. Let # be a fresh symbol and define $hash(R)$ to be the function that transforms strings in $\mathcal{L}(R)$ by rewriting every character to #,

$$hash(R)(u) \triangleq \underbrace{\# \cdots \#}_{|u| \text{ times}}$$

Let $mark(R, k)$ be the relation that marks the regions of strings that are inaccessible to a k-observer by obscuring them with #:

$$\begin{aligned}
mark(\emptyset, k) &\triangleq \{\} \\
mark(u, k) &\triangleq \{(u, u)\} \\
mark(R_1{\cdot}R_2, k) &\triangleq mark(R_1, k){\cdot}mark(R_2, k) \\
mark(R_1 \mid R_2, k) &\triangleq mark(R_1, k) \;\&\; (\mathcal{L}(R_1) - \mathcal{L}(R_2)) \\
&\quad \cup\; mark(R_2, k) \;\&\; (\mathcal{L}(R_2) - \mathcal{L}(R_1)) \\
&\quad \cup\; mark(R_1, k) \;\&\; mark(R_2, k) \\
mark(R_1{}^*, k) &\triangleq mark(R_1, k)^* \\
mark(R_1{:}j, k) &\triangleq \begin{cases} mark(R_1, k) & \text{if } k \sqsupseteq j \\ hash(R_1) & \text{otherwise} \end{cases}
\end{aligned}$$

The definition of $mark$ uses the operations of union, concatenation, and iteration, which we lift to relations in the obvious way. The most interesting case is for union. In general, the languages denoted by a pair of annotated regular expressions can overlap, so we need to specify how to mark strings that are described by both expressions as well as strings that are only described by one of the expressions. There are three cases: To handle the strings described by only one of the expressions, we use an intersection operator that restricts a marking relation Q to a regular language L:

$$Q \mathbin{\&} L \triangleq \{(u,v) \mid (u,v) \in Q \wedge u \in L\}$$

To handle strings described by both expressions, we use an intersection operator that merges markings

$$Q_1 \mathbin{\&} Q_2 \triangleq \{(u, \mathit{merge}(v_1, v_2)) \mid (u, v_i) \in Q_i\},$$

where:

$$\begin{array}{rcl} \mathit{merge}(\epsilon,\epsilon) & = & \epsilon \\ \mathit{merge}(\#{\cdot}v_1, \_{\cdot}v_2) & = & \#{\cdot}\mathit{merge}(v_1,v_2) \\ \mathit{merge}(\_{\cdot}v_1, \#{\cdot}v_2) & = & \#{\cdot}\mathit{merge}(v_1,v_2) \\ \mathit{merge}(c{\cdot}v_1, c{\cdot}v_2) & = & c{\cdot}\mathit{merge}(v_1,v_2). \end{array}$$

The effect is that characters marked by either relation are marked in the result.

Although *mark* is a relation in general, we are actually interested in cases where it is a function. Unfortunately, the operations of concatenation and iteration used in the definition of *mark* do not yield a function in general due to ambiguity. We therefore impose the following condition:

5.3.2 Definition [Well-Formed Security-Annotated Regular Expression]: R is a *well formed* expression if and only if each subexpression of the form $R_1{\cdot}R_2$ is unambiguously concatenable ($\mathcal{L}(R_1){\cdot^!}\mathcal{L}(R_2)$) and each subexpression of the form R^* is unambiguously iterable ($\mathcal{L}(R)^{!*}$).

5.3.3 Proposition: If R is well formed, then $\mathit{mark}(R,k)$ is a function.

In what follows, we will tacitly assume that all annotated expressions under discussion are well formed. When we define typing rules for secure lens primitives, we will be careful to ensure well-formedness.

Let *erase* be the function on $(\Sigma \cup \{\#\})$ that copies characters in Σ and erases $\#$ symbols. Define $\sim_k$ as the relation induced by marking and then erasing:

$$\begin{array}{rcl} \mathit{hide}_k(u) & \triangleq & \mathit{erase}(\mathit{mark}(R,k)(u)) \\ \sim_k & \triangleq & \{(u,v) \mid \mathit{hide}_k(u) = \mathit{hide}_k(v)\} \end{array}$$

It is easy to see that $\sim_k$ is an equivalence relation.

5.3.4 Lemma: Let R_1 and R_2 be well-formed annotated regular expressions over a finite lattice $\mathcal{K}$. It is decidable whether R_1 and R_2 are equivalent.

Proof sketch: Deciding equivalence for regular languages is straightforward. Moreover, each relation $\sim_k$ is induced by $\mathit{hide}_k(\cdot)$, which is definable as a rational function—a class for which equivalence is decidable [Ber79]. □

To illustrate the semantics, consider the lattice $(\{\mathsf{Public}, \mathsf{Secret}\}, \sqsubseteq)$ with $\mathsf{Public} \sqsubseteq \mathsf{Secret}$, and take R_1 to be the annotated expression $\texttt{[a-z]} : \mathsf{Secret}$. Then for every string u in $\mathcal{L}(R_1)$ we have $\mathit{mark}(R_1, \mathsf{Public})(u) = \#$, and so $\mathit{hide}_{\mathsf{Public}}(u) = \epsilon$, and $\sim_{\mathsf{Public}}$ is the total relation. For the annotated relation $R_1{}^*$, the equivalence $\sim_{\mathsf{Public}}$ is also total because $\mathit{mark}(R_1{}^*, \mathsf{Public})$ maps every u in $\mathcal{L}(R_1{}^*)$ to a sequence of $\#$ symbols and so $\mathit{hide}_{\mathsf{Public}}(u) = \epsilon$. More interestingly, for R_2 defined as

$$(\texttt{[a-z]}{:}\mathsf{Public}){\cdot}(\texttt{[0-4]}{:}\mathsf{Secret}) \mid (\texttt{[a-z]}{:}\mathsf{Public}){\cdot}(\texttt{[5-9]}{:}\mathsf{Secret}),$$

and any string $c{\cdot}n$ in $\mathcal{L}(R_2)$ we have:

$$\begin{array}{rcl} \mathit{mark}(R_2, \mathsf{Public})(c{\cdot}n) & = & c\# \\ \mathit{hide}_{\mathsf{Public}}(c{\cdot}n) & = & c \end{array}$$

It follows that $(c{\cdot}n) \ \sim_{\mathsf{Public}} \ (d{\cdot}m)$ if and only if $c = d$. Finally, for $R_2{}^*$ the equivalence $\sim_{\mathsf{Public}}$ identifies $(c_1{\cdot}n_1 \cdots c_i{\cdot}n_i)$ and $(d_1{\cdot}m_1 \cdots d_j{\cdot}m_j)$ if and only if $i = j$ and $c_i = d_i$ for i from 1 to n.

As we remarked above, there are other reasonable ways to define $\sim_k$. For example, instead of marking and erasing, we could instead compose *mark* with a function that compresses sequences of # symbols into a single #. The equivalence induced by this function would allow low-security observers to determine the presence and location of high-security data, but would obscure its content. We could even take the equivalence induced by the *mark* function itself! This semantics would reveal the presence, location, and length of high-security data to low-security observers. There may well be scenarios where one of these alternative semantics more accurately models the capabilities of low-security observers. For simplicity, we will use the erasing semantics in this chapter.

5.4 Syntax

Having identified the semantic space of secure lenses and defined security-annotated regular expressions, we now turn to syntax, developing secure versions of our string lens combinators. The functional components of these secure lenses are identical to the basic lens versions defined in Chapter 2 (so we elide them), but their typing rules are enhanced with an information-flow analysis that guarantees the secure lens laws.

Copy The simplest lens, *copy* E, takes a well-formed annotated regular expression as an argument. It copies strings belonging to E in both directions.

$$\boxed{\dfrac{E \text{ well-formed}}{\textit{copy}\ E \in E \overset{🔒}{\Longleftrightarrow} E}}$$

5.4.1 Lemma: Let $E \in \mathcal{R}$ be a well-formed security-annotated regular expression. Then *copy* E is a secure lens in $E \overset{🔒}{\Longleftrightarrow} E$.

The proof that *copy* E obeys the secure lens laws is straightforward because the equivalence relations on the source and view are identical.

Constant The *const* lens takes as arguments two well-formed annotated regular expressions E and F, with F a singleton. It maps every source string in E to u, the unique element of F, in the *get* direction, and restores the discarded source string in the reverse direction.

$$\boxed{\dfrac{E, F \text{ well-formed} \qquad F = \{u\}}{\textit{const}\ E\ F \in E \overset{🔒}{\Longleftrightarrow} F}}$$

5.4.2 Lemma: Let E and F be well-formed security-annotated regular expressions such that $F = \{u\}$ for some string u. Then *const* E F is a secure lens in $E \overset{🔒}{\Longleftrightarrow} F$.

Typically F will just be the bare string u, but occasionally it will be useful to decorate it with integrity labels (e.g., see the examples involving the union combinator below). The typing rule for *const* places no additional labels on the source and view types. This is safe: the *get* function maps every string in E to u, so GETNOLEAK holds trivially. The *put* restores the source exactly—including any high-integrity data—so GETPUT also holds trivially.

Union The union combinator uses some new notation, which is explained below.

$$\dfrac{\begin{array}{c} S_1 \cap S_2 = \emptyset \\ l_1 \in S_1 \overset{🔒}{\Longleftrightarrow} V_1 \\ l_2 \in S_2 \overset{🔒}{\Longleftrightarrow} V_2 \\ q = \bigvee\{k \mid k \text{ min obs. } V_1 \neq V_2 \wedge V_1 \;\&\; V_2 \text{ agree}\} \\ p = \bigvee\{k \mid k \text{ min obs. } S_1 \cap S_2 = \emptyset\} \end{array}}{l_1 \,|\, l_2 \in (S_1 \,|\, S_2){:}\,q \overset{🔒}{\Longleftrightarrow} (V_1 \,|\, V_2){:}\,p}$$

5.4.3 Lemma: Let $l_1 \in S_1 \overset{🔒}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{🔒}{\Longleftrightarrow} V_2$ be secure lenses such that $S_1 \cap S_2 = \emptyset$. Then $l_1 \,|\, l_2$ is a secure lens in $(S_1 \,|\, S_2){:}\,q \overset{🔒}{\Longleftrightarrow} (V_1 \,|\, V_2){:}\,p$ where the label q is $\bigvee\{k \mid k \text{ min obs. } V_1 \neq V_2 \wedge V_1 \;\&\; V_2 \text{ agree}\}$ and the label p is $\bigvee\{k \mid k \text{ min obs. } S_1 \cap S_2 = \emptyset\}$.

Like the basic lens version, the union lens uses a membership test on the source string to select a lens in the *get* direction. As is usual with conditionals, the typing rule for union needs to be designed carefully to take implicit flows of confidential information into account. As an example illustrating why, consider the union of the following two lenses:

$$\begin{array}{lcll} l_1 & \triangleq & \texttt{[0-4]}{:}\mathsf{Secret} \leftrightarrow \texttt{A} & \in \texttt{[0-4]}{:}\mathsf{Secret} \overset{🔒}{\Longleftrightarrow} \texttt{A} \\ l_2 & \triangleq & \texttt{[5-9]}{:}\mathsf{Secret} \leftrightarrow \texttt{B} & \in \texttt{[5-9]}{:}\mathsf{Secret} \overset{🔒}{\Longleftrightarrow} \texttt{B} \end{array}$$

We might be tempted to assign it the type obtained by taking the unions of the source and view types of the smaller lenses:

$$l_1 \,|\, l_2 \in (\texttt{[0-4]}{:}\mathsf{Secret} \,|\, \texttt{[5-9]}{:}\mathsf{Secret}) \overset{🔒}{\Longleftrightarrow} (\texttt{A} \,|\, \texttt{B})$$

But this would be wrong: the *get* function leaks information about which branch was selected, as demonstrated by the following counterexample to GETNOLEAK. By the semantics of annotated regular expressions, we have $\mathsf{0} \sim_{\mathsf{Public}} \mathsf{5}$, since $\mathit{hide}_{\mathsf{Public}}$ maps both 0 and 5 to the empty string. But:

$$(l_1 \,|\, l_2).\mathit{get}\ \mathsf{0} = \texttt{A} \not\sim_{\mathsf{Public}} \texttt{B} = (l_1 \,|\, l_2).\mathit{get}\ \mathsf{5}$$

Most languages with information-flow type systems deal with these implicit flows by raising the security level of the result. Formally, they escalate the label on the type of the result by joining it with the label of the data used in the conditional test. Our typing rule for the union lens is based on this idea, although the computation of the label is somewhat complicated because the conditional test is membership in S_1 or S_2, so "the label of the data used in the conditional test" is the least label that can distinguish strings in S_1 from those in S_2. Returning to our example with $(l_1 \,|\, l_2)$ and the two-point lattice, Secret is the only such label, so we label the entire view as secret.

For annotated regular expressions, we can decide whether a given label distinguishes strings in S_1 from those in S_2, and so we can compute the least such label as $\mathcal{P}$ is finite. Let k be a label in $\mathcal{P}$. We say that k *observes* $S_1 \cap S_2 = \emptyset$ if and only if for every string $s_1 \in S_1$ and $s_2 \in S_2$ we have $s_1 \not\sim_k s_2$. Note that k observes $S_1 \cap S_2 = \emptyset$ if and only if the codomains of the rational function $\mathit{hide}_k(\cdot)$ for S_1

and S_2 are disjoint. As the codomain of a rational function is computable and a regular language, we can decide whether k observes the disjointness of S_1 and S_2. In a general lattice there may be several labels that observe the disjointness of S_1 and S_2. The label p we compute for the view type is the join of the set of minimal labels that observe their disjointness.

In the *put* direction, the union lens selects a lens using membership tests on the view and the source. Here we need to consider the integrity of the source data, since modifying the view can result in l_2 being used for the *put* function even though l_1's *get* function was used to generate the original view, or vice versa. To safely handle these situations, we need to treat the source string as more tainted. For example, consider the union of:

$$\begin{aligned} l_1 &\triangleq (\mathit{del}\ \texttt{[0-4]}\!:\!\mathsf{Trusted})\cdot(\mathit{copy}\ \texttt{[A-Q]}\!:\!\mathsf{Tainted}) \\ &\quad \in (\texttt{[0-4]}\!:\!\mathsf{Trusted}\cdot\texttt{[A-Q]}\!:\!\mathsf{Tainted}) \overset{\text{🔒}}{\Longleftrightarrow} (\texttt{[A-Q]}\!:\!\mathsf{Tainted}) \\ l_2 &\triangleq (\mathit{del}\ \texttt{[5-9]}\!:\!\mathsf{Trusted})\cdot(\mathit{copy}\ \texttt{[F-Z]}\!:\!\mathsf{Tainted}) \\ &\quad \in (\texttt{[5-9]}\!:\!\mathsf{Trusted}\cdot\texttt{[F-Z]}\!:\!\mathsf{Tainted}) \overset{\text{🔒}}{\Longleftrightarrow} (\texttt{[F-Z]}\!:\!\mathsf{Tainted}) \end{aligned}$$

This lens does not have secure lens type obtained by taking the union of the source and view types

$$\begin{aligned} &(\texttt{[0-4]}\!:\!\mathsf{Trusted}\cdot\texttt{[A-Q]}\!:\!\mathsf{Tainted}) \mid (\texttt{[5-9]}\!:\!\mathsf{Trusted}\cdot\texttt{[F-Z]}\!:\!\mathsf{Tainted}) \\ \overset{\text{🔒}}{\Longleftrightarrow}\ &(\texttt{[A-Q]}\!:\!\mathsf{Tainted} \mid \texttt{[F-Z]}\!:\!\mathsf{Tainted}) \end{aligned}$$

because the *put* function sometimes fails to maintain the integrity of the number in the source, as demonstrated by the following counterexample to GetPut. By the semantics of annotated regular expressions, we have $\texttt{Z} \approx_{\mathsf{Trusted}} \texttt{A}$, since $\mathit{hide}_{\mathsf{Trusted}}$ maps both to the empty string. But

$$(l_1 \mid l_2).\mathit{put}\ \texttt{Z}\ \texttt{0A} = \texttt{5Z}\ \not\approx_{\mathsf{Trusted}} \texttt{0A}$$

To obtain a sound typing rule for union, we need to raise the integrity label on the source—i.e., consider the source more tainted. We do this by annotating the source type with the least label q such that we can transform a string belonging to $V_1 - V_2$ to a string belonging to V_2 (or vice versa) by modifying q-tainted data.

Formally, we compute q as the join of the minimal set of labels in $\mathcal{Q}$ that observe that V_1 and V_2 are not identical—e.g., for the lens above, Tainted. To ensure that $v \approx_k^{(S_1 \mid S_2)} (l_1 \mid l_2).\mathit{get}\ s$ implies $v \approx_k^{S_1} l_1.\mathit{get}\ s$ for every v in V_1 and s in S_1, and similarly l_2, we also require that q observe that V_1 and V_2 denote the same equivalence relations on strings in their intersection; we write this condition as "V_1 & V_2 agree." Both of these properties can be decided for annotated regular expressions using elementary constructions.

An important special case arises when V_1 and V_2 coincide. Since both lenses are capable of handling the entire view type in this case, the same lens will always selected for *put* as was selected for *get*. For example, the union of

$$\begin{aligned} l_1 &\triangleq (\mathit{del}\ \texttt{[0-4]}\!:\!\mathsf{Trusted})\cdot(\mathit{copy}\ \texttt{[A-Z]}\!:\!\mathsf{Tainted}) \\ &\quad \in (\texttt{[0-4]}\!:\!\mathsf{Trusted}\cdot\texttt{[A-Z]}\!:\!\mathsf{Tainted}) \overset{\text{🔒}}{\Longleftrightarrow} (\texttt{[A-Z]}\!:\!\mathsf{Tainted}) \\ l_2 &\triangleq (\mathit{del}\ \texttt{[5-9]}\!:\!\mathsf{Trusted})\cdot(\mathit{copy}\ \texttt{[A-Z]}\!:\!\mathsf{Tainted}) \\ &\quad \in (\texttt{[5-9]}\!:\!\mathsf{Trusted}\cdot\texttt{[A-Z]}\!:\!\mathsf{Tainted}) \overset{\text{🔒}}{\Longleftrightarrow} (\texttt{[A-Z]}\!:\!\mathsf{Tainted}) \end{aligned}$$

does have the type:

$$\begin{aligned} &(\texttt{[0-4]}\!:\!\mathsf{Trusted}\cdot\texttt{[A-Z]}\!:\!\mathsf{Tainted}) \mid (\texttt{[5-9]}\!:\!\mathsf{Trusted}\cdot\texttt{[A-Z]}\!:\!\mathsf{Tainted}) \\ \overset{\text{🔒}}{\Longleftrightarrow}\ &\texttt{[A-Z]}\!:\!\mathsf{Tainted} \end{aligned}$$

Our typing rule captures this case: if $V_1 = V_2$ then q is the join of the empty set, which is the minimal element Trusted. Annotating with Trusted, the least element in $\mathcal{Q}$, is semantically equivalent to having no annotation at all.

Concatenation Perhaps surprisingly, the concatenation operator also has an interesting typing rule as a secure lens.

$$\boxed{\dfrac{\begin{array}{c} l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1 \qquad S_1 \cdot^! S_2 \\ l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2 \qquad V_1 \cdot^! V_2 \\ q = \bigvee\{k \mid k \text{ min obs. } V_1 \cdot^! V_2\} \\ p = \bigvee\{k \mid k \text{ min obs. } S_1 \cdot^! S_2\} \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2){:}\, q \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \cdot V_2){:}\, p}}$$

5.4.4 Lemma: Let $l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2$ be secure lenses such that $S_1 \cdot^! S_2$ and $V_1 \cdot^! V_2$. Then $(l_1 \cdot l_2)$ is a secure lens in $(S_1 \cdot S_2){:}\, q \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \cdot V_2){:}\, p$ where the label q is $\bigvee\{k \mid k \text{ min obs. } V_1 \cdot^! V_2\}$ and the label p is $\bigvee\{k \mid k \text{ min obs. } S_1 \cdot^! S_2\}$.

As with the union lens, the typing rule for concatenation also needs to be designed carefully to take implicit flows of information into account. Here the implicit flows stem from the way that the concatenation operator splits strings. As an example, consider a lens l_1 that maps `a0` to `A` and `a1` to `a`, and a lens l_2 that maps `b0` to `B` and `b1` to `b`, where all of the source data is private except for the `1`, which is public:

$$\begin{aligned} l_1 &\triangleq ((\mathsf{a}{:}\,\mathsf{Secret}) \cdot (\mathsf{1}{:}\,\mathsf{Public}) \leftrightarrow \mathtt{A}) \mid ((\mathsf{a}{:}\,\mathsf{Secret}) \cdot (\mathsf{0}{:}\,\mathsf{Secret}) \leftrightarrow \mathtt{a}) \\ &\quad \in (\mathsf{a}{:}\,\mathsf{Secret} \cdot (\mathsf{0}{:}\,\mathsf{Secret} \mid \mathsf{1}{:}\,\mathsf{Public})) \overset{\text{🔒}}{\Longleftrightarrow} (\mathtt{A} \mid \mathtt{a}) \\ l_2 &\triangleq ((\mathsf{b}{:}\,\mathsf{Secret}) \cdot (\mathsf{1}{:}\,\mathsf{Public}) \leftrightarrow \mathtt{B}) \mid ((\mathsf{b}{:}\,\mathsf{Secret}) \cdot (\mathsf{0}{:}\,\mathsf{Secret}) \leftrightarrow \mathtt{b}) \\ &\quad \in (\mathsf{b}{:}\,\mathsf{Secret} \cdot (\mathsf{0}{:}\,\mathsf{Secret} \mid \mathsf{1}{:}\,\mathsf{Public})) \overset{\text{🔒}}{\Longleftrightarrow} (\mathtt{B} \mid \mathtt{b}) \end{aligned}$$

The concatenation of l_1 and l_2 does not have the type obtained by concatenating their source and view types,

$$\begin{array}{cl} & ((\mathsf{a}{:}\,\mathsf{Secret} \cdot (\mathsf{0}{:}\,\mathsf{Secret} \mid \mathsf{1}{:}\,\mathsf{Public})) \cdot (\mathsf{b}{:}\,\mathsf{Secret} \cdot (\mathsf{0}{:}\,\mathsf{Secret} \mid \mathsf{1}{:}\,\mathsf{Public}))) \\ \overset{\text{🔒}}{\Longleftrightarrow} & ((\mathtt{A} \mid \mathtt{a}) \cdot (\mathtt{B} \mid \mathtt{b})) \end{array}$$

because *get* exposes the way that the source string was split. For example, $\mathtt{a1b0} \sim_{\mathsf{Public}} \mathtt{a0b1}$ because $\mathit{hide}_{\mathsf{Public}}(\mathtt{a1b0}) = \mathtt{1} = \mathit{hide}_{\mathsf{Public}}(\mathtt{a0b1})$ but

$$(l_1 \cdot l_2).\mathit{get}\ \mathtt{a1b0} = \mathtt{Ab}\ \not\sim_{\mathsf{Public}} \mathtt{aB} = (l_1 \cdot l_2).\mathit{get}\ \mathtt{a0b1}.$$

As with union, we deal with this implicit flow of information by raising the confidentiality level of the data in the view, annotating the view type with the least label that observes the unambiguous concatenability of the source types. Formally, we say k *observes* $S_1 \cdot^! S_2$ if and only if for every $s_1 \cdot s_2$ and $s_1' \cdot s_2'$ in $S_1 \cdot S_2$ with $s_1 \cdot s_2 \sim_k s_1' \cdot s_2'$ we have $s_1 \sim_k s_1'$ and $s_2 \sim_k s_2'$. We can decide whether a given label observes the unambiguous concatenability of two annotated regular expressions using an elementary construction.

In the reverse direction, the concatenation lens splits the source and view strings in two, applies the *put* components of l_1 and l_2 to the corresponding pieces of each, and concatenates the results. An analogous problem now arises with integrity, so we escalate the label on the source type with the least label that observes the unambiguous concatenability of the view types.

Kleene Star The Kleene star lens is similar to concatenation.

$$\boxed{\dfrac{\begin{array}{c} S^{!*} \qquad V^{!*} \\ l \in S \overset{\text{🔒}}{\Longleftrightarrow} V \\ q = \bigvee \{k \mid k \text{ min obs. } V^{!*}\} \\ p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\} \end{array}}{l^* \in (S^*){:}\, q \overset{\text{🔒}}{\Longleftrightarrow} (V^*){:}\, p}}$$

5.4.5 Lemma: Let $l \in S \overset{\text{🔒}}{\Longleftrightarrow} V$ be a secure lens such that $S^{!*}$ and $V^{!*}$. Then l^* is a secure lens in $(S^*){:}\, q \overset{\text{🔒}}{\Longleftrightarrow} (V^*){:}\, p$ where $q = \bigvee \{k \mid k \text{ min obs. } V^{!*}\}$ and $p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\}$.

As with union and concatenation, we need to escalate the confidentiality label on the view side and the integrity label on the source side. To see why, consider the following lens:

$$l \triangleq \mathtt{A}{:}\,\mathsf{Secret} \leftrightarrow \mathtt{B}{:}\,\mathsf{Public} \in \mathtt{A}{:}\,\mathsf{Secret} \overset{\text{🔒}}{\Longleftrightarrow} \mathtt{B}{:}\,\mathsf{Public}$$

It is not the case that

$$l^* \in (\mathtt{A}{:}\,\mathsf{Secret})^* \overset{\text{🔒}}{\Longleftrightarrow} (\mathtt{B}{:}\,\mathsf{Public})^*,$$

because $\mathtt{AAA} \sim_{\mathsf{Public}} \mathtt{AA}$ but

$$l^*.\mathit{get}\ \mathtt{AAA} = \mathtt{BBB} \not\sim_{\mathsf{Public}} \mathtt{BB} = l^*.\mathit{get}\ \mathtt{AA}.$$

The problem is that *get* leaks the length of the source string, which is secret. Thus, we need to escalate the confidentiality label on the view type by the least label observing the unambiguous iterability of the source type.

Likewise, if we consider integrity, it is not the case that the iteration of

$$l \triangleq \mathtt{[0\text{-}9]}{:}\,\mathsf{Trusted} \leftrightarrow \mathtt{A}{:}\,\mathsf{Tainted} \in \mathtt{[0\text{-}9]}{:}\,\mathsf{Trusted} \overset{\text{🔒}}{\Longleftrightarrow} \mathtt{A}{:}\,\mathsf{Tainted}$$

has type

$$l^* \in (\mathtt{[0\text{-}9]}{:}\,\mathsf{Trusted})^* \overset{\text{🔒}}{\Longleftrightarrow} (\mathtt{A}{:}\,\mathsf{Tainted})^*,$$

as demonstrated by the following counterexample to GetPut:

$$\begin{array}{rl} & \mathtt{A} \approx_{\mathsf{Trusted}} \mathtt{AAA} = l^*.\mathit{get}\ \mathtt{123} \\ \text{but} & l^*.\mathit{put}\ \mathtt{A}\ \mathtt{123} = \mathtt{1} \not\approx_{\mathsf{Trusted}} \mathtt{123}. \end{array}$$

Here the problem is that the update shortens the length of the view, which causes the iteration operator to discard trusted data in the source. Thus, we need to escalate the integrity label by the join of the minimal label that observes the unambiguous iterability of the view type.

Sequential Composition The sequential composition lens has a straightforward type.

$$\boxed{\dfrac{\begin{array}{c} l_1 \in S \overset{\text{🔒}}{\Longleftrightarrow} U \\ l_2 \in U \overset{\text{🔒}}{\Longleftrightarrow} V \end{array}}{l_1;l_2 \in S \overset{\text{🔒}}{\Longleftrightarrow} V}}$$

5.4.6 Lemma: Let $l_1 \in S \overset{\text{🔒}}{\Longleftrightarrow} U$ and $l_2 \in U \overset{\text{🔒}}{\Longleftrightarrow} V$ be secure lenses. Then $l_1;l_2$ is a secure lens in $S \overset{\text{🔒}}{\Longleftrightarrow} V$.

As is usual for composition, the typing rule requires that the view type of the first lens and the source type of the second lens be identical. This is essential for ensuring the secure lens laws.

Filter The secure version of the *filter* lens allows us to hide information in a list of source items.

$$\frac{\begin{array}{c} E, F \text{ well-formed} \qquad E \cap F = \emptyset \qquad (E \,|\, F)^{!*} \\ q = \bigvee \{k \mid k \text{ min obs. } E^{!*}\} \\ p \sqsupseteq \bigvee \{k \mid k \text{ observes } E \cdot^{!} F \text{ and } F \cdot^{!} E\} \end{array}}{\mathit{filter}\ E\ F \in (E{:}\,q \,|\, F{:}\,p)^{*} \overset{🔒}{\Longleftrightarrow} E^{*}}$$

5.4.7 Lemma: Let E and F be well-formed security-annotated regular expressions such that $E \cap F = \emptyset$ and $(E \,|\, F)^{!*}$. Then for every label p such that $p \sqsupseteq \bigvee \{k \mid k \text{ observes } E \cdot^{!} F \text{ and } F \cdot^{!} E\}$ the secure lens *filter* $E\ F$ is in $(E{:}\,q \,|\, F{:}\,p)^{*} \overset{🔒}{\Longleftrightarrow} E^{*}$ where $q = \bigvee \{k \mid k \text{ min obs. } E^{!*}\}$.

The typing rule for *filter* captures the fact that none of the Fs appear in the view, so the Fs in the source can be assigned any confidentiality label that observes the unambiguous concatenability of Es and Fs. Since observers with clearance lower than p cannot distinguish sources that differ only in Fs, it is simple to show GetNoLeak: if two sources are related by $\sim_{\mathsf{Public}}$ then their filterings are related by $\sim_{\mathsf{Public}}$. In the backwards direction, we require that the integrity label on the Es be the join of the set of labels that minimally observe that E is unambiguously iterable.

Subsumption Secure lenses admit a rule of subsumption that allows us to escalate the integrity level on the source and the confidentiality level on the view.

$$\frac{\begin{array}{c} q \in \mathcal{Q} \qquad p \in \mathcal{P} \\ l \in S \overset{🔒}{\Longleftrightarrow} V \end{array}}{l \in S{:}\,q \overset{🔒}{\Longleftrightarrow} V{:}\,p}$$

5.4.8 Lemma: Let $l \in S \overset{🔒}{\Longleftrightarrow} V$ be a secure lens and let q be a label in $\mathcal{Q}$ and p a label in $\mathcal{P}$ be labels. Then l is also a secure lens in $S{:}\,q \overset{🔒}{\Longleftrightarrow} V{:}\,p$.

It may seem silly to escalate labels arbitrarily, but it is occasionally useful—e.g., to make the types agree when forming the sequential composition of two lenses.

5.5 Dynamic Secure Lenses

Using a static analysis to track tainted data is effective but conservative—it forces us to label source data as tainted if the *put* function ever produces a tainted result, even if there are many inputs for which it does not. A different idea is to augment lenses with dynamic tests that check if *put* can preserve the integrity of the trusted data in the source for a *particular* view and source. This makes it possible for lenses to make very fine-grained decisions about which views to accept and which to reject, and lets us assign relaxed types to many of our primitives while still retaining strong integrity guarantees.

At the same time that we extend lenses with dynamic tests, we also address a subtle interaction between confidentiality and integrity that we have ignored thus far in this chapter. In the preceding sections, we have assumed that the confidentiality and integrity annotations are completely orthogonal—the semantics of types treats them as independent, and each behavioral law only mentions a single kind of label. However, the protocol for propagating updates to views, in which the owner of the source data tests whether the original and updated views agree on trusted data, can reveal information—possibly confidential—about the source to the user of the view. In this section, we eliminate the possibility of such leaks by adding a new behavioral law requiring that testing whether a given view can be handled (now using arbitrary dynamic tests) must not leak confidential information. An analogous fix can be made in the purely static type system described in the preceding section by placing extra constraints on the equivalence relations denoted by security-annotated expressions.

Formally, we let $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{Q}$ be a set of *clearances*. A user with clearance (j, k) may access data at confidentiality level j and modify data tainted at integrity level k. We extend lenses with a new function

$$l.\mathit{safe} \in \mathcal{C} \to \mathcal{L}(V) \to \mathcal{L}(S) \to \mathbb{B}$$

that returns $\top$ if and only if a user with clearance (j, k) can safely *put* a particular view and source back together. We replace the hypothesis that $v \approx_k s$ in the GetPut law with *safe*, requiring, for all $(j, k) \in \mathcal{C}$ and $s \in S$ and $v \in V$ that

$$\frac{l.\mathit{safe}\ (j, k)\ v\ s}{l.\mathit{put}\ v\ s \approx_k s} \qquad \text{(GetPut)}$$

and revise the protocol for propagating updates to the view accordingly: before the user of the view invokes *put*, the owner of the source checks that the old and new views are safe for the user's clearance.

The *safe* function, which is an arbitrary function, can reveal information about the source. We therefore add a new law stipulating it must not reveal confidential information, formulated as a non-interference property for every $(j, k) \in \mathcal{C}$, every s and s' in S, and every v and v' in V:

$$\frac{v \sim_j v' \qquad s \sim_j s'}{l.\mathit{safe}\ (j, k)\ v\ s = l.\mathit{safe}\ (j, k)\ v'\ s'} \qquad \text{(SafeNoLeak)}$$

For technical reasons[4] we also need a law stipulating that *put* must be non-interfering:

$$\frac{\begin{array}{cc} l.\mathit{safe}\ (j, k)\ v\ s & s \sim_j s' \\ l.\mathit{safe}\ (j, k)\ v\ s' & v \sim_j v' \end{array}}{l.\mathit{put}\ v\ s \sim_j l.\mathit{put}\ v'\ s'} \qquad \text{(PutNoLeak)}$$

With these refinements, we now present dynamic versions of our secure string lens combinators.

Copy For *copy* the *safe* function checks that the new view and original source agree on k-trusted data.

$$\boxed{\frac{E \text{ well-formed} \qquad \forall\ (j, k) \in \mathcal{C}.\ \sim_j \subseteq \approx_k}{\mathit{copy}\ E \in E \overset{\text{🔒}}{\Longleftrightarrow} E}}$$

$$\boxed{\mathit{safe}\ (j, k)\ v\ s = v \approx_k s}$$

5.5.1 Lemma: Let E be a well-formed security-annotated regular expression such that for every clearance (j, k) in $\mathcal{C}$ we have $\sim_j \subseteq \approx_k$. Then *copy* E is a dynamic secure lens in $E \overset{\text{🔒}}{\Longleftrightarrow} E$.

[4] The *safe* component of the composition operator is defined in terms of the *put* function of one of its sublenses, so we need to know that the *put* function doesn't leak information to prove SafeNoLeak.

To ensure that *safe* does not leak information, we require that $\sim_j$ must refine $\approx_k$ for every $(j, k) \in \mathcal{C}$. This condition captures the interaction between the confidentiality and integrity lattices.

Constant For *const*, the view type is a singleton, so we choose a *safe* function that is always true.

$$\boxed{\dfrac{E, F \text{ well-formed} \qquad F = \{u\}}{\mathit{const}\ E\ F\ d \in E \overset{🔒}{\Longleftrightarrow} F}}$$

$$\boxed{\mathit{safe}\ (j, k)\ v\ s = \top}$$

5.5.2 Lemma: Let E and F be well-formed security-annotated regular expressions such that $F = \{u\}$ for some string u. Then *const* E F is a dynamic secure lens in $E \overset{🔒}{\Longleftrightarrow} F$.

Concatenation For the concatenation lens, we choose a *safe* function that tests if the unique substrings of the source and view are safe for l_1 and l_2. It also checks whether j observes the unambiguous concatenability of the source and view types—this is needed to prove PutNoLeak and SafeNoLeak.

$$\boxed{\dfrac{\begin{array}{cc} l_1 \in S_1 \overset{🔒}{\Longleftrightarrow} V_1 & S_1 \cdot^{!} S_2 \\ l_2 \in S_2 \overset{🔒}{\Longleftrightarrow} V_2 & V_1 \cdot^{!} S_2 \\ \multicolumn{2}{c}{p = \bigvee \{k \mid k \text{ min obs. } S_1 \cdot^{!} S_2\}} \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2) \overset{🔒}{\Longleftrightarrow} (V_1 \cdot V_2){:}p}}$$

$$\boxed{\begin{array}{l} \mathit{safe}\ (j, k)\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) = \\ \quad j \text{ observes } S_1 \cdot^{!} S_2 \text{ and } V_1 \cdot^{!} V_2 \\ \qquad \wedge\, l_1.\mathit{safe}\ (j, k)\ v_1\ s_1 \wedge l_2.safe\ (j, k)\ v_2\ s_2 \end{array}}$$

5.5.3 Lemma: Let $l_1 \in S_1 \overset{🔒}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{🔒}{\Longleftrightarrow} V_2$ be dynamic secure lenses such that $S_1 \cdot^{!} S_2$ and $V_1 \cdot^{!} V_2$. Then $l_1 \cdot l_2$ is a dynamic secure lens in $(S_1 \cdot S_2) \Longleftrightarrow (V_1 \cdot V_2) : p$ where the label p is $\bigvee \{k \mid k \text{ min obs. } S_1 \cdot^{!} S_2\}$.

Union For the union lens, the *safe* function tests whether the source and view can be processed by the same sublens. Additionally, because *safe* can be used to determine whether the source came from S_1 or S_2, it only returns true if j observes their disjointness and if V_1 and V_2 agree in their intersection.

$$\frac{\begin{array}{c} S_1 \cap S_2 = \emptyset \\ l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1 \\ l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2 \\ p = \bigvee\{k \mid k \text{ min obs. } S_1 \cap S_2 = \emptyset\} \end{array}}{l_1 \,|\, l_2 \in (S_1 \,|\, S_2) \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \,|\, V_2){:}\,p}$$

$$\begin{array}{l} \textit{safe}\ (j,k)\ v\ s = \\ \quad j \text{ observes } S_1 \cap S_2 = \emptyset \text{ and } V_1 \mathbin{\&} V_2 \text{ agree} \\ \quad \bigwedge \begin{cases} l_1.\textit{safe}\ (j,k)\ v\ s & \text{if } v \in V_1 \wedge s \in S_1 \\ l_2.\textit{safe}\ (j,k)\ v\ s & \text{if } v \in V_2 \wedge s \in S_2 \\ \bot & \text{otherwise} \end{cases} \end{array}$$

5.5.4 Lemma: Let $l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2$ be dynamic secure lenses such that $(\mathcal{L}(S_1) \cap \mathcal{L}(S_2)) = \emptyset$. Then $l_1 \,|\, l_2$ is a dynamic secure lens in $(S_1 \,|\, S_2) \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \,|\, V_2) : p$ where the label p is $\bigvee\{k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset\}$.

Kleene Star For the Kleene star lens, *safe* checks that the view is the same length as the one generated from the source. Because *safe* can be used to determine the length of the source, we require that j observe the unambiguous concatenability of S and V (which implies that j can distinguish strings of different lengths).

$$\frac{\begin{array}{c} l \in S \overset{\text{🔒}}{\Longleftrightarrow} V \\ p = \bigvee\ \{k \mid k \text{ min obs. } S^{!*}\} \end{array}}{l^* \in S^* \overset{\text{🔒}}{\Longleftrightarrow} (V^*){:}\,p}$$

$$\begin{array}{l} \textit{safe}\ (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) = \\ \quad j \text{ observes } S^{!*} \text{ and } V^{!*} \\ \quad \wedge\, n = m\ \wedge\ l.\textit{safe}\ (j,k)\ v_i\ s_i \text{ for } i \in \{1, \ldots, n\} \end{array}$$

5.5.5 Lemma: Let $l \in S \overset{\text{🔒}}{\Longleftrightarrow} V$ be a dynamic secure lens such that $S^{!*}$ and $V^{!*}$. Then l^* is a dynamic secure lens in $(S^*) \overset{\text{🔒}}{\Longleftrightarrow} (V^*){:}\,p$ where $p = \bigvee\ \{k \mid k \text{ min obs. } S^{!*}\}$.

Sequential Composition For sequential composition, the *safe* function requires the conditions implied by l_1's *safe* function on the intermediate view computed by l_2's *put* on the view and the original source.

$$\frac{\begin{array}{c} l_1 \in S \overset{\text{🔒}}{\Longleftrightarrow} U \\ l_2 \in U \overset{\text{🔒}}{\Longleftrightarrow} V \end{array}}{l_1; l_2 \in S \overset{\text{🔒}}{\Longleftrightarrow} V}$$

$$\textit{safe}\ (j,k)\ s\ v = l_1.\textit{safe}\ (j,k)\ (l_2.\textit{put}\ v\ (l_1.\textit{get}\ s))\ s$$

5.5.6 Lemma: Let $l_1 \in S \overset{🔒}{\Longleftrightarrow} U$ and $l_2 \in U \overset{🔒}{\Longleftrightarrow} V$ be dynamic secure lenses. Then $(l_1;l_2)$ is a dynamic secure lens in $S \overset{🔒}{\Longleftrightarrow} V$.

The composition operator is the reason for PutNoLeak as well as the condition in SafeNoLeak which stipulates that *safe* must be non-interfering in its source and view arguments (rather than just its source argument). We could relax these conditions by only requiring PutNoLeak for lenses used as the second argument to a composition operator and the full version of SafeNoLeak for lenses used as the first argument. This would give us yet more flexibility in designing *safe* functions (at the cost of complicating the type system since we would need to track several different kinds of lens types). We defer this extension to future work.

Filter Finally, the *safe* function for the *filter* lens checks that the new view and filtered source agree on k-trusted data. Additionally, to ensure that *safe* does not leak information about the source, *safe* also checks that j observes the way the way that Es and Fs are split in the source, as well as the unambiguous iterability of E.

$$\frac{\begin{array}{c} E, F \text{ well-formed} \qquad E \cap F = \emptyset \qquad (E\,|\,F)^{!*} \\ p \sqsupseteq \bigvee\{k \mid k \text{ observes } E \cdot^! F \text{ and } F \cdot^! E\} \\ \forall\, (j,k) \in \mathcal{C}.\ \sim_j^E \subseteq \approx_k^E \end{array}}{\textit{filter}\ E\ F \in (E\,|\,F{:}p)^* \overset{🔒}{\Longleftrightarrow} E^*}$$

$$\begin{array}{l} \textit{safe}\ (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) = \\ \quad j \text{ observes } E \cdot^! F \text{ and } F \cdot^! E \\ \quad \wedge\ j \text{ and } k \text{ observe } E^{!*} \\ \quad \wedge\ (v_1 \cdots v_n) \approx_k (\mathsf{str\_filter}\ E\ (s_1 \cdots s_m)) \end{array}$$

5.5.7 Lemma: Let E and F be well-formed security-annotated regular expressions such that $E \cap F = \emptyset$ and $(E\,|\,F)^{!*}$ and for every clearance (j,k) in $\mathcal{C}$ we have $\sim_j^E \subseteq \approx_k^E$. Then for every confidentiality label p such that $p \sqsupseteq \bigvee\{k \mid k \text{ observes } E \cdot^! F \text{ and } F \cdot^! E\}$, the dynamic secure lens *filter* E F is in $(E\,|\,F{:}p)^* \overset{🔒}{\Longleftrightarrow} E^*$.

The revised lens definitions in this section illustrate how dynamic tests can be incorporated into the secure lens framework, providing fine-grained mechanisms for updating security views and relaxed types for many of our secure string lens combinators. However, they represent just one point in a large design space. We can imagine wanting to equip lenses with several different *safe* functions—e.g., some accepting more views but offering weaker guarantees about the integrity of source data, and others that accept fewer views but offer correspondingly stronger guarantees. We plan to investigate the tradeoffs along these axes in future work.

5.6 Summary

Secure lenses provide a powerful mechanism for doing fine-grained data sharing across trust boundaries. The views built using secure lenses are robust, since they make it impossible to disclose source information hidden by the view, and reliable since they come equipped with strong guarantees about integrity.

Chapter 6

Boomerang

"Purely applicative languages are poorly applicable."

—Alan J. Perlis

Our technical development in the preceding chapters has focused exclusively on the syntax and semantics of core set of string lens combinators. These combinators are powerful enough to express a large class of useful transformations but they are not very good tools for programming in the large—writing substantial lens programs only using these low-level operators would be extremely tedious. We don't do this. Instead, we have built a full-blown programming language around our combinators called Boomerang. This chapter highlights some of Boomerang's main features, focusing on high-level syntax, typechecking, and engineering issues.

6.1 Syntax

The most critical needs in a language for writing large lens programs are abstraction facilities—i.e., mechanisms for factoring out common patterns into generic definitions, assigning intuitive names to the various pieces of a large program, and so on. Boomerang is organized as a functional language, so it comes equipped with a rich suite of abstraction mechanisms that make it easy to develop large lens programs.

Boomerang is based on the polymorphic λ-calculus [Gir72, Rey74], extended with subtyping, dependent types, and refinement types. Figure 6.1 presents the core syntax of the language. Since the language is based on a standard formalism, we highlight its main features by example rather than giving a complete description. See [Pie02, Chapter 23] for a textbook description.

A Boomerang program is a functional program over base types `string`, `regexp`, `canonizer`, `lens`, etc. The standard library includes each of our string lens combinators as primitives. To use a lens defined in a Boomerang program to manipulate a string—e.g., to use its *get* function to compute a view—we first evaluate the functional program to obtain a value of type `lens` and then use the appropriate function to transform the string. For example, recall the `xml_elt` helper function, which we used to define the `composer` lens. It takes a string `t` and a lens `l` as arguments and builds a lens that transforms XML elements named `t` using `l` to process the children of the element:

```
let xml_elt (t:string) (l:lens) : lens =
    del WHITESPACE*
  . del ("<" . t . ">")
  . l
```

```
m ::= module x = d*                                   Modules

d ::=                                                 Declarations
        type 'a* x = (| x of t)+                      type
      | let p : t = e                                 let
      | test e = e                                    unit test
      | test e : t                                    type test

t ::=                                                 Types
        forall 'a => t                                universal
      | 'a                                            variable
      | x:t -> t                                      dependent function
      | (x:t where t)                                 refinement
      | t * t                                         product
      | t* X                                          data
      | unit | int | bool | char | string | regexp    base
      | canonizer | lens

e ::=                                                 Expressions
        x                                             variable
      | k                                             constant
      | fun (x:t) : t -> e                            function
      | e e                                           application
      | fun 'a -> e                                   type function
      | e{e}                                          type application
      | let p : t = e in e                            let
      | match e with (| p -> e) +  : t                case
      | e,e                                           pair
      | < (e:)? e >                                   match
      | e o e?                                        operator

p ::= _ | x | k | p,p | X p                           Patterns

k ::= 'c' | "s" | n | b | () | [^?c-c]                Constants

o ::= <-> | | | . | ~ | * | + | ? | {n,n?} | ; | & | -     Operators
      | && | || | < | > | <= | >=
```

Figure 6.1: Core Boomerang Syntax

```
  . del WHITESPACE*
  . del ("</" . t . ">")
```

Having defined xml_elt, we can instantiate it to obtain lenses for processing specific XML elements. For example, the name and dates lenses, defined by

```
let name : lens = xml_elt "name" (copy (ALPHA . " " . ALPHA) )
let dates : lens = xml_elt "dates" (copy (YEAR . "-" . YEAR) )
```

process strings of the form

```
<name>Jean Sibelius</name>
```

and

```
<dates>1865-1957</dates>
```

respectively. Compare this program written using xml_elt to one written only using low-level combinators:

```
let name : lens =
    del WHITESPACE*
  . del "<name>"
  . copy (ALPHA . " " . ALPHA)
  . del WHITESPACE*
  . del "</name>"

let dates : lens =
    del WHITESPACE*
  . del "<dates>"
  . copy (YEAR . "-" . YEAR)
  . del WHITESPACE*
  . del "</dates>"
```

It should be clear that the first program, written using the helper function, is simpler to write, reason about, and maintain. Moreover, we can assign the helper function an intuitive name, which allows the programmer to work at an appropriate level of abstraction—here, in terms of XML elements. To build a lens that handles the name and dates for a composer, the programmer simply writes

```
    name . ins ", " . dates
```

which reflects the essential nature of the transformation, rather than

```
    del WHITESPACE*
  . del "<name>"
  . copy (ALPHA . " " . ALPHA)
  . del WHITESPACE*
  . del "</name>"
  . ins ", "
  . del WHITESPACE*
  . del "<dates>"
  . copy (YEAR . "-" . YEAR)
```

```
    . del WHITESPACE*
    . del "</dates>"
```

which exposes all of the low-level details of the lens and obscures the actual transformation being done to the source.

Boomerang has a number of other features designed to simplify lens development. A simple module system provides a way to separate definitions into distinct namespaces:

```
module M =
  let l : lens = copy [A-Z] . del [0-9]
  let x : string = "A"
  let y : string = "1"
end
```

Built-in unit tests gives programmers an easy way to check the behavior of their program during development. Unit tests also provide useful documentation:

```
test (M.l).get (M.x . M.y) = "A"
```

Unit tests can also be used to check errors:

```
test (M.l).get M.x = error
                     (* type error: M.x not in (stype M.l) *)
```

The language includes user-defined data types, polymorphism, and pattern matching, which make it possible to write lenses parameterized on data structures such as lists. As an example, let us build a generic lens for escaping special characters in strings. To represent the escape codes for a particular format, we will use a list of pairs of characters and strings. The characters represents symbols that need to be escaped and the strings represent the escaped versions. Here is the usual definition of polymorphic lists from the List module:

```
type 'a t = Nil | Cons of 'a * 'a t
```

The escape codes for XML PCDATA are represented by the following structure:

```
let xml_escs : (char * string) List.t =
  #{char * string}[('>',"&gt;"); ('<',"&l;"); ('&',"&amp;")]
```

Note that Boomerang requires that programmers instantiate polymorphic definitions explicitly—the language does not (yet) support type inference (we plan to extend the language with inference in the future, after we better understand some of the other features of the language—see the discussion of dependent and refinement types in the next section). The unescaped function takes a list of escape codes and builds a regular expressions that describes the set of unescaped characters by folding down the list:

```
let unescaped (escs : (char * string) List.t) : regexp =
  List.fold_left{char * string}{regexp}
    (fun (r:regexp) (p:char * string) ->
       r | fst{char}{string} p)
    EMPTY
    escs
```

The lens for escaping a single character is defined by folding down the list of escape codes and, at each step, taking the union of the accumulated lens and the lens that rewrites between the unescaped character and the escaped string:

```
let escape_char (escs : (char * string) List.t) : lens =
  List.fold_left{char * string}{lens}
    (fun (l:lens) (p : char * string) ->
       let from,to : char * string = p in
       from <-> to | l)
    (copy (ANYCHAR - (unescaped escs)))
    escs
```

The initial lens supplied to the fold copies every character that does not need to be escaped. The lens that handles escaping for whole strings rather than single characters, is obtained by iterating `escape_char` using Kleene star:

```
let escape (escs : (char * string) List.t) : lens =
  (escape_char escs)*
```

The `xml_esc` lens is a straightforward instantiation of `escape` with the list of escape codes `xml_esc`:

```
let xml_esc : lens =
  escape xml_escs
```

We can verify that it behaves as expected using a unit test:

```
test xml_esc.get
  "Duke Ellington & His Orchestra" =
  "Duke Ellington &amp; His Orchestra"
```

For comparison, here is how we would have to write the same lens only using string lens combinators:

```
let xml_esc : lens =
  ( '>' <-> "&gt;"
    '<' <-> "&lt;"
    '&' <->  "&amp;"
    copy [^<>&] )*
```

In this case, using the combinators is not too painful. However, the lens only handles escaping for XML PCDATA. If we needed to define a lens for escaping CSV, we would need to write another lens with the same essential structure from scratch

```
let csv_esc : lens =
  ( ',' <-> "\\,"
    '\n' <-> "\\n"
    '\\' <-> "\\\\"
    copy [^,\n\\] )*
```

By parameterizing `escape` on the list of escape codes, we avoid having to repeat the common parts of these definitions.

Most languages describe general-purpose computations. Boomerang is a language specifically designed for manipulating strings. As such, we have equipped it with a number of features aimed at making it easier to describe strings, regular expressions, and operations on strings. For example, rather than handling regular expressions using an external library, as in most general-purpose languages, Boomerang has special syntax for defining regular expressions directly in the language. This lets programmers manipulate regular expressions directly, rather than having to wrap them up as string literals and pass them off to a function from a library. The following unit tests illustrates the syntax for regular expressions in Boomerang:

```
test matches [a-z]* "abc" = true
test matches [^a-z]* "abc" = false
test matches ([^a-z]* | [a-z]{3}) "abc" = true
```

The function `matches` checks if the set of strings denoted by its regular expression argument includes its string argument.

Another important language feature is overloading: we use the same symbols to denote operations such as concatenation, union, Kleene star, difference, etc. on characters, strings, regular expressions, lenses, and so on. The Boomerang typechecker automatically resolves overloaded symbols and selects the appropriate operator. The following unit tests demonstrate several uses of overloading involving the concatenation operator:

```
test 'a' . 'b' : string
test "a" . "b" : string
test [a] . [b] : regexp
test (copy "a" . copy "b") : lens
```

They also illustrate Boomerang's unit tests can be used to check types.

Boomerang recognizes the following subtyping relationships between base types:

$$\texttt{char} <: \texttt{string} <: \texttt{regexp} <: \texttt{lens}$$

Subtyping simplifies many programs—e.g., a character can be used in a context expecting a string, a regular expression, or even as a lens. When the subsumption rule is used during typechecking, the system inserts a run-time coercion to convert the value from one run-time type to the other [BCGS91]. For example, if we concatenate a character with a string,

```
test 'a' . "b" = "ab"
```

the typechecker first inserts a coercion that converts the character to a string, and then performs the actual concatenation operation on strings. That is, Boomerang expands the above code to the following more explicit version:

```
test string_concat (string_of_char 'a') "b" = "ab"
```

More interesting—and more useful—is the coercion from regular expressions to lenses. It uses the copy primitive to construct a lens that copies strings belonging to the regular expression in both direction. This turns out to be quite convenient because omitting copy simplifies many lens programs. The following Boomerang declarations illustrate all of these coercions:

```
test 'a'.get 'a' = "a"
test [^a].get "b" = "b"
test "a"*.get "aaa" = "aaa"
```

We do not treat `lens` as a subtype of `canonizer` even though every lens can be used as a canonizer. This is necessary to ensure that the treatment of subtyping is coherent [BCGS91].

6.2 Typechecking

Boomerang comes equipped with a very expressive type system: in addition to the standard types found in the polymorphic λ-calculus—sums, products, functions, and polymorphism—it includes dependent function types and refinement types. A dependent function type "`x:t1 -> t2`" generalizes the ordinary function type "`t1 -> t2`" by allowing `t2` to depend on the value of the argument supplied for `x`. This feature most useful in combination with refinement types. A refinement type "`(x:t where p)`" constrains values of type `t` by requiring that they satisfy the predicate `p`.

Together, these precise types can be used to express extremely detailed properties of programs. In Boomerang, we use them to encode the typing rules for our lens combinators. For example, here is the type of the concatenation lens as declared in the Boomerang standard library:

```
test lens_concat :
(l1:lens ->
(l2:lens where splittable l1.stype l2.stype
           && splittable l1.vtype l2.vtype) ->
(lens in (l1.stype . l2.stype) <-> (l1.vtype . l2.vtype)))
```

The `splittable` function used in the refinement on `l2` is a binary predicate that tests if two regular expressions are unambiguously concatenable. It states that the source and view types must each be unambiguously concatenable and it guarantees that the source and view types of the lens it constructs are the concatenations of the corresponding types from `l1` and `l2`. Refinement types make it possible to express the requirements on the types of the lenses while dependent function types make it possible for the refinement on `l2` to refer to `l1`. The notation used in the return type, "`lens in S <-> V`", desugars to an ordinary refinement type

```
(l:lens where l.stype = S && l.vtype = V)
```

where `l` is fresh. Another example is the union lens:

```
test lens_union :
(l1:lens ->
(l2:lens where disjoint l1.stype l2.stype) ->
(lens in (l1.stype | l2.stype) <-> (l1.vtype | l2.vtype)))
```

As with the concatenation lens, it uses a refinement type to express a constraint on its arguments—here, that the source types are disjoint. The default lens has the following type:

```
test default :
(l:lens ->
((string in l.vtype) -> (string in l.stype)) ->
(lens in l.stype <-> l.vtype))
```

This type requires that the function map strings belonging to the view type of `l` to strings belonging to the source type of `l`. The notation "`string in R`" desugars to "`(u:string where matches R u)`" where `u` is a fresh variable. The `get`, `put` and `create` functions, which extract the component functions of a lens have the following types:

```
test get :
  (l:lens ->
  (string in l.stype) ->
```

```
      (string in l.vtype))
    test put :
      (l:lens ->
      (string in l.vtype) ->
      (string in l.vtype) ->
      (string in l.stype))
    test create :
      (l:lens ->
      (string in l.vtype) ->
      (string in l.stype))
```

These declarations ensure that the strings supplied as sources and views have the correct type.

An earlier version of Boomerang did not support dependent and refinement types. Instead, the conditions specified in the typing rule for each primitive lens were checked in the native code implementing the primitive. This approach was safe—because we evaluate the functional program before we use the lens, any conditions mentioned in its typing rules were checked before we used the lens. However, as we began to develop larger libraries of lens code, we discovered a problem—checking the conditions in the primitives means that errors are reported late and so programmers have to trace back through the evaluation of the functional program to find the source of the errors. For example, recall the escape_char function defined previously:

```
let escape_char (escs : (char * string) List.t) : lens =
  List.fold_left{char * string}{lens}
    (fun (l:lens) (p : char * string) ->
       let from,to : char * string = p in
       from <-> to | l)
    (copy (ANYCHAR - (unescaped escs)))
    escs
```

It takes an escape character and a list of escape codes and constructs a lens—the union of all of the lenses that handle individual characters—that escapes a single character. Because it is defined using union, there is actually a subtle constraint on the type of escs—the list must not contain repeated characters. If we apply escape_char to a list where a character appears twice

```
test escape_char
  #{char * string}[('<',"&lt;"); ('<',"&lt;")] =
  error
```

the union lens will trigger an error because the source types of the two lenses combined in the last iteration of the fold,

```
'<' <-> "&lt;"
```

and

```
  ( '<' <-> "&lt;" | copy (ANYCHAR - [^<]) )
```

are not disjoint. But the union lens is not a good place to report this error—it requires the programmer to trace through the evaluation of the function to determine the cause of the error. This only becomes worse as programs grow in size. It also breaks modularity—the programmer may need to examine code from other modules to find the cause of a type error.

Dependent and refinement types provide a way for Boomerang programmers to express the precise constraints on programs. Thus, when errors do occur, they can be detected early and blame can be assigned to correct location in the program [FF02]. For example, here is another version of escape_char that uses precise types to express the conditions on its argument:

```
let escape_char
  (escs : (char * string) List.t where
    disjoint_chars
      (List.map{char * string}{char} fst{char}{string} escs))
  : lens =
    List.fold_left{char * string}{lens}
      (fun (l:lens) (p : char * string) ->
         let from,to : char * string = p in
         from <-> to | l)
      (copy (ANYCHAR - (unescaped escs))) escs
```

The predicate disjoint_chars checks if a list of characters is disjoint:

```
let disjoint_chars (cs : char List.t) : bool =
  let _,res : regexp * bool =
    List.fold_left{char}{regexp * bool}
      (fun (p:regexp * bool) (c:char) ->
         let r,b : regexp * bool = p in
         (r|c, b && not (matches r c)))
      (EMPTY,true)
      cs in
  res
```

If we apply this version of escape_char to a list with non-disjoint characters, the error will be detected as soon as we evaluate the application and not when we evaluate the union combinator.

Boomerang's typechecker is implemented in the hybrid style, using contracts [Fla06, WF07, FF02]. A static typechecker uses a coarse analysis to rule out obviously ill-formed programs and inserts dynamic checks to verify detailed constraints expressed by dependent and refinement types. Greenberg, Pierce, and Weirich are currently investigating the foundations of this approach [GPW10].

6.3 Implementation

We have built a full prototype implementation of the Boomerang system. This system includes an interpreter for the surface language, native implementations of the core basic, resourceful, and quotient lens combinators, and generic lens libraries for handling escaping, lists, sorting, and XML. Only the type system for secure lenses is not yet implemented.

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 the concatenation of two languages and the iteration of a single language are unambiguous. We have implemented an efficient regular expression library in OCaml based on Brzozowski derivatives [Brz64, ORT09]. The library makes heavy use of hash consing and memoization to avoid recomputing results and a clever algorithm for deciding ambiguity due to Møller [Mø01].

Almost all of the examples typeset in a typewriter font in this dissertation have been generated from a literate source file and checked against our implementation.

6.4 Augeas

Lenses have recently been adopted in industry. Red Hat Linux, Inc., has developed a tool for managing operating system configuration files called Augeas that is directly based on Boomerang [Lut08]. Lenses are used in Augeas is to map flat configuration files of the kind typically found under the `/etc` directory in Unix systems to simplified tree structures that are easy to manipulate using scripts. The language that Augeas programmers use to write lenses is based on an early version of Boomerang—it uses the same set of string lens combinators, the same surface syntax, and (an early version of) our design for resourceful lenses. It also extends the language with some new combinators for indicating tree structure in the view.

Here is a lens developed by Pinson [Pin08] that build a view over the preference files generated by the APT package management tool. The source files for this lens are blocks of text separated by blank lines where each block is a list of key-value pairs:

```
Explanation: Backport packages have lowest priority
Package: *
Pin: release a=backports
Pin-Priority: 100

Explanation: My packages have highest priority
Package: *
Pin: release l=Raphink, v=3.0
Pin-Priority: 700
```

The view are tree structures (or in this case, sequences of trees) representing the same essential information:

```
{ "1"
 { "Explanation" = "Backport packages have lowest priority" }
 { "Package"     = "*" }
 { "Pin"         = "release"
  { "a" = "backports" } }
 { "Pin-Priority" = "100" } }
{}
{ "2"
 { "Explanation" = "My packages have highest priority" }
 { "Package"     = "*" }
 { "Pin"         = "release"
   { "l" = "Raphink" }
   { "v" = "3.0"     } }
 { "Pin-Priority" = "700" } }
```

In this notation, tree nodes are indicated using curly braces and each node has a label, an optional value, and a sequence of children. For example, the subtree

```
{ "Pin"          = "release"
    { "a" = "backports" } }
```

has the label “`Pin`”, value “`release`”, and one child.

Here is the definition of the lens that computes this view in Augeas:

```
module AptPreferences =
   autoload xfm
   (* helpers *)
   let colon        = del /:[ \t]*/ ": "
   let eol          = del /[ \t]*\n/ "\n"
   let value_to_eol = store /([^ \t\n].*[^ \t\n]|[^ \t\n])/
   let value_to_spc = store /[^, \t\n]+/
   let comma = del /,[ \t]*/ ", "
   let equal = Util.del_str "="
   let spc   = Util.del_ws_spc
   let empty = [ del /[ \t]*\n/ "" ]
   let simple_entry (kw:string) =
     [ key kw . colon . value_to_eol . eol ]
   let key_value (kw:string) =
     [ key kw . equal . value_to_spc ]
   let pin_keys = key_value "a"
                  key_value "c"
                  key_value "l"
                  key_value "o"
                  key_value "v"
   let pin = [ key "Pin" . colon
             . value_to_spc . spc
             . pin_keys
             . ( comma . pin_keys )*
             . eol ]
   let entries = simple_entry "Explanation"
                 simple_entry "Package"
                 simple_entry "Pin-Priority"
                 pin
   let record = [ seq "record" . entries+ ]
   let lns = empty* . ( record . empty )* . record?
   let filter = incl "/etc/apt/preferences"
              . Util.stdexcl
   let xfm = transform lns filter
```

This program uses many of the same primitives as Boomerang—e.g., the `del`, (`.`), (`|`), and (`*`) lenses—as well as some new primitives for building trees in the view. The `store` E primitive matches a string described by E and stores it as the value of the enclosing subtree. The `key` E primitive matches a string described by E and stores it as the label of the enclosing subtree. The `seq` x labels the enclosing subtree with the next value from a counter identified by x. The `[`l`]` primitive builds a tree node. It uses the `key` or `seq` primitive in l (which must be unique) to generate the label, the `store` primitive (which also must be unique) to generate the value, and l to generate the children. For example, the `record` lens, declared above as

```
let record = [ seq "record" . entries+ ]
```

generates a single tree labeled by the current value of the `record` counter and containing a non-empty list of children, each generated by `entries`.

Augeas contributors have developed lenses for a large number of formats including each of the following configuration file formats:

aliases.aug
aptprefs.aug
aptsources.aug
bbhosts.aug
crontab.aug
darkice.aug
dhclient.aug
dnsmasq.aug
dpkg.aug
dput.aug
exports.aug
fstab.aug
logrotate.aug
monit.aug
gdm.aug
group.aug
grub.aug
hosts.aug
inifile.aug
inittab.aug
interfaces.aug
limits.aug
ntp.aug
openvpn.aug
pam.aug
passwd.aug
php.aug
phpvars.aug
postfix_main.aug
postfix_master.aug
puppet.aug
rsyncd.aug
samba.aug
services.aug
shellvars.aug
slapd.aug
soma.aug
spacevars.aug
squid.aug
sshd.aug
sudoers.aug
sysctl.aug
util.aug
vsftpd.aug
webmin.aug
xinetd.aug
xorg.aug
yum.aug

Augeas is also beginning to be used in other projects including the Puppet configuration management tool [Lab09], the Netcf network interface configuration tool [Lut09], and the Squeal tool, which allows users to issue SQL-like queries on the file system [Mal09].

6.5 Grammars

This final section describes an extension to Boomerang for defining lenses using grammars. Boomerang's functional infrastructure and precise type system go a long way toward making high-level lens programming convenient. But ultimately, it still requires that programmers write programs in terms of low-level combinators. For lenses that rearrange the source data in complicated ways, using combinators can be quite tedious—the programmer has to massage the source data into the correct position in the view using operators like *permute*. This section describes a different approach using syntax based on grammars. Our design is inspired by XSugar [BMS08]. Programs are expressed as a pair of intertwined grammars. The system uses the first grammar to parse the source, binding pieces of it to variables, and it uses the second grammar as a template for producing the view.

To illustrate how grammars work, consider the composer lens again. Here it is written as a grammar:

```
let WS : regexp = WHITESPACE*
let composer : lens =
 grammar composer ::=
   WS
   "<composer>" WS
    "<name>" n:(key (ALPHA . " " . ALPHA)) "</name>" WS
    "<dates>" d:(YEAR . "-" . YEAR) "</dates>" WS
    "<nationality>" ALPHA "</nationality>" WS
   "</composer>"
  <->
   n ", " d
end
```

The grammar contains a single production named composer that has one rule. It transforms strings by parsing them according to the pattern on one side of the <-> symbol and pretty printing the resulting parse tree—i.e., the bindings of variables to strings—using the pattern on the other side as a template. For example, in the *get* direction, the left-hand side of `composers` parses the XML source, binding the name of the composer to `n` and the dates to `d`, and builds the view by concatenating `n` and `d` with a comma and space between them. The XML formatting and nationality are discarded as they are not bound to any variables. In the *put* direction, it parses the view using the right-hand side of the grammar and produces the new source by pretty printing the bindings for `n` and `d` using the left-hand side as a template.

We can define a grammar that handles a non-empty list of composers using a recursive production:

```
let composer_list : lens =
  grammar composer_list :: =
      c:< composer > <-> c
    | c:< composer > cs:composer_list <-> c "\n" cs
  end
```

The production has two rules: the first handles lists with a single composer while the second handles lists with more than one composer. Note that the mechanisms of resourceful lenses can be used with grammars—both rules treat each composer as a reorderable chunk. This allows the grammar version of the lens to handle updates to views that involve reorderings, just like the combinator version.

The final grammar describes the composers lens:

```
let composers : lens =
  grammar composers ::=
      "<composers>" WS "</composers>" <-> ""
    | "<composers>" cs:composer_list WS "</composers>" <-> cs
  end
```

It has two rules: one for the empty case and another for the non-empty case.

This lens behaves the same as the version described using combinators. For example, when we apply the *get* function to the original XML source, we get it produces the view

```
Jean Sibelius, 1865-1956
Aaron Copland, 1910-1990
Benjamin Briten, 1913-1976
```

as expected.

To some extent, the choice of whether to program the lens using combinators or grammars is a matter of taste. However, the grammar approach is often simpler when we need to reorder data in going from source to view. For example, suppose that we wanted to swap the order of the name and dates for each composer in the view. As described in Chapter 2, we can do this using the swap lens, but the combinator program becomes much more complicated—we need to place the swap operator carefully in our program to lift the dates over the names:

```
let composer : lens =
  xml_elt "composer"
    ( ( xml_elt "name" (copy (ALPHA . " " . ALPHA) )
      ~ ( xml_elt "dates"  (copy (YEAR . "-" . YEAR) )
        . ins ", " ) )
    . xml_elt "nationality" ( del_default ALPHA "Unknown" ) )
```

This approach becomes complicated as soon as the transformation reorders multiple pieces of information in the source (even if we use the n-ary generalization of swap, *permute*, described in Chapter 2). By contrast, the grammar version of the lens can be easily modified to obtain the behavior we want—we just invert the order of the variables `n` and `d` on the right-hand side of the rule, replacing "`n ", " D`" with "`d ", " n`". Thus, grammars and variables provide a natural way to describe many transformations on strings that reorder information in going from source to view.

Grammar are fully-integrated into the Boomerang system, and can be freely combined with all of the language's other features. They are implemented by a source-to-source translator that maps productions to combinator expressions. Formally, their syntax is given the following extension to the grammar for Boomerang defined at the beginning of this chapter:

$$\begin{array}{rcl}
e & ::= & \ldots \mid \texttt{grammar}\ p\ (\texttt{and}\ p)^*\ \texttt{end} \\
p & ::= & x\ \texttt{::=r\textbackslash}\ \ (\texttt{|}\ r)^* \\
r & ::= & {a_L}^*\ \texttt{<->}\ {a_R}^* \\
a_L & ::= & x\ \texttt{:}\ e \mid e \\
a_R & ::= & x \mid e
\end{array}$$

A grammar is a list of productions; a production p consists of a name x and a set of rules (separated by `|`); a rule r consists of two lists of atoms, one on the left-hand side and another on the right-hand side; an atom a_L on the left is either a bare expression or an expression labeled with a variables; and an atom on the right is either a bare expression or a variable (we do not need to bind variables to expressions on the right because each variable will have a binding on the left). We impose several well-formedness conditions on grammars: First, we require that variables be used linearly—i.e., every variable occurring in a rule must be used exactly once on each side of the rule. Second, we require that grammars be right-recursive. This condition is essential—without it, grammars could be used to describe context-free languages and Boomerang's type system is based on regular languages. It turns out that imposing linearity on variables and right-linearity separately on the left and right-hand sides of rules ensures a kind of joint right-linearity: every well-formed rule has one of two forms

$$\begin{array}{rcll}
a_1 \ldots a_k & \texttt{<->} & b_1 \ldots b_l & \text{or} \\
a_1 \ldots a_k\ x\texttt{:}e_{k+1} & \texttt{<->} & b_1 \ldots b_l\ x &
\end{array}$$

where each of the a_i and b_j atoms are not recursive.

The first step in the compilation is transforming individual rules to lenses. There are two cases. For non-recursive rules, we construct a lens that maps between the left and right-hand sides directly. For example, the rule in the `composer` production compiles to the following lens:

```
permute
  #{int}[1;2;3;4;5]
  #{lens}[ del ( WS . "<composer>" . WS . "<name>" )
         ; key (ALPHA . " " . ALPHA)
         ; "</name>" . WS . "<dates>" <-> COMMA . SPACE
         ; copy (YEAR . "-" . YEAR)
         ; del ( "</dates>" . WS . "<nationality>" . ALPHA .
                "</nationality>" . WS . "</composer>") ]
```

The list of integers represents a permutation. In this case, it is just the identity permutation—the variables `n` and `d` appear in the same order on the left and right-hand sides of the rule. However, more generally, we need to permute the views produced by each lens. We calculate the appropriate permutation by comparing the list of variables mentioned in each rule. In the same way, we compile the non-recursive prefixes of recursive rules—i.e., all but the final atom—and associate the resulting lens with the variable named in the right-most position. The result after compiling each rule, is a right-linear grammar with lenses as non-terminals. For example compiling the `composer_list` production yields the following:

$$\begin{array}{l} \mathit{composer\_list} \\ \quad ::= \texttt{(permute \#\{int\}[1] \#\{lens\}[< composer >])} \\ \qquad \mid \texttt{(permute \#\{int\}[1] \#\{lens\}[< composer >])}\ \mathit{composer\_list} \end{array}$$

To complete the compilation, we eliminate recursion by transforming it into iteration, using a generalization of the standard construction on ordinary grammars. There are again two cases. If $x_i ::= p_1 \ldots p_k$ is the only production, then we partition its rules into two sets: recursive rules go into S_1, and non-recursive rules into S_2. We then construct the following lens for x_i:

$$x_i = \big(|_{(k\ x_i)\in S_1}\ \ k\big)^{*} \cdot \big(|_{l\in S_2}\ \ l\big)$$

It is straightforward to verify that this lens describes the same transformation as r_i. If there are multiple productions, we eliminate one by replacing references to it with a similarly constructed lens, and repeat the compilation.

There is one restriction of the compilation that bears mentioning. The typing rules for our lens combinators check unambiguity *locally*—i.e., for every concatenation and iteration. Our compilation only produces a well-typed lens if the grammar is “locally unambiguous” in this sense.

6.6 Summary

The Boomerang language provides convenient high-level notation for programming with lenses and an expressive type system for establishing correctness. It also includes an extension for describing lenses using grammars instead of combinators. In our experience, these features are critical for developing lens programs of substantial size. We have used them to develop a number of lenses for real-world data formats including electronic address books, calendars, bibliographies, and scientific data. Our design has also been used in industry: the Augeas tool uses a language based directly on Boomerang.

Chapter 7

Related Work

The structures investigated in this dissertation—lenses and their associated behavioral laws—are not completely new. Similar structures have been studied for decades in the database community. Also, programming languages that—in some way—can be run both forwards and backwards have been studied previously. This chapter summarizes previous work on these topics, highlighting the key differences to our work.

Broadly speaking, lenses have several features that distinguish them from previous work on view update translators in databases. One is that lenses transform whole states rather than "update operations". Another is that they treat well-behavedness as a form of type assertion. Lenses are also novel in their treatment of totality—i.e., they guarantee that propagating the update to the view will not fail at run time. On the programming language side, much of the previous related work focuses on reversible languages—i.e., languages where the source and view are isomorphic. The languages we have developed appear to be the first that are based on a formal semantic foundation in which information can be discarded in the forward direction. Lenses are also unique in identifying totality as a primary goal and in emphasizing types—i.e., compositional reasoning about well-behavedness—as an organizing design principle. Lastly, to the best of our knowledge, other work has not tackled the issues related to ignorable, ordered, and confidential data addressed in this dissertation.

7.1 Foundations

The foundations of correct view update were studied extensively by database researchers in the late 1970s and 1980s. This thread of work is closely related to the semantics of lenses. We discuss here the main similarities and differences between our work and these classical approaches to view update—in particular Dayal and Bernstein's notion [DB82] of "correct update translation," Bancilhon and Spyratos's [BS81] notion of "update translation under a constant complement," Gottlob, Paolini, and Zicari's "dynamic views" [GPZ88], and the "triggers" offered by commercial database systems.

The view update problem concerns translating updates on a view into "reasonable" updates on the underlying database. We can break this broad problem statement into several specific questions: First, what is a "reasonable" translation of an update? Second, how should we handle updates for which there is *no* reasonable way of translating its effect to the underlying source? And third, how should we deal updates for which there are *many* reasonable translations to choose from? We consider these questions in order.

One can imagine many possible ways of assigning a precise meaning to "reasonable update translation," but there turns out to be widespread agreement in the literature with most frameworks adopting one of two basic positions. The stricter position is captured in Bancilhon and Spyratos's [BS81] notion of the *complement* of a view, which must include at least the information missing from the view. When the complement is fixed there exists at most one update of the database that reflects a given update on the view while leaving the complement unchanged—i.e., that "translates updates under a constant complement." This approach has influenced numerous later works in the area, including recent papers by Lechtenbörger [Lec03] and Hegner [Heg04].

The other, more permissive, definition of "reasonable" is captured in Gottlob, Paolini, and Zicari's "dynamic views" [GPZ88]. They present a general framework and identify two special cases, one equivalent to Bancilhon and Spyratos's constant complement translators and the other—which they advocate on pragmatic grounds—their own dynamic views. Our notion of lenses adopts the same, more permissive, attitude towards reasonable behavior of update translation. Indeed, modulo some technicalities, the set of well-behaved lenses is isomorphic to the set of dynamic views in the sense of Gottlob, Paolini, and Zicari and the set of very well-behaved lenses is isomorphic to the set of translators under constant complement in the sense of Bancilhon and Spyratos. Dayal and Bernstein's [DB82] seminal paper on "correct update translation" also adopts the more permissive position. Their notion of "exactly performing an update" corresponds to the PutGet law.

The tradeoffs between these two perspectives on reasonable update translations have been further refined by Hegner [Heg90, Heg04], who introduces the term *closed view* for views that can be updated independently, without having to consider the effect on the underlying source, and *open view* for views where the user is explicitly aware that they are updating some data derived from the source. Hegner advocates the closed-world approach, but notes that both choices have advantages in different contexts—e.g., the open-view approach is more permissive, allowing more updates in general, while the closed-world approach provides users with a robust abstraction of the source.

Hegner [Heg04] also formalizes an additional condition (which has also been noted by others—e.g., by [DB82]) called *monotonicity* of update translations. It states that an update that only adds records to the view should be translated just into additions to the database, and that an update that adds more records to the view should be translated to a larger update to the database and similarly for deletions.

The treatment of updatable views in commercial database systems has evolved over time. The SQL-92 standard states that views defined using simple operators over a single base relation are updatable [IA92]. The later SQL:2008 standard gives a more complicated collection of conditions that treats many more views as updatable [IA08]. Systems such as Oracle [FL05, Lor05], Microsoft's SQL Server [Mic05], and IBM's DB2 system [IBM04] typically provide two quite different mechanisms for constructing updatable views. Simple views defined using select, project, and a very restricted form of join (i.e., where the keys of one relation are a subset of those in the other) are considered *inherently updatable*. For these, the notion of reasonableness is essentially the constant complement position. Alternatively, programmers can make an arbitrary view updatable by adding a *relational trigger* that is invoked whenever an update is attempted on the view. These triggers can execute arbitrary code to update the underlying database and the notion of reasonableness is left entirely to the programmer.

The second question posed at the beginning of this section was how to deal with the possibility that some updates cannot be translated back to the source in a reasonable way. The simplest response is to simply let the translation fail in situations where the effect on the source would be unreasonable. The advantage of this approach is that the system can determine reasonableness on a case-by-case basis, allowing translations that usually give reasonable results but that fail under rare conditions. The disadvantage is that it loses the ability to perform offline updates to the view—to test if a given update can be translated, we need the underlying database. Another possible approach is to restrict the set of operations on the view to ones that can always be translated. A third approach—the one adopted in our work—is to restrict the type of the view so that *arbitrary* type-respecting updates are guaranteed to succeed.

The third question posed above was how to deal with the possibility that there may be multiple reasonable translations for a given update. One attractive idea is to somehow restrict the set of reasonable translations so that this possibility does not arise—i.e., so that every translatable update has a unique translation. For example, under the constant complement approach, after fixing the choice of a complement, every update has a unique translation. Hegner's additional condition of monotonicity [Heg04] ensures that (at least for updates consisting of only inserts or only deletes), the translation of an update is unique, independent of the choice of complement. Another possibility is to place an ordering on possible translations of a given update and choose one that is minimal in this ordering. This idea plays a central role, for example, in Johnson, Rosebrugh, and Dampney's account of view update in the Sketch Data Model [JRD01]. However, Buneman, Khanna, and Tan [BKT02] have established a variety of intractability results for the problem of inferring minimal view updates in the relational setting for query languages that include both join and either project or union. This dissertation pursues an entirely different approach: rather than trying to constrain the framework so that updates are determined, we provide programmers with tools for describing the view and the intended update policy together. That is, we design primitives that allow programmers to pick a policy for translating updates.

7.2 Programming Languages

At the level of syntax, many different kinds of bidirectional programming have been explored across a surprisingly diverse range of communities, including programming languages, databases, program transformation, constraint-based user interfaces, software engineering, and quantum computing. One useful way of classifying these languages is by the "shape" of the semantic space in which their transformations live. We identify three major classes:

Bidirectional languages pair a *get* function of type $S \rightarrow V$ with a *put* function of type $V \rightarrow S \rightarrow S$. The *get* function may project away source information; the *put* function restores it.

Bijective languages have *put* functions with the type $V \rightarrow S$—i.e., there is no source argument to refer to. To avoid loss of information, the *get* and *put* functions must form a (perhaps partial) bijection between S and V.

Reversible languages go a step further, demanding that it be possible to undo the work performed by any function by applying the function "in reverse".

In the first class of languages, the work that is fundamentally the most similar to ours is Meertens's treatment of *constraint maintainers* for constraint-based user interfaces [Mee98]. Meertens's semantic setting is actually even more general: the *get* and *put* functions are *relations*, not just functions, and also symmetric: *get* relates pairs from $S \times V$ to elements of V and *put* relates pairs in $V \times S$ to elements of S. The idea is that a constraint maintainer forms a connection between two 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 some desired relationship between the objects is always maintained. Taking the special case where the *get* relation is actually a function (which is important for Meertens because this is the case where composition [in the sense of our " ; " combinator] preserves well behavedness), yields essentially our well behaved lenses.

Meertens proposes a variety of combinators for building constraint maintainers, most of which have analogs among our lenses, but some of his combinators do not support compositional reasoning about well-behavedness. For example, when he considers constraint maintainers for lists, he adopts an operation-based approach, focusing on constraint maintainers that work with structures in terms of the "edit scripts" that might have produced them.

Bidirectional languages capable of duplicating data in the *get* direction have been the focus of recent work by the Programmable Structured Documents group at the University of Tokyo.

Early work by Mu, Hu, and Takeichi on "injective languages" for view-update-based structure editors [MHT04a] adopted a semantic framework similar to lenses. Although their transformations obey our GetPut law, their notion of well-behaved transformations is informed by different goals than ours, leading to a weaker form of the PutGet law. They are concerned with using view-to-view transformations to simultaneously restore invariants *within* the view as well as updating the source. For example, a view may maintain two lists where the name field of each element in one list must match the name field in the corresponding element in the other list. If an element is added to the first list, then the change must not only be propagated to the source, it must also be added to the second list in the view. It is easy to see that PutGet cannot hold in these situations. Like Meertens, they assume that the view is modified by inserting editing tags that mark modified fields as "updated." These marks are used by the *put* function and then discarded—another change to the view that violates PutGet. Consequently, to support invariant preservation within the view, and to support edit tags, their transformations only obey a much weaker variant of PutGet called PutGetPut.

Another paper by Hu, Mu, and Takeichi [HMT08, MHT06] applies a bidirectional programming language quite closely related to ours to the design of "programmable editors" for structured documents. As in their earlier work [MHT04a], they support preservation of local invariants in the *put* direction. Instead of annotating the view with modification marks, they assume that a *put* or a *get* occurs after *every* modification to either view. They use this "only one update" assumption to choose the correct inverse for the lens that copied data in the *get* direction—because only one branch could have been modified. Consequently, they can *put* the data from the modified branch and overwrite the unmodified branch. Here, as in their earlier work, the notion of well behavedness is weakened to PutGetPut.

Yet another line of work by the same group investigated bidirectional languages with variable binding. Languages that allow unrestricted occurrences of variables implicitly support duplication, since data can be copied by programs that use a variable several times. The goal of this work is to develop a bidirectional semantics for XQuery [LHT07]. As in the earlier work, they propose relaxed variants of the lens laws and develop a semantics based on sophisticated propagation of annotated values.

One possible connection between their work and our quotient lenses is an condition proposed in article by Hu, Mu, and Takeichi [HMT08]. This is formulated in terms of an ordering on edited values that captures when one value is "more edited" than another. They propose strengthening the laws to require that composing *put* and *get* produce an abstract structure that is more edited in this sense, calling this property *update preservation*. We hope to investigate the relationship between our quotient-lens PutGet law and their PutGetPut plus update preservation. However, we suspect that the comparison may prove difficult to make, however, because our framework is "state based"—the *put* function only sees the state of the data structure resulting from some set of edits, not the edits themselves—while theirs assumes an "operation-based" world in which the locations and effects of edit operations are explicitly indicated in the data.

Another paper by Matsuda, Hu, Nakano, Hamana, and Takeichi proposes a mechanism for describing lenses using ordinary, unidirectional programs [MHN$^+$07]. It defines a "bidirectionalization" transformation that takes programs written in a restricted λ-calculus and calculates an explicit complement, in the sense of Bancilhon and Spyratos. This provides a way to construct a lenses from a description of its *get* function. There are two key difference between their work and ours. First, although there are many well-behaved lenses with a given *get* function, their construction picks just one lens. Second, their lenses are not total—their *put* function may fail if the user makes a significant change to the view.

Languages for Bijective Transformations

There is an active community of program transformation researchers working on *program inversion* and *inverse computation*—see, for example, Abramov and Glück [AG00, AG02] and the many papers cited there. Program inversion [Dij79] derives an inverse program from a standard program. Inverse computation [McC56] computes a possible input of a program from a particular output. One approach to inverse computation is to design languages that produce easily invertible expressions—for example, languages that can only express injective functions, where every program is trivially invertible. These languages bear some intriguing similarities to ours, but differ in a number of ways, primarily in their focus on the bijective case.

In the database community, Abiteboul, Cluet, and Milo [ACM97] defined a declarative language of *correspondences* between parts of trees in a data forest. In turn, these correspondence rules can be used to translate one tree format into another through non-deterministic Prolog-like computation. This process assumes an isomorphism between the two data formats. The same authors later defined a system for bidirectional transformations based around the concept of *structuring schemas*—i.e., parse grammars annotated with semantic information [ACM98]. Their *get* functions involve parsing, while *put* functions involve unparsing. Again, to avoid ambiguous abstract updates, they restricted themselves to *lossless* grammars that define an isomorphism between concrete and abstract views.

A number of other systems provide some linguistic mechanisms for describing essentially bijective transformations. XSugar [BMS08] is a bidirectional language that targets the case where one structure is represented in XML and the other structure is a string. Transformations in XSugar are specified using pairs of intertwined grammars. Our design for grammars in Boomerang discussed in Chapter 6 was inspired by XSugar. A similar language biXid [KH06] also specifies bidirectional conversions between pairs of XML documents. However, unlike XSugar, biXid allows ambiguous productions and rules that are non-linear in their use of variables.

The PADS system [FG05] generates a data type, parser, and pretty printer for ad hoc data from a declarative description of the format. PADS comes with a rich collection of primitives for handling a wide variety of data including characters, strings, fixed-with integers, floating point values, lists, etc. Recent work on PADS has focused on developing mechanisms for learning data descriptions automatically [FWZW08]. Kennedy's pickling combinators [Ken04] describe serializers and deserializers. Benton [Ben05] and Ramsey [Ram03] each proposed systems for mapping between the values in a host language and the run-time values manipulated by an embedded interpreter.

JT [EG07] synchronizes programs written in different high level languages, such as C and Jekyll, an extension of C with features from ML. JT uses a notion of distance to decide how to propagate changes, allowing the detection of non local edits. The synchronization seems to work well in many cases but there is no claim that the semantics of the synchronized programs are the same.

Wadler's *views* [Wad87], extend algebraic pattern matching to abstract data types. Programmers supply *in* and *out* functions to map between views and the underlying structures they are generated from. In certain simple situations, Wadler observes that the *in* and *out* functions can be defined together.

Languages for Reversible Transformations

Lenses are the first work we are aware of in which totality and compositional reasoning about totality are taken as primary design goals. Nevertheless, in all of the languages discussed above there is an expectation that programmers will want their transformations to be "total enough"—i.e., that the sets of inputs for which the *get* and *put* functions are defined should be large enough for some given purpose. In particular, we expect that *put* functions should accept a suitably large set of views for each source, since the whole point of these languages is to allow editing through a view. A quite different class of languages have been designed to support *reversible* computation, in which the *put* functions are only ever applied to a result of the corresponding *get* functions. While the goals of these languages are quite different from ours—they have nothing to do with view update—there are intriguing similarities in the basic approach.

Landauer [Lan61] observed that non-injective functions are logically irreversible, and that irreversibility requires generating and dissipating some heat per machine cycle. Bennett [Ben73] demonstrated that this irreversibility was actually not inevitable by constructing a *reversible Turing machine*, showing that thermodynamically reversible computers were plausible. Baker [Bak92] argued that irreversible primitives were only part of the problem; irreversibility at the "highest levels" of computer usage cause the most difficulty due to information loss. Consequently, he advocated the design of programs that "conserve information." Because deciding reversibility of large programs is unsolvable, he proposed designing languages that guaranteed that all well-formed programs are reversible, i.e., designing languages whose primitives were reversible and whose combinators preserved reversibility. A considerable body of work has developed around these ideas [MHT04b]. Abramsky recently developed a compositional translation from λ-calculus to a reversible abstract machine model [Abr05].

7.3 Databases

Research on view update translation in the context of databases has tended to focus on taking an existing language for defining *get* functions (e.g., relational algebra) and then considering how to infer corresponding *put* functions—either automatically or with some user assistance. Our approach is to design languages in which the definitions of *get* and *put* go hand-in-hand. We also go beyond classical work in the relational setting by directly transforming and updating arbitrary data structures rather than just relations.

Work by Bohannon, Pierce, and Vaughan [BVP06] extends the lens framework described here to the relational model. Their lenses are based on the primitives of classical relational algebra, with additional annotations that specify the desired "update policy" in the *put* direction. They develop a type system, using record predicates and functional dependencies, to aid compositional reasoning about well-behavedness. The chapter on view update in Date's textbook [Dat03] articulates a similar perspective on translating relational updates.

Masunaga [Mas84] described an automated algorithm for translating updates on views defined by relational algebra. The core idea was to annotate where the "semantic ambiguities" arise, indicating they must be resolved either with knowledge of underlying database semantic constraints or by interactions with the user.

Keller [Kel85] catalogued all possible strategies for handling updates to a select-project-join view and showed in his thesis that these are exactly the set of translations that satisfy a small set of intuitive criteria: no side effects, one-step changes, no unnecessary changes, simplest replacements, and no delete-insert pairs. Keller later proposed allowing users to choose an update translator at view definition time by engaging in an interactive dialog with the system and answering questions about potential sources of ambiguity in update translation [Kel86]. Building on this foundation, Barsalou, Siambela, Keller, and Wiederhold [BSKW91] described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa [MT85] presented a tool for exploring the effects of choosing a view update policy. Their tool shows the update translation for update requests supplied by the user; by considering all possible valid sources, it predicts whether the desired update would in fact be reflected back into the view after applying the translated update to the concrete database.

The Clio tool, developed by a team of researchers including Miller, Hernandez, Haas, Yan, Ho, Fagin, and Popa [MHH^{+}01], manages heterogeneous data transformation and integration. Clio provides a graphical interface for visualizing two schemas and for specifying correspondences between their fields—a schema mapping. Although they focus on *get* functions, if bidirectional mappings are needed, they can be constructed. Recent work by Karvounarakis and Ives proposes bidirectional schema mappings for doing data integration of heterogeneous sources [KI08].

Atzeni and Torlone [AT97, AT96] described a tool for translating views and observed that if it is possible to translate any source view to and from a shared view, then there is a bidirectional transformation between any pair of sources. They only consider mappings where the structures are isomorphic.

Complexity bounds have also been studied for various versions of the view update inference problem. Cosmadakis and Papadimitriou [CP84] considered the view update problem for a single relation, where the view is a projection of the underlying relation, and showed that there are polynomial time algorithms for determining whether insertions, deletions, and tuple replacements to a projection view are translatable into concrete updates. As mentioned above, Buneman, Khanna, and Tan [BKT02] established a variety of intractability results for the problem of inferring "minimal" view updates in the relational setting for query languages that include both join and either project or union.

In the context of XML data, Braganholo, Heuser, and Vittori [dPBHV01], and Braganholo, Davidson, and Heuser [BDH03] and others studied the problem of updating relational databases "presented as XML." Their solution requires a one-to-one mapping between XML view elements and objects in the database to ensure that updates are unambiguous.

Tatarinov, Ives, Halevy, and Weld [TIHW01] described a mechanism for translating updates on XML structures that are stored in an underlying relational database. In this setting there is again an isomorphism between the relational source and the XML view, so updates are unambiguous—rather, the problem is choosing the most efficient way of translating a given XML update into a sequence of relational operations.

The view update problem has also been studied in the context of object-oriented databases. Scholl, Laasch, and Tresch [SLT91] restrict the notion of views to queries that preserve object identity. The view update problem is greatly simplified in this setting, as the objects contained in the view are the objects of the database, and an update on the view is directly an update on objects of the database.

Another problem that is sometimes mentioned in connection with view update translation is that of *incremental view maintenance* [GMS93]—efficiently recalculating an abstract view after a small update to the underlying concrete view. Although the phrase "view update problem" is sometimes, confusingly, used for work in this domain, these problems are fundamentally different.

7.4 Model Transformations

In the software engineering community, there has been interest in using lenses for model-driven software development. The model-driven approach allows programmers to derive implementations (e.g., Java classes) from formal specifications (e.g., UML diagrams). In systems based on model transformations, developers often need to maintain complex relationships between the model and the code—e.g., *refining* models to code and checking *conformance* of models to respective metamodels. Model transformations are mechanisms for establishing—and re-establishing, in the presence of change—relationships among models and between models and code [CH06]. Bidirectional model transformations are of particular interest if the related artifacts can be edited independently [AC08]. Formalisms such as *triple graph grammars* can be used to describe bidirectional transformations between models [Sch95]. Recently, Stevens [Ste08a, Ste08b, Ste07] has applied lenses to model transformations. Similar ideas have also been pursued by [XLH^{+}07] and [HHKN09]. The survey paper by Czarnecki, Foster, Hu, Lämmel, Schürr, Terwilliger describes the connections between bidirectional languages, view update, and model transformations [CFH$^+$09].

7.5 Security

The use of views as a mechanism for controlling access to confidential data has a long history in relational systems. Denning, Akl, Heckman, Lunt, Morgenstern, Neumann, and Schell proposed a framework that uses views to enforce access control policies with multiple confidentiality levels [DAH$^+$87]. Security views were first proposed as a security mechanism for XML data by Stoica and Farkas [SF02] and were later studied extensively by Fan and his colleagues in a series of papers [FCG04, FGJK06, FGJK07]. The key difference between previous work on security views and the framework proposed in Chapter 5, of course, is support for updates. Additionally, most previous systems do not provide a way to formally characterize the data kept confidential by the view—the query that defines the view essentially *is* the privacy policy. Lastly, views in previous systems have typically been virtual, while the views constructed using lenses are materialized. Fan [FCG04] has argued that materializing views is not practical, because many different security views are often needed when policies are complex. We find this argument compelling in the traditional database setting, where data sources are typically very large, but believe that there are also many applications where building materialized security views will be practical. Moreover, in at least some applications, views *must* be materialized—e.g., if the regraded view is intended to be sent over the network and displayed in a web browser.

The idea of using static analyses to track flows of information in programs was originally proposed by Denning and Denning [DD77] and has since been applied in a variety of languages, including Jif [Mye99], a secure variant of Java, and FlowCaml [PS03], a secure variant of OCaml. The excellent survey article by Sabelfeld and Myers [SM03] gives a general overview of the entire area and numerous citations.

Rather less work has focused on applying information-flow analyses to data processing languages. The developers of ℂDuce, a functional language for processing XML data, studied an extension of the language where labels corresponding to security levels are propagated dynamically [BBC03]. Foster, Green, and Tannen proposed a mechanism for ensuring non-interference properties of tree transformations using a semantics that propagates dynamic provenance annotations [FGT08]. The Fable language also propagates security labels dynamically [SCH08, CSH07]. Fable does not fix a particular semantics for label propagation, but instead provides a general framework that enforces a strict boundary between ordinary program code, which must treat labels opaquely, and security code, which may manipulate labels freely. Thus, it can be used to implement a variety of static and dynamic techniques for tracking information flows in programs. Cheney, Ahmed, and Ucar have introduced a general framework for comparing static and dynamic approaches to many dependency analyses including information flow [CAA07].

Integrity can be treated as a formal dual to confidentiality, as was first noted by Biba [Bib77]. Thus, most of the languages discussed above can also be used to track integrity properties of data. However, as noted by Li, Mao, and Zdancewic [LMZ03], information-flow analyses provide weaker guarantees for integrity compared to confidentiality when code is untrusted. Specific mechanisms for tracking integrity have also been included in a variety of languages: Perl has a simple taint tracking mechanism for data values [WCO00]. Wassermann and Su proposed a more powerful approach based on a dynamic analysis of generated strings that tracks tainted data in PHP scripts [WS07]. [STFW01] developed a taint analysis for C code using the `cqual` system. Finally, researchers at IBM have recently implemented a taint analysis tool for Java designed to scale to industrial-size web applications [TPF$^+$09].

Chapter 8

Summary and Future Work

This dissertation demonstrates that bidirectional programming languages are an effective means of defining updatable views. Starting from the foundations, we developed the framework of basic lenses, which are bidirectional transformations characterized by intuitive semantic laws. Then, building on this foundation, we designed a language for describing lenses on strings, with natural syntax based on the regular operators and a type system that guarantees well behavedness. We also studied the complications that arise when lenses are used to manipulate data containing unimportant details, ordered data, and confidential data. Finally, we described the implementation of all these ideas in the Boomerang language.

These results do not, however, close the book on bidirectional programming languages. On the contrary, our work can be extended in many directions. This final chapter identifies some specific avenues for future research.

8.1 Data Model

In this dissertation, we have primarily focused on lenses for strings. We did this as a matter of research strategy—to keep our focus on issues related to bidirectional languages and avoid getting distracted by technicalities related to the data model. However, the semantic space of lenses is completely generic so we can also instantiate it with other structures. We plan to investigate lenses for richer structures—e.g., objects, graphs, complex values, etc.—in future work. One motivation is that richer data models are simply a better fit for many applications. Another is that working with a richer data model should lead naturally to new and interesting lens primitives. It will also force us to extend concepts such as chunks, which are central to the semantics of resourceful lenses presented in Chapter 4, to richer settings. The main challenge will be developing type systems that are powerful enough to express the constraints needed to ensure well behavedness.

We, and others, have already made some progress in this area. In previous work we developed lenses for trees [FGM$^+$07], and Bohannon, Pierce, and Vaughan investigated lenses for relations [BVP06]. Recently, Hidaka, Hu, Kato, and Nakano proposed a bidirectional language based on the graph query language UnQL [HHKN09]. Graphs are of particular interest because they are often used to represent software at various levels of abstraction in model-driven development.

8.2 Syntax

At the level of syntax, our investigation has focused almost exclusively on combinators (with one exception—the grammars described in Chapter 6). This low-level approach to language design has proven to be an effective strategy for making progress on fundamental issues—e.g., how standard constructs such as conditionals, products, iteration operators, etc. should behave as lenses—but most programmers find the "point-free" style of programming unintuitive. We are interested in developing new lens languages based on more familiar syntactic constructs. One promising idea is to start from the nested relational calculus (NRC) [BNTW95]. NRC has a rich data model that can represent a wide variety of structures but a simple programming model based on structural recursion. A key technical challenge will be developing the machinery for interpreting NRC bidirectionally including dealing with variables. Another challenge in supporting richer syntax will be finding elegant ways to deal with lenses defined by recursion. Semantically, recursive lens definitions do not raise any deep issues—see [FGM$^+$07, Section 3]—but modifying the framework to support general recursive definitions requires changing the definition of lenses so that the component functions are partial. This means that reasoning about totality needs to be done separately.

8.3 Audit

One of the main unresolved tensions in the design of lenses involves the interplay between totality and very well behavedness. We want lenses to be total, but we also want them to preserve the underlying source data to the extent possible. Unfortunately, for certain transformations, we simply cannot have both of these—we either need to reject certain updates, or we need to allow the *put* function to make heavy changes to the source. In this dissertation, we have resolved this tension by simply taking totality to be the primary consideration and allowing lenses that are only well behaved. This seems to be a reasonable choice, but it means that a lens can sometimes have an unintended effect on the underlying source.

One idea for improving the usability of lenses that are only well behaved is to have them generate logs of their behavior. The idea is that instead of simply returning the updated source, the *put* function would generate a log of update operations to be performed on the source. These logs would provide more refined information about the nature of the update produced by the lens and would provide a concrete artifact that the owner of the source could use to return the system to a good state if they decide that the update is unacceptable. The main challenges will be designing elegant representations for logs and a theory of correct audit that makes it possible to compare and evaluate different auditing strategies. This work will also connect the operation-based approach to view update, which is often used in databases, with the state-based approach.

A related topic is exploring the relationship between view update and view maintenance. The goal of view maintenance systems is to propagate source updates to the view in an efficient manner. Intuitively, because many updates only affect a small portion of the view, it should be possible to translate them to small incremental updates. The complements used in bidirectional languages closely resemble the "trace" artifacts that have been proposed in the context of self-adjusting computation [ABH06]—a general technique for implementing incremental maintenance. We are interested in making the connection between bidirectional and self-adjusting computation explicit. This will be interesting in its own right, and will also have a significant practical benefit, leading to efficient mechanisms for maintaining the views defined by lenses.

8.4 Optimization

Another area for further work is optimization. Currently, the Boomerang interpreter implements the lens primitives directly. However, it should be possible to optimize the performance of many lens programs using algebraic rewritings. For example, the lens $(l_1{}^*;\ l_2{}^*)$ behaves the same as $(l_1; l_2)^*$

(when both are well typed), but the second lens should run substantially faster since it only has to iterate over the source once. We are interested in developing an algebraic theory of lenses that could serve as the basis for an optimizing compiler. We would also like to explore lenses that process the source in streaming fashion. This idea is motivated by examples such the UniProt database, where the sources can be several gigabytes in size! The idea in these streaming lenses would be to develop operators such as an iteration combinator whose *get* function produces the view one elements one at a time, rather than operating on the whole sequence. Similarly, we would need to retool *put* function to operate on elements of the view one piece at a time. To optimize the memory requirements of streaming lenses, we also plan to investigate lenses that use minimal complements.

8.5 Security

The secure lenses described in Chapter 5 use both static and dynamic mechanisms to verify security properties of lens programs. Unlike the rest of the work described in this dissertation, however, these analyses have not yet implemented. We hope to pursue this line of work in the future, implementing the secure lens type system and exploring connections to mechanisms that have been proposed in other settings—e.g., dynamic label propagation [ZM07, SST07] and provenance [BKT01, FGT08]. We would also like to explore declassification operators [ML97], quantitative measures of information flow [ME08], and formal notions of privacy [MS07] in the context of secure lenses. We believe that the time is especially ripe for languages designed with security in mind because although many systems manipulate sensitive data, few languages provide any tools for establishing that they do so correctly. As a result, relatively minor programming bugs often lead to massive security breaches. Secure lenses address a simple instance of this problem—providing a way to share data securely at fine levels of granularity—but much work remains.

Bibliography

[ABH06] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, 2006.

[Abr05] Sampson Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005.

[AC08] Michal Antkiewicz and Krzysztof Czarnecki. Design Space of Heterogeneous Synchronization. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Revised Papers*, volume 5235 of *LNCS*, pages 3–46. Springer, 2008.

[ACM97] Serge Abiteboul, Sophie Cluet, and Tova Milo. Correspondence and translation for heterogeneous data. In *International Conference on Database Theory (ICDT), Delphi, Greece*, 1997.

[ACM98] Serge Abiteboul, Sophie Cluet, and Tova Milo. A logical view of structure files. *VLDB Journal*, 7(2):96–114, 1998.

[AG00] Sergei M. Abramov and Robert Glück. The universal resolving algorithm: Inverse computation in a functional language. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837, pages 187–212. Springer-Verlag, 2000.

[AG02] Sergei M. Abramov and Robert Glück. Principles of inverse computation and the universal resolving algorithm. In Torben Mogensen, David Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 2002.

[And04] D. Calvin Andrus. Toward a complex adaptive intelligence community: The wiki and the blog. *Studies in Intelligence*, 49, September 2004.

[AT96] Paolo Atzeni and Riccardo Torlone. Management of multiple models in an extensible database design tool. In *Proceedings of EDBT'96, LNCS 1057*, 1996.

[AT97] Paolo Atzeni and Riccardo Torlone. MDM: a multiple-data model tool for the management of heterogeneous database schemes. In *Proceedings of ACM SIGMOD, Exhibition Section*, pages 528–531, 1997.

[Bak92] Henry G. Baker. Nreversal of fortune—the thermodynamics of garbage collection. In *International Workshop on Memory Managment (IWMM), St. Malo, France*, number 637 in Lecture Notes in Computer Science, pages 507–524. Springer-Verlag, September 1992.

[BBC03] 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.

[BCF+10] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. Technical Report MS-CIS-10-01, University of Pennsylvania, Department of Computer and Information Science, January 2010.

[BCGS91] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991.

[BCPV07] 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.

[BDH03] Vanessa Braganholo, Susan Davidson, and Carlos Heuser. On the updatability of XML views over relational databases. In *WebDB 2003*, 2003.

[Ben73] Charles H. Bennet. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[Ben05] Nick Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4):503–542, 2005.

[Ber79] Jean Berstel. *Transductions and Context-Free Languages*. Teubner Verlag, 1979.

[BFP+08] 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.

[Bib77] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR 76-372, The MITRE Corporation, 1977.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, number 1973 in LNCS, pages 316–330. Springer, 2001.

[BKT02] 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 (PODS), Madison, WI*, pages 150–158, 2002.

[Bla77] Meera Blattner. Single-valued a-transducers. *Journal of Computer and System Sciences*, 15(3):310–327, 1977.

[BMS08] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4–5):385–406, 2008. Short version in DBPL '05.

[BNTW95] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *TCS*, 149(1):3–48, 1995.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[BS81] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.

[BSKW91] Thierry Barsalou, Niki Siambela, Arthur M. Keller, and Gio Wiederhold. Updating relational databases through object-based views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Denver, CO*, pages 248–257, 1991.

[BVP06] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Chicago, TL*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[CAA07] James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. In *Symposium on Database Programming Languages (DBPL), Vienna, Austria*, pages 138–152, 2007.

[CFH$^+$09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. GRACE meeting notes, state of the art, and outlook. In *International Conference on Model Transformations (ICMT), Zurich, Switzerland*, pages 260–283, June 2009. Invited paper.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[CP84] S. S. Cosmadakis and Christos H. Papadimitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984.

[CSH07] 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/.

[DAH$^+$87] D.E. Denning, S.G. Akl, M. Heckman, T.F. Lunt, M. Morgenstern, P.G. Neumann, and R.R. Schell. Views for multilevel database security. *IEEE Transactions on Software Engineering*, 13(2):129–140, 1987.

[Dat03] C. J. Date. *An Introduction to Database Systems (Eighth Edition)*. Addison Wesley, 2003.

[DB82] 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.

[DD77] Dorothy E. Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Dij79] Edsger W. Dijkstra. Program inversion. In Friedrich L. Bauer and Manfred Broy, editors, *Program Construction, International Summer School, July 26 - August 6, 1978, Marktoberdorf, germany*, volume 69 of *Lecture Notes in Computer Science*. Springer, 1979.

[dPBHV01] Vanessa de Paula Braganholo, Carlos A. Heuser, and Cesar Roberto Mariano Vittori. Updating relational databases through XML views. In *Proc. 3rd Int. Conf. on Information Integration and Web-based Applications and Services (IIWAS)*, 2001.

[EG07] Robert Ennals and David Gay. Multi-language synchronization. In *European Symposium on Programming (ESOP), Braga, Portugal*, volume 4421 of *Lecture Notes in Computer Science*, pages 475–489. Springer-Verlag, 2007.

[Ege05] David T. Eger. Bit level types, 2005. Unpublished manuscript. Available from `http://www.yak.net/random/blt/blt-drafts/03/blt.pdf`.

[FCG04] 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.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Pittsburgh, PA*, pages 48–59, October 2002.

[FG05] 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.

[FGJK06] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. SMOQE: A system for providing secure access to XML. In *International Conference on Very Large Data Bases (VLDB), Seoul, Korea*, pages 1227–1230, September 2006.

[FGJK07] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Rewriting regular XPath queries on XML views. In *International Conference on Data Engineering (ICDE), Istanbul, Turkey*, pages 666–675, April 2007.

[FGK$^+$07] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4), June 2007. Short version in DBPL '05.

[FGM$^+$07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007. Short version in POPL '05.

[FGT08] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: Queries and provenance. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Vancouver, BC*, pages 271–280, June 2008.

[FL05] Steve Fogel and Paul Lane. *Oracle Database Administrator's Guide*. Oracle, June 2005.

[Fla06] Cormac Flanagan. Hybrid type checking. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Charleston, SC*, pages 245–256, 2006.

[FP08] J. Nathan Foster and Benjamin C. Pierce. *Boomerang Programmer's Manual*, 2008. Available from `http://www.seas.upenn.edu/~harmony/`.

[FPP08] J. Nathan Foster, Alexandre Pilkiewcz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, BC*, pages 383–395, September 2008.

[FPZ09] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. Updatable security views. In *IEEE Computer Security Foundations Symposium (CSF), Port Jefferson, NY*, July 2009.

[FWZW08] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, CA*, pages 421–434, 2008.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arith-mé-tique d'ordre supérieur*. Thèse d'état, University of Paris VII, 1972. Summary in J. E. Fenstad, editor, *Scandinavian Logic Symposium*, pp. 63–92, North-Holland, 1971.

[GK07] Michael Greenberg and Shriram Krishnamurthi. Declarative Composable Views, May 2007. Undergraduate Honors Thesis, Brown University.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD Conference*, pages 157–166, 1993.

[GPW10] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, pages 353–364, January 2010.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems (TODS)*, 13(4):486–524, 1988.

[Heg90] 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.

[Heg04] Stephane J. Hegner. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence*, 40:63–125, 2004. Summary in *Foundations of Information and Knowledge Systems*, 2002, pp. 230–249.

[HHKN09] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. A compositional approach to bidirectional model transformation. Technical report, May 2009. New Ideas and Emerging Results Track.

[HMT08] 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. Short version in PEPM '04.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, Boston, MA, USA, 1979.

[IA92] ISO-ANSI. Database language SQL2 standard, 1992. X3H2-92-154.

[IA08] ISO-ANSI. Database language SQL standard, 2008. IEC 9075(1-4,9-11,13,14):2008.

[IBM04] IBM. *IBM DB2 Universal Database Administration Guide: Implementation*, 2004.

[JRD01] Michael Johnson, Robert Rosebrugh, and C. N. G. Dampney. View updates in a semantic data modelling paradigm. In *ADC '01: Proceedings of the 12th Australasian conference on Database technologies*, pages 29–36, Washington, DC, USA, 2001. IEEE Computer Society.

[Kel85] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Portland, OR*, 1985.

[Kel86] A. M. Keller. Choosing a view update translator by dialog at view definition time. In *VLDB'86*, 1986.

[Ken04] Andrew J. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

[KH06] 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.

[KI08] G. Karvounarakis and Z.G. Ives. Bidirectional mappings for data and update exchange. In *11th International Workshop on the Web and Databases, WebDB*, 2008.

[Lab09] Reductive Labs. Using Puppet with Augeas, January 2009. Available from http://reductivelabs.com/trac/puppet/wiki/PuppetAugeas.

[Lan61] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961. (Republished in IBM Jour. of Res. and Devel., 44(1/2):261-269, Jan/Mar. 2000).

[Lec03] Jens Lechtenbörger. The impact of the constant complement approach towards view updating. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), San Diego, CA*, pages 49–55. ACM, June 9–12 2003.

[LHT07] 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.

[LMZ03] Peng Li, Yun Mao, and Steve Zdancewic. Information Integrity Policies. In *Proceedings of the First Workshop on Formal Aspects in Security and Trust (FAST)*, Pisa, Italy, September 2003.

[Lor05] Diana Lorentz. *Oracle Database SQL Reference*. Oracle, December 2005.

[Lut08] David Lutterkort. Augeas–A configuration API. In *Linux Symposium, Ottawa, ON*, pages 47–56, 2008.

[Lut09] David Lutterkort. Netcf: A library for configuring network interfaces, March 2009. Available from https://fedorahosted.org/netcf.

[Mal09] Dave Malcolm. Squeal: An SQL-like interface for the command line, June 2009. Available from https://fedorahosted.org/squeal.

[Mas84] Y. Masunaga. A relational database view update translation mechanism. In *VLDB'84*, 1984.

[McC56] John McCarthy. The inversion of functions defined by turing machines. In Claude E. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Mathematical Studies*, volume 34, pages 177–181. Princeton University Press, 1956.

[ME08] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Tuscon, AZ*, pages 193–205, 2008.

[Mee98] Lambert Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript, available from ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps.

[MHH+01] Renée J. Miller, Mauricio A. Hernandez, Laura M. Haas, Lingling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. The clio project: Managing heterogeneity. *ACM SIGMOD Record*, 30(1):78–83, March 2001.

[MHN+07] 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.

[MHT04a] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bidirectional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 2–20, November 2004.

[MHT04b] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction (MPC)*, 2004.

[MHT06] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. Bidirectionalizing tree transformation languages: A case study. *JSSST Computer Software*, 23(2):129–141, 2006.

[Mic05] Microsoft. *Creating and Maintaining Databases*, 2005.

[ML97] Andrew Myers and Barbara Liskov. A decentralized model for information flow control. In *ACM Symposium on Operating Systems Principles (SOSP), Saint Malo, France*, pages 129–142, 1997.

[Mø01] Anders Møller. The brics automaton package, 2001.

[MS07] Gerome Miklau and Dan Suciu. A formal analysis of information disclosure in data exchange. *Journal of Computer and Systems Sciences*, 73(3):507–534, 2007.

[MT85] C. M. B. Medeiros and F. W. Tompa. Understanding the implications of view update policies. In *VLDB'85*, 1985.

[Mye99] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Antonio, TX*, pages 228–241, 1999.

[ORT09] Scott Owens, John Reppy, and Aaron Turon. Regular expression derivatives reexamined. *Journal of Functional Programming*, 19:173–190, March 2009.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pin08] Raphael Pinson. Creating a lens step-by-step, July 2008. Available from http://augeas.net/page/Creating_a_lens_step_by_step.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

[Ram03] 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.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Paris, France*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Sch95] 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.

[SCH08] 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.

[SF02] 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.

[SLT91] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updatable Views in Object-Oriented Databases. In *Proc. 2nd Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, volume 566 of *Lecture Notes in Computer Science*. Springer, 1991.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[SST07] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *20th IEEE Computer Security Foundations Symposium (CSF)*, pages 203–217, July 2007.

[Ste07] Perdita Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007), Proceedings*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007.

[Ste08a] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Revised Papers*, volume 5235 of *LNCS*, pages 408–424. Springer, 2008.

[Ste08b] Perdita Stevens. Towards an Algebraic Theory of Bidirectional Transformations. In *Graph Transformations, 4th International Conference, ICGT 2008, Proceedings*, volume 5214 of *LNCS*, pages 1–17. Springer, 2008.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Conference on USENIX Security Symposium (SSYM), Washington DC*, pages 16–16, 2001.

[TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *ACM SIGMOD International Conference on Management of Data (SIGMOD), Santa Barbara, CA*, 2001.

[TPF$^+$09] Omer Tripp, Marco Pistoia, Stephen Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland*, 2009. To appear.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany*, 1987.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl (3rd Edition)*. O'Reilly, July 2000.

[WF07] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming, Freiburg, Germany*, pages 15–26, 2007.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, CA*, pages 32–41, 2007.

[XLH$^+$07] 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.

[ZM07] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3):67–84, March 2007.

Proofs

This appendix contains the proofs for each of the results in our technical development, including well-behavedness proofs for each of our primitives.

Basic Lens Proofs

$$\boxed{\frac{E \in \mathcal{R}}{\mathit{copy}\ E \in [\![E]\!] \Longleftrightarrow [\![E]\!]}}$$

2.3.20 Lemma: Let E be a regular expression. Then *copy* E is a basic lens in $[\![E]\!] \Longleftrightarrow [\![E]\!]$.

Proof: We prove each basic lens law separately.

▶ **GetPut:** Let e be a string in $[\![E]\!]$. We calculate as follows

$$\begin{aligned}
&(\mathit{copy}\ E).\mathit{put}\ ((\mathit{copy}\ E).\mathit{get}\ e)\ e \\
&= (\mathit{copy}\ E).\mathit{put}\ e\ e && \text{by definition } (\mathit{copy}\ E).\mathit{get} \\
&= e && \text{by definition } (\mathit{copy}\ E).\mathit{put}
\end{aligned}$$

and obtain the required result.

▶ **PutGet:** Let e and e' be strings in $[\![E]\!]$. We calculate as follows

$$\begin{aligned}
&(\mathit{copy}\ E).\mathit{get}\ ((\mathit{copy}\ E).\mathit{put}\ e'\ e) \\
&= (\mathit{copy}\ E).\mathit{get}\ e' && \text{by definition } (\mathit{copy}\ E).\mathit{put} \\
&= e' && \text{by definition } (\mathit{copy}\ E).\mathit{get}
\end{aligned}$$

and obtain the required result.

▶ **CreateGet:** Let e be a string in $[\![E]\!]$. We calculate as follows

$$\begin{aligned}
&(\mathit{copy}\ E).\mathit{get}\ ((\mathit{copy}\ E).\mathit{create}\ e) \\
&= (\mathit{copy}\ E).\mathit{get}\ e && \text{by definition } (\mathit{copy}\ E).\mathit{create} \\
&= e && \text{by definition } (\mathit{copy}\ E).\mathit{get}
\end{aligned}$$

and obtain the required result. □

$$\boxed{\frac{E \in \mathcal{R} \qquad [\![E]\!] \neq \emptyset \qquad u \in \Sigma^*}{\mathit{const}\ E\ u \in [\![E]\!] \Longleftrightarrow \{u\}}}$$

2.3.21 Lemma: Let E be a regular expression and u a string such that $[\![E]\!] \neq \emptyset$. Then *const* $E\ u$ is a basic lens in $[\![E]\!] \Longleftrightarrow \{u\}$.

Proof: We prove each basic lens law separately.

▶ **GetPut:** Let $e \in [\![E]\!]$. We calculate as follows

$$\begin{aligned} & (\mathit{const}\ E\ u).\mathit{put}\ ((\mathit{const}\ E\ u).\mathit{get}\ e)\ e \\ = {} & (\mathit{const}\ E\ u).\mathit{put}\ u\ e && \text{by definition } (\mathit{const}\ E\ u).\mathit{get} \\ = {} & e && \text{by definition } (\mathit{const}\ E\ u).\mathit{put} \end{aligned}$$

and obtain the required result.

▶ **PutGet:** Let $u \in \{u\}$ and $e \in [\![E]\!]$. We calculate as follows

$$\begin{aligned} & (\mathit{const}\ E\ u).\mathit{get}\ ((\mathit{const}\ E\ u).\mathit{put}\ u\ e) \\ = {} & (\mathit{const}\ E\ u).\mathit{get}\ e && \text{by definition } (\mathit{const}\ E\ u).\mathit{put} \\ = {} & u && \text{by definition } (\mathit{const}\ E\ u).\mathit{get} \end{aligned}$$

and obtain the required result.

▶ **CreateGet:** Let $u \in \{u\}$. We calculate as follows

$$\begin{aligned} & (\mathit{const}\ E\ u).\mathit{get}\ ((\mathit{const}\ E\ u).\mathit{create}\ u) \\ = {} & (\mathit{const}\ E\ u).\mathit{get}\ (\mathit{representative}(E)) && \text{by definition } (\mathit{const}\ E\ u).\mathit{create} \\ = {} & u && \text{by definition } (\mathit{const}\ E\ u).\mathit{get} \end{aligned}$$

and obtain the required result. □

$$\boxed{\dfrac{l \in S \Longleftrightarrow V \qquad f \in V \to S}{\mathit{default}\ l\ f \in S \Longleftrightarrow V}}$$

2.3.22 Lemma: Let $l \in S \Longleftrightarrow V$ be a basic lens and $f \in V \to S$ a total function. Then *default* $l\ f$ is a basic lens in $S \Longleftrightarrow V$.

Proof: We prove each basic lens law separately.

▶ **GetPut:** Immediate by GetPut for l.

▶ **PutGet:** Immediate by PutGet for l.

▶ **CreateGet:** Immediate by PutGet for l. □

$$\boxed{\dfrac{\begin{array}{ll} l_1 \in S_1 \Longleftrightarrow V_1 & S_1 \cdot^! S_2 \\ l_2 \in S_2 \Longleftrightarrow V_2 & V_1 \cdot^! V_2 \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2) \Longleftrightarrow (V_1 \cdot V_2)}}$$

2.3.23 Lemma: Let $l_1 \in S_1 \Longleftrightarrow V_1$ and $l_2 \in S_2 \Longleftrightarrow V_2$ be basic lenses such that $S_1 \cdot^! S_2$ and $V_1 \cdot^! V_2$. Then $l_1 \cdot l_2$ is a basic lens in $(S_1 \cdot S_2) \Longleftrightarrow (V_1 \cdot V_2)$.

Proof: We prove each basic lens law separately.

▶ **GetPut:** Let $s \in (S_1 \cdot S_2)$. As $S_1 \cdot^! S_2$ there exist unique strings $s_1 \in S_1$ and $s_2 \in S_2$ such that $s = s_1 \cdot s_2$. Using these facts, we calculate as follows

$$\begin{aligned}
& (l_1 \cdot l_2).\mathit{put}\ ((l_1 \cdot l_2).\mathit{get}\ s)\ s \\
= {} & (l_1 \cdot l_2).\mathit{put}\ ((l_1 \cdot l_2).\mathit{get}\ (s_1 \cdot s_2))\ (s_1 \cdot s_2) && \text{by definition } s_1 \text{ and } s_2 \\
= {} & (l_1 \cdot l_2).\mathit{put}\ ((l_1.\mathit{get}\ s_1) \cdot (l_2.\mathit{get}\ s_2))\ (s_1 \cdot s_2) && \text{by definition } (l_1 \cdot l_2).\mathit{get} \\
= {} & (l_1.\mathit{put}\ (l_1.\mathit{get}\ s_1)\ s_1) \cdot (l_2.\mathit{put}\ (l_2.\mathit{get}\ s_2)\ s_2) && \text{by definition } (l_1 \cdot l_2).\mathit{put} \\
& \quad \text{with } V_1 \cdot^! V_2 \text{ and } \mathrm{cod}(l_1.\mathit{get}) = V_1 \text{ and } \mathrm{cod}(l_2.\mathit{get}) = V_2 \\
= {} & s_1 \cdot s_2 && \text{by } \textsc{GetPut} \text{ for } l_1 \text{ and } l_2 \\
= {} & s && \text{by definition } s_1 \text{ and } s_2
\end{aligned}$$

and obtain the required equality.

▶ **PutGet:** Let $v \in (V_1 \cdot V_2)$ and $s \in (S_1 \cdot S_2)$. As $V_1 \cdot^! V_2$ there exist unique strings $v_1 \in V_1$ and $v_2 \in V_2$ such that $v = v_1 \cdot v_2$. Similarly, as $S_1 \cdot^! S_2$ there exist unique strings $s_1 \in S_1$ and $s_2 \in S_2$ such that $s = s_1 \cdot s_2$. Using these facts, we calculate as follows

$$\begin{aligned}
& (l_1 \cdot l_2).\mathit{get}\ ((l_1 \cdot l_2).\mathit{put}\ v\ s) \\
= {} & (l_1 \cdot l_2).\mathit{get}\ ((l_1 \cdot l_2).\mathit{put}\ (v_1 \cdot v_2)\ s && \text{by definition } v_1 \text{ and } v_2 \\
= {} & (l_1 \cdot l_2).\mathit{get}\ ((l_1 \cdot l_2).\mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) && \text{by definition } s_1 \text{ and } s_2 \\
= {} & (l_1 \cdot l_2).\mathit{get}\ ((l_1.\mathit{put}\ v_1\ s_1) \cdot (l_2.\mathit{put}\ v_2\ s_2)) && \text{by definition } (l_1 \cdot l_2).\mathit{put} \\
= {} & (l_1.\mathit{get}\ (l_1.\mathit{put}\ v_1\ s_1)) \cdot (l_2.\mathit{get}\ (l_2.\mathit{put}\ v_2\ s_2)) && \text{by definition } (l_1 \cdot l_2).\mathit{get} \\
& \quad \text{with } \mathrm{cod}(l_1.\mathit{put}) = S_1 \text{ and } \mathrm{cod}(l_2.\mathit{put}) = S_2 \text{ and } S_1 \cdot^! S_2 \\
= {} & v_1 \cdot v_2 && \text{by } \textsc{PutGet} \text{ for } l_1 \text{ and } l_2 \\
= {} & v && \text{by definition } v_1 \text{ and } v_2
\end{aligned}$$

and obtain the required equality.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{\begin{gathered} S_1 \cap S_2 = \emptyset \\ l_1 \in S_1 \Longleftrightarrow V_1 \\ l_2 \in S_2 \Longleftrightarrow V_2 \end{gathered}}{l_1 \,|\, l_2 \in (S_1 \cup S_2) \Longleftrightarrow (V_1 \cup V_2)}}$$

2.3.24 Lemma: Let $l_1 \in S_1 \Longleftrightarrow V_1$ and $l_2 \in S_2 \Longleftrightarrow V_2$ be basic lenses such that the intersection $S_1 \cap S_2$ of the source types is empty. Then $l_1 \,|\, l_2$ is a basic lens in $(S_1 \cup S_2) \Longleftrightarrow (V_1 \cup V_2)$.

Proof: We prove each basic lens law separately.

▶ **GetPut:** Let $s \in (S_1 \cup S_2)$. We analyze two cases.

Case $s \in S_1$: We calculate as follows

$$\begin{aligned}
& (l_1 \,|\, l_2).\mathit{put}\ ((l_1 \,|\, l_2).\mathit{get}\ s)\ s \\
= {} & (l_1 \,|\, l_2).\mathit{put}\ (l_1.\mathit{get}\ s)\ s && \text{by the definition of } (l_1 \,|\, l_2).\mathit{get} \\
& \quad \text{with } s \in S_1 \\
= {} & l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ s && \text{by the definition of } (l_1 \,|\, l_2).\mathit{put} \\
& \quad \text{with } \mathrm{cod}(l.\mathit{get}) = V_1 \text{ and } s \in S_1 \\
= {} & s && \text{by } \textsc{PutGet} \text{ for } l_1
\end{aligned}$$

and obtain the required equality.

Case $s \in S_2$**:** Symmetric to the previous case.

▶ **PutGet:** Let $v \in V_1 \cup V_2$ and $s \in S_1 \cup S_2$. We analyze several cases.

Case $v \in V_1$ **and** $s \in S_1$**:** We calculate as follows

$$\begin{aligned}
&(l_1 \,|\, l_2).\mathit{get}\ ((l_1 \,|\, l_2).\mathit{put}\ v\ s) && \\
&= (l_1 \,|\, l_2).\mathit{get}\ (l_1.\mathit{put}\ v\ s) && \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
&\quad \text{with } v \in V_1 \text{ and } s \in S_1 && \\
&= l_1.\mathit{get}\ (l_1.\mathit{put}\ v\ s) && \text{by definition } (l_1 \,|\, l_2).\mathit{get} \\
&\quad \text{with } \mathrm{cod}(l_1.\mathit{put}) = S_1 && \\
&= v && \text{by } \textsc{PutGet} \text{ for } l
\end{aligned}$$

and obtain the required equality.

Case $v \in V_2$ **and** $s \in S_2$**:** Symmetric to the previous case.

Case $v \in V_1 - V_2$ **and** $s \in S_2$**:** We calculate as follows

$$\begin{aligned}
&(l_1 \,|\, l_2).\mathit{get}\ ((l_1 \,|\, l_2).\mathit{put}\ v\ s) && \\
&= (l_1 \,|\, l_2).\mathit{get}\ (l_1.\mathit{create}\ v) && \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
&\quad \text{with } v \in V_1 - V_2 \text{ and } s \in S_2 && \\
&= l_1.\mathit{get}\ (l_1.\mathit{create}\ v) && \text{by definition } (l_1 \,|\, l_2).\mathit{get} \\
&\quad \text{with } \mathrm{cod}(l_1.\mathit{create}) = S_1 && \\
&= v && \text{by } \textsc{CreateGet} \text{ for } l
\end{aligned}$$

and obtain the required equality.

Case $v \in V_2 - V_1$ **and** $s \in S_1$**:** Symmetric to the previous case.

▶ **CreateGet:** Let $v \in V_1 \cup V_2$. We analyze two cases.

Case $v \in V_1$**:** We calculate as follows

$$\begin{aligned}
&(l_1 \,|\, l_2).\mathit{get}\ ((l_1 \,|\, l_2).\mathit{create}\ v) && \\
&= (l_1 \,|\, l_2).\mathit{get}\ (l_1.\mathit{create}\ v) && \text{by definition } (l_1 \,|\, l_2).\mathit{create} \\
&\quad \text{with } v \in V_1 && \\
&= l_1.\mathit{get}\ (l_1.\mathit{create}\ v) && \text{by definition } (l_1 \,|\, l_2).\mathit{get} \\
&\quad \text{with } \mathrm{cod}(l_1.\mathit{create}) = S_1 && \\
&= v && \text{by } \textsc{CreateGet} \text{ for } l
\end{aligned}$$

and obtain the required equality.

Case $v \in V_2 - V_1$**:** Symmetric to the previous case. □

$$\boxed{\dfrac{\begin{array}{c} S^{!*} \qquad V^{!*} \\ l \in S \Longleftrightarrow V \end{array}}{l^* \in S^* \Longleftrightarrow V^*}}$$

2.3.25 Lemma: Let $l \in S \Longleftrightarrow V$ be a basic lens such that $S^{!*}$ and $V^{!*}$. Then l^* is a basic lens in $S^* \Longleftrightarrow V^*$.

Proof: We prove each basic lens law separately.

▶ **GetPut**: Let $s \in S^*$. As $S^{!*}$ there exist unique strings s_1 to s_n in S such that $s = (s_1 \cdots s_n)$. Using these facts, we calculate as follows

$$\begin{array}{ll}
\quad l^*.\mathit{put}\ (l^*.\mathit{get}\ s)\ s & \\
= l^*.\mathit{put}\ (l^*.\mathit{get}\ (s_1 \cdots s_n))\ (s_1 \cdots s_n) & \text{by definition } s_1 \text{ to } s_n \\
= l^*.\mathit{put}\ ((l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n))\ (s_1 \cdots s_n) & \text{by definition } l^*.\mathit{get} \\
= l.\mathit{put}\ (l.\mathit{get}\ s_1)\ s_1 \cdots l.\mathit{put}\ (l.\mathit{get}\ s_n)\ s_n & \text{by definition } l^*.\mathit{put} \\
\qquad \text{with } V^{!*} \text{ and } \mathrm{cod}(l.\mathit{get}) = V & \\
= s_1 \cdots s_n & \text{by } \textsc{GetPut} \text{ for } l \\
= s & \text{by definition } s_1 \text{ to } s_n
\end{array}$$

and obtain the required equality.

▶ **PutGet**: Let $v \in V^*$ and $s \in S^*$. As $V^{!*}$ there exist unique strings v_1 to v_n in V such that $v = (v_1 \cdots v_n)$. Similarly, as $S^{!*}$ there exist unique strings s_1 to s_m in S such that $s = (s_1 \cdots s_m)$. Using these facts, we calculate as follows

$$\begin{array}{ll}
\quad l^*.\mathit{get}\ (l^*.\mathit{put}\ v\ s) & \\
= l^*.\mathit{get}\ (l^*.\mathit{put}\ (v_1 \cdots v_n)\ s & \text{by definition } v_1 \text{ to } v_n \\
= l^*.\mathit{get}\ (l^*.\mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)) & \text{by definition } s_1 \text{ to } s_m \\
= l^*.\mathit{get}\ (s'_1 \cdots s'_n) & \text{by the definition of } l^*.\mathit{put} \\
\qquad \text{where } s'_i = \begin{cases} l.\mathit{put}\ v_i\ s_i\ i \in \{1, \ldots, \min(n, m)\} \\ l.\mathit{create}\ v_i\ i \in \{m+1, \ldots, n\} \end{cases} & \\
= (l.\mathit{get}\ s'_1) \cdots (l.\mathit{get}\ s'_n) & \text{by the definition of } l^*.\mathit{get} \\
\qquad \text{with } V^{!*} \text{ and } \mathrm{cod}(l.\mathit{put}) = \mathrm{cod}(l.\mathit{create}) = V & \\
= v_1 \cdots v_n & \text{by } \textsc{PutGet} \text{ and } \textsc{CreateGet} \text{ for } l \\
= v & \text{by definition } v_1 \text{ to } v_n
\end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof of PUTGET. □

$$\boxed{\dfrac{\begin{array}{c} l_1 \in S \Longleftrightarrow U \\ l_2 \in U \Longleftrightarrow V \end{array}}{l_1;l_2 \in S \Longleftrightarrow V}}$$

2.3.26 Lemma: Let $l_1 \in S \Longleftrightarrow U$ and $l_2 \in U \Longleftrightarrow V$ be basic lenses. Then $l_1;l_2$ is a basic lens in $S \Longleftrightarrow V$.

Proof: We prove each basic lens law separately.

▶ **GetPut**: Let $s \in S$. We calculate as follows

$$\begin{array}{ll}
\quad (l_1;l_2).\mathit{put}\ ((l_1;l_2).\mathit{get}\ s)\ s & \\
= (l_1;l_2).\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ s & \text{by definition } (l_1;l_2).\mathit{get} \\
= l_1.\mathit{put}\ (l_2.\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (l_1.\mathit{get}\ s))\ s & \text{by definition } (l_1;l_2).\mathit{put} \\
= l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ s & \text{by } \textsc{GetPut} \text{ for } l_2 \\
= s & \text{by } \textsc{GetPut} \text{ for } l_1
\end{array}$$

and obtain the required equality.

▶ **PutGet:** Let $v \in V$ and $s \in S$. We calculate as follows

$$\begin{array}{ll}
\quad (l_1;l_2).\mathit{get}\ ((l_1;l_2).\mathit{put}\ v\ s) & \\
= (l_1;l_2).\mathit{get}\ (l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s) & \text{by definition } (l_1;l_2).\mathit{put} \\
= l_2.\mathit{get}\ (l_1.\mathit{get}\ (l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s)) & \text{by definition } (l_1;l_2).\mathit{get} \\
= l_2.\mathit{get}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s)) & \text{by } \textsc{PutGet} \text{ for } l_1 \\
= v & \text{by } \textsc{PutGet} \text{ for } l_2
\end{array}$$

and obtain the required equality.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{[\![E]\!] \cap [\![F]\!] = \emptyset \qquad ([\![E]\!] \cup [\![F]\!])^{!*}}{\mathit{filter}\ E\ F \in ([\![E]\!] \cup [\![F]\!])^{*} \Longleftrightarrow [\![E]\!]^{*}}}$$

2.3.27 Lemma: Let E and F be regular expressions such that the intersection $[\![E]\!] \cap [\![F]\!]$ of $[\![E]\!]$ and $[\![F]\!]$ is empty and $([\![E]\!] \cup [\![F]\!])^{!*}$. Then *filter* $E\ F$ is a basic lens in $([\![E]\!] \cup [\![F]\!])^{*} \Longleftrightarrow [\![E]\!]^{*}$.

Proof: We prove each basic lens law separately. To shorten the proof, we will abbreviate (*filter* $E\ F$) as l.

▶ **GetPut:** Let $s \in ([\![E]\!] \cup [\![F]\!])^{*}$ be a string. As $([\![E]\!] \cup [\![F]\!])^{!*}$, there exist unique strings s_1 to s_n such that $s = (s_1 \cdots s_n)$. The proof is by induction on n.

Case $n = 0$**:** By the assumption of the case we have $s = \epsilon$. The equality

$$l.\mathit{put}\ (l.\mathit{get}\ \epsilon)\ \epsilon = \epsilon$$

follows immediately from the definitions of $l.\mathit{get}$ and $l.\mathit{put}$, str_filter, and str_unfilter.

Case $n > 0$ **and** $s_1 \in F$**:** By the assumptions of the case we have $s = s_1 \cdot (s_2 \cdots s_n)$. To lighten the notation, let $s' = (s_2 \cdots s_n)$. We calculate as follows

$$\begin{array}{ll}
\quad l.\mathit{put}\ (l.\mathit{get}\ s)\ s & \\
= l.\mathit{put}\ (l.\mathit{get}\ (s_1 \cdot s'))\ (s_1 \cdot s') & \text{by definition } s_1 \text{ and } s' \\
= l.\mathit{put}\ (\mathsf{str\_filter}\ E\ (s_1 \cdot s'))\ (s_1 \cdot s') & \text{by definition } l.\mathit{get} \\
= l.\mathit{put}\ (\mathsf{str\_filter}\ E\ s')\ (s_1 \cdot s') & \text{by definition } \mathsf{str\_filter} \\
\quad\quad \text{with } s_1 \in F & \\
= \mathsf{str\_unfilter}\ F\ (\mathsf{str\_filter}\ E\ s')\ (s_1 \cdot s') & \text{by definition } l.\mathit{put} \\
= s_1 \cdot (\mathsf{str\_unfilter}\ F\ (\mathsf{str\_filter}\ E\ s')\ s') & \text{by definition } \mathsf{str\_unfilter} \\
\quad\quad \text{with } s_1 \in F & \\
= s_1 \cdot (\mathsf{str\_unfilter}\ F\ (l.\mathit{get}\ s')\ s') & \text{by definition } l.\mathit{get} \\
= s_1 \cdot (l.\mathit{put}\ (l.\mathit{get}\ s')\ s') & \text{by definition } l.\mathit{put} \\
= s_1 \cdot s' & \text{by induction hypothesis} \\
= s & \text{by definition } s_1 \text{ and } s'
\end{array}$$

and obtain the required equality.

Case $n > 0$ **and** $s_1 \in E$**:** Similar to the previous case.

▶ **PutGet:** Let $s \in ([\![E]\!] \cup [\![F]\!])^{*}$ and $v \in [\![E]\!]^{*}$ be strings. As $([\![E]\!] \cup [\![F]\!])^{!*}$, there exist unique strings s_1 to s_n such that $s = (s_1 \cdots s_n)$ and v_1 to v_k such that $v = (v_1 \cdots v_k)$. The proof is by induction on n with an inner induction on k in the case where $n = 0$.

Case $n = 0$ and $k = 0$: By the assumption of the case we have $s = v = \epsilon$. The equality

$$l.\mathit{get}\ (l.\mathit{put}\ \epsilon\ \epsilon) = \epsilon$$

follows immediately from the definitions of $l.\mathit{get}$ and $l.\mathit{put}$, str_filter, and str_unfilter.

Case $n = 0$ and $k > 0$: By the assumption of the case we have $s = \epsilon$. To lighten the notation, let $v' = (v_2 \cdots v_k)$. We calculate as follows

$$\begin{aligned}
&l.\mathit{get}\ (l.\mathit{put}\ v\ s) && \\
&= l.\mathit{get}\ (l.\mathit{put}\ v\ \epsilon) && \text{by definition } s \\
&= l.\mathit{get}\ (l.\mathit{put}\ (v_1 \cdot v')\ \epsilon) && \text{by definition } v_1 \text{ and } v' \\
&= l.\mathit{get}\ (\mathsf{str\_unfilter}\ F\ (v_1 \cdot v')\ \epsilon) && \text{by definition } l.\mathit{put} \\
&= l.\mathit{get}\ (v_1 \cdot (\mathsf{str\_unfilter}\ F\ v'\ \epsilon)) && \text{by definition } \mathsf{str\_unfilter} \\
&\quad \text{with } v_1 \in F && \\
&= \mathsf{str\_filter}\ E\ (v_1 \cdot (\mathsf{str\_unfilter}\ F\ v'\ \epsilon)) && \text{by definition } l.\mathit{get} \\
&= v_1 \cdot (\mathsf{str\_filter}\ E\ (\mathsf{str\_unfilter}\ F\ v'\ \epsilon)) && \text{by definition } \mathsf{str\_filter} \\
&\quad \text{with } v_1 \in F && \\
&= v_1 \cdot (l.\mathit{get}\ (l.\mathit{put}\ v'\ \epsilon)) && \text{by definition } l.\mathit{put} \\
&= v_1 \cdot v' && \text{by induction hypothesis} \\
&= v && \text{by definition } v_1 \text{ and } v'
\end{aligned}$$

and obtain the required equality.

Case $n > 0$ and $s_1 \in F$: To lighten the notation, let $s' = (s_2 \cdots s_n)$. We calculate as follows

$$\begin{aligned}
&l.\mathit{get}\ (l.\mathit{put}\ v\ s) && \\
&= l.\mathit{get}\ (l.\mathit{put}\ v\ (s_1 \cdot s')) && \text{by definition } s_1 \text{ and } s' \\
&= l.\mathit{get}\ (\mathsf{str\_unfilter}\ F\ v\ (s_1 \cdot s')) && \text{by definition } l.\mathit{put} \\
&= l.\mathit{get}\ (s_1 \cdot (\mathsf{str\_unfilter}\ F\ v\ s')) && \text{by definition } \mathsf{str\_unfilter} \\
&= \mathsf{str\_filter}\ E\ (s_1 \cdot (\mathsf{str\_unfilter}\ F\ v\ s')) && \text{by definition } l.\mathit{get} \\
&= \mathsf{str\_filter}\ E\ (\mathsf{str\_unfilter}\ F\ v\ s') && \text{by definition } \mathsf{str\_filter} \\
&= l.\mathit{get}\ (l.\mathit{put}\ v\ s') && \text{by definition } l.\mathit{put} \\
&= v && \text{by induction hypothesis}
\end{aligned}$$

and obtain the required equality.

Case $n > 0$ and $s_1 \in E$: Similar to the previous case.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{\begin{array}{cc} l_1 \in S_1 \Longleftrightarrow V_1 & S_1 \cdot^! S_2 \\ l_2 \in S_2 \Longleftrightarrow V_2 & V_2 \cdot^! V_1 \end{array}}{l_1 \sim l_2 \in (S_1 \cdot S_2) \Longleftrightarrow (V_1 \cdot V_2)}}$$

2.3.28 Lemma: Let $l_1 \in S_1 \Longleftrightarrow V_1$ and $l_2 \in S_2 \Longleftrightarrow V_2$ be basic lenses such that $(S_1 \cdot^! S_2)$ and $(V_2 \cdot^! V_1)$. Then $(l_1 \sim l_2)$ is a basic lens in $(S_1 \cdot S_2) \Longleftrightarrow (V_2 \cdot V_1)$.

Proof: We prove each basic lens law separately.

▶ **GetPut**: Let $s \in S_1 \cdot S_2$. As $S_1 \cdot^! S_2$ there exist unique strings $s_1 \in S_1$ and $s_2 \in S_2$ such that $s = s_1 \cdot s_2$. Using these facts, we calculate as follows

$$\begin{aligned}
& (l_1 \sim l_2).\mathit{put}\ ((l_1 \sim l_2).\mathit{get}\ s)\ s \\
={} & (l_1 \sim l_2).\mathit{put}\ ((l_1 \sim l_2).\mathit{get}\ (s_1 \cdot s_2))\ (s_1 \cdot s_2) && \text{by definition } s_1 \text{ and } s_2 \\
={} & (l_1 \sim l_2).\mathit{put}\ ((l_2.\mathit{get}\ s_2) \cdot (l_1.\mathit{get}\ s_1))\ (s_1 \cdot s_2) && \text{by definition } (l_1 \sim l_2).\mathit{get} \\
={} & (l_1.\mathit{put}\ (l_1.\mathit{get}\ s_1)\ s_1) \cdot (l_2.\mathit{put}\ (l_2.\mathit{get}\ s_2)\ s_2) && \text{by definition } (l_1 \sim l_2).\mathit{put} \\
& \quad \text{with } V_2 \cdot^! V_1 \text{ and } \mathrm{cod}(l_1.\mathit{get}) = V_1 \text{ and } \mathrm{cod}(l_2.\mathit{get}) = V_2 \\
={} & s_1 \cdot s_2 && \text{by GETPUT for } l_1 \text{ and } l_2 \\
={} & s && \text{by definition } s_1 \text{ and } s_2
\end{aligned}$$

and obtain the required equality.

▶ **PutGet**: Let $v \in V_2 \cdot V_1$ and $s \in S_1 \cdot S_2$. As $V_2 \cdot^! V_1$ there exist unique strings $v_2 \in V_2$ and $v_1 \in V_1$ such that $v = (v_2 \cdot v_1)$. Similarly, as $S_1 \cdot^! S_2$ there exist unique strings $s_1 \in S_1$ and $s_2 \in S_2$ such that $s = s_1 \cdot s_2$. Using these facts, we calculate as follows

$$\begin{aligned}
& (l_1 \sim l_2).\mathit{get}\ ((l_1 \sim l_2).\mathit{put}\ v\ s) \\
={} & (l_1 \sim l_2).\mathit{get}\ ((l_1 \sim l_2).\mathit{put}\ (v_2 \cdot v_1)\ s) && \text{by definition } v_2 \text{ and } v_1 \\
={} & (l_1 \sim l_2).\mathit{get}\ ((l_1 \sim l_2).\mathit{put}\ (v_2 \cdot v_1)\ (s_1 \cdot s_2)) && \text{by definition } s_1 \text{ and } s_2 \\
={} & (l_1 \sim l_2).\mathit{get}\ ((l_1.\mathit{put}\ v_1\ s_1) \cdot (l_2.\mathit{put}\ v_2\ s_2)) && \text{by definition } (l_1 \sim l_2).\mathit{put} \\
={} & (l_2.\mathit{get}\ (l_2.\mathit{put}\ v_2\ s_2)) \cdot (l_1.\mathit{get}\ (l_1.\mathit{put}\ v_1\ s_2)) && \text{by definition } (l_1 \sim l_2).\mathit{get} \\
& \quad \text{with } S_1 \cdot^! S_2 \text{ and } \mathrm{cod}(l_1.\mathit{put}) = S_1 \text{ and } \mathrm{cod}(l_2.\mathit{put}) = S_2 \\
={} & v_2 \cdot v_1 && \text{by PUTGET for } l_2 \text{ and } l_1 \\
={} & v && \text{by definition } v_2 \text{ and } v_1
\end{aligned}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof of PUTGET. □

$$\boxed{\dfrac{\llbracket E \rrbracket \cdot^! \llbracket E \rrbracket}{\mathit{merge}\ E \in (\llbracket E \rrbracket \cdot \llbracket E \rrbracket) \Longleftrightarrow \llbracket E \rrbracket}}$$

2.3.29 Lemma: Let E be a regular expression such that $(\llbracket E \rrbracket \cdot^! \llbracket E \rrbracket)$. Then *merge* E is a basic lens in $(\llbracket E \rrbracket \cdot \llbracket E \rrbracket) \Longleftrightarrow \llbracket E \rrbracket$.

Proof: We prove each basic lens law separately.

▶ **GetPut**: Let $s \in (\llbracket E \rrbracket \cdot \llbracket E \rrbracket)$. As $S_1 \cdot^! S_2$ there exist unique strings e_1 and e_2 belonging to $\llbracket E \rrbracket$ such that $s = e_1 \cdot e_2$. We calculate as follows

$$\begin{aligned}
& (\mathit{merge}\ E).\mathit{put}\ ((\mathit{merge}\ E).\mathit{get}\ s)\ s \\
={} & (\mathit{merge}\ E).\mathit{put}\ ((\mathit{merge}\ E).\mathit{get}\ (e_1 \cdot e_2))\ (e_1 \cdot e_2) && \text{by definition } e_1 \text{ and } e_2 \\
={} & (\mathit{merge}\ E).\mathit{put}\ e_1\ (e_1 \cdot e_2) && \text{by definition } (\mathit{merge}\ E).\mathit{get} \\
={} & \begin{cases} e_1 \cdot e_1 \text{ if } e_1 = e_2 \\ e_1 \cdot e_2 \text{ otherwise} \end{cases} && \text{by definition } (\mathit{merge}\ E).\mathit{put} \\
={} & e_1 \cdot e_2 && \text{using the given equalities} \\
={} & s && \text{by definition } e_1 \text{ and } e_2
\end{aligned}$$

and obtain the required equality.

▶ **PutGet:** Let $v \in [\![E]\!]$ and $s \in ([\![E]\!]\cdot[\![E]\!])$. As $[\![E]\!]\cdot^{!}[\![E]\!]$ there exist unique strings e_1 and e_2 belonging to $[\![E]\!]$ such that $s = e_1{\cdot}e_2$. Using these facts, we calculate as follows

$$\begin{array}{ll}
\ (\mathit{merge}\ E).\mathit{get}\ ((\mathit{merge}\ E).\mathit{put}\ v\ s) & \\
= (\mathit{merge}\ E).\mathit{get}\ ((\mathit{merge}\ E).\mathit{put}\ v\ (e_1{\cdot}e_2)) & \text{by definition } e_1 \text{ and } e_2 \\
= \begin{cases} (\mathit{merge}\ E).\mathit{get}\ (v{\cdot}v) & \text{if } e_1 = e_2 \\ (\mathit{merge}\ E).\mathit{get}\ (v{\cdot}e_2) & \text{otherwise} \end{cases} & \text{by definition } (\mathit{merge}\ E).\mathit{put} \\
= v & \text{by definition } (\mathit{merge}\ E).\mathit{get}
\end{array}$$

and obtain the required equality.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{[\![E]\!]\cdot^{!}[\![E]\!]}{\mathit{dup}\ E \in [\![E]\!] \Longleftrightarrow \{e_1{\cdot}e_2 \in ([\![E]\!]\cdot[\![E]\!]) \mid e_1 = e_2\}}}$$

2.3.30 Lemma: Let E be a regular expression such that $[\![E]\!]\cdot^{!}[\![E]\!]$. Then *dup* E is a basic lens in $[\![E]\!] \Longleftrightarrow \{e_1{\cdot}e_2 \in ([\![E]\!]\cdot[\![E]\!]) \mid e_1 = e_2\}$.

Proof: We prove each basic lens law separately.

▶ **GetPut:** Let $e \in [\![E]\!]$. We calculate as follows

$$\begin{array}{ll}
\ (\mathit{dup}\ E).\mathit{put}\ ((\mathit{dup}\ E).\mathit{get}\ e)\ e & \\
= (\mathit{dup}\ E).\mathit{put}\ (e{\cdot}e)\ e & \text{by definition } (\mathit{dup}\ E).\mathit{get} \\
= e & \text{by definition } (\mathit{dup}\ E).\mathit{put}
\end{array}$$

and obtain the required equality.

▶ **PutGet:** Let $v \in \{e_1{\cdot}e_2 \in [\![E]\!]\cdot[\![E]\!] \mid e_1 = e_2\}$ and $e \in [\![E]\!]$ be strings. As $[\![E]\!]\cdot^{!}[\![E]\!]$ there exist unique strings e_1 and e_2 such that $v = e_1{\cdot}e_2$. Also, by the type of v we have that $e_1 = e_2$. Using these facts, we calculate as follows

$$\begin{array}{ll}
\ (\mathit{dup}\ E).\mathit{get}\ ((\mathit{dup}\ E).\mathit{put}\ v\ s) & \\
= (\mathit{dup}\ E).\mathit{get}\ ((\mathit{dup}\ E).\mathit{put}\ (e_1{\cdot}e_2)\ e) & \\
\ \text{by definition } e_1 \text{ and } e_2 & \\
= (\mathit{dup}\ E).\mathit{get}\ e_1 & \text{by definition } (\mathit{dup}\ E).\mathit{put} \\
= e_1{\cdot}e_1 & \text{by definition } (\mathit{dup}\ E).\mathit{get} \\
= e_1{\cdot}e_2 & \text{by } e_1 = e_2 \\
= v & \text{by definition } e_1 \text{ and } e_2
\end{array}$$

and obtain the required equality.

▶ **CreateGet:** Similar to the proof of PutGet. □

Quotient Lens Proofs

$$\boxed{\dfrac{l \in S \Longleftrightarrow V}{\mathit{lift}\ l \in S/{=} \Longleftrightarrow V/{=}}}$$

3.2.1 Lemma: Let $l \in S \Longleftrightarrow V$ be a basic lens. Then *lift* l is a quotient lens in $S/{=} \Longleftrightarrow V/{=}$.

Proof: We prove each quotient lens law separately.

▶ **GetEquiv**: Let s and s' be strings in S such that $s = s'$. We immediately have $(\mathit{lift}\ l).\mathit{get}\ s = (\mathit{lift}\ l).\mathit{get}\ s'$.

▶ **PutEquiv**: Let v and v' be strings in V such that $v = v'$ and, likewise, let s and s' be strings in S such that $s = s'$. We immediately have $(\mathit{lift}\ l).\mathit{put}\ v\ s = (\mathit{lift}\ l).\mathit{get}\ v'\ s'$.

▶ **CreateEquiv**: Let v and v' be strings in V such that $v = v'$. We immediately have $(\mathit{lift}\ l).\mathit{create}\ v = (\mathit{lift}\ l).\mathit{create}\ v'$.

▶ **GetPut**: Immediate by the basic lens version of GetPut for l.

▶ **PutGet**: Immediate by the basic lens version of PutGet for l.

▶ **CreateGet**: Immediate by the basic lens version of CreateGet for l. □

$$\frac{\begin{array}{c} q \in S \leftarrow\!\!\!\!\twoheadrightarrow U/{\sim_U} \\ l \in U/{\sim_U} \Longleftrightarrow V/{\sim_V} \\ \sim_S \triangleq \{(s, s') \in S \times S \mid q.\mathit{canonize}\ s \sim_U q.\mathit{canonize}\ s'\} \end{array}}{\mathit{lquot}\ q\ l \in S/{\sim_S} \Longleftrightarrow V/{\sim_V}}$$

3.2.3 Lemma: Let q be a canonizer $S \leftarrow\!\!\!\!\twoheadrightarrow U/{\sim_U}$ and let l be a quotient lens in $U/{\sim_U} \Longleftrightarrow V/{\sim_V}$. Then $\mathit{lquot}\ q\ l$ is a quotient lens in $S/{\sim_S} \Longleftrightarrow V/{\sim_V}$ where $s \sim_S s'$ if and only if $q.\mathit{canonize}\ s \sim_U q.\mathit{canonize}\ s'$.

Proof: We prove each quotient lens law separately. To shorten the proof, abbreviate $\mathit{lquot}\ q\ l$ as k.

▶ **GetEquiv**: Let s and s' be strings in S such that $s \sim_S s'$. By the definition of $\sim_S$ we have:

$$q.\mathit{canonize}\ s \sim_U q.\mathit{canonize}\ s'$$

Using this equivalence, we calculate as follows

$$\begin{array}{rll} & k.\mathit{get}\ s & \\ = & l.\mathit{get}\ (q.\mathit{canonize}\ s) & \text{by definition } k.\mathit{get} \\ \sim_V & l.\mathit{get}\ (q.\mathit{canonize}\ s') & \text{by GetEquiv for } l \\ = & k.\mathit{get}\ s' & \text{by definition } k.\mathit{get} \end{array}$$

and obtain the required equivalence.

▶ **PutEquiv**: Let v and v' be strings in V such that $v \sim_V v'$. Likewise, let s and s' be strings in S such that $s \sim_S s'$. We will prove that

$$k.\mathit{put}\ v\ s \sim_S k.\mathit{put}\ v'\ s'$$

by showing that:

$$q.\mathit{canonize}\ (k.\mathit{put}\ v\ s) \sim_U q.\mathit{canonize}\ (k.\mathit{put}\ v'\ s')$$

By the definition of $\sim_S$ we have:

$$q.\mathit{canonize}\ s \sim_U q.\mathit{canonize}\ s'$$

Using this equivalence, we calculate as follows:

$$\begin{array}{rll} & q.\mathit{canonize}\ (k.\mathit{put}\ v\ s) & \\ = & q.\mathit{canonize}\ (q.\mathit{choose}\ (l.\mathit{put}\ v\ (q.\mathit{canonize}\ s))) & \text{by definition } k.\mathit{put} \\ \sim_U & l.\mathit{put}\ v\ (q.\mathit{canonize}\ s) & \text{by ReCanonize for } q \\ \sim_U & l.\mathit{put}\ v'\ (q.\mathit{canonize}\ s') & \text{by PutEquiv for } l \\ \sim_U & q.\mathit{canonize}\ (q.\mathit{choose}\ (l.\mathit{put}\ v'\ (q.\mathit{canonize}\ s'))) & \text{by ReCanonize for } q \\ = & q.\mathit{canonize}\ (k.\mathit{put}\ v'\ s') & \text{by definition } k.\mathit{put} \end{array}$$

The required equivalence follows by transitivity. We will silently use elementary facts about equivalence relations such as the transitivity of $\sim_U$ throughout this dissertation.

▶ **CreateEquiv**: Similar to the proof of PutEquiv.

▶ **GetPut**: Let $s \in S$ be a string. We will prove that

$$k.put\ (k.get\ s)\ s \sim_S s$$

by showing that:

$$q.canonize\ (k.put\ (k.get\ s)\ s) \sim_U q.canonize\ s$$

To shorten the proof, abbreviate $q.canonize\ s$ as u. We calculate as follows

$$\begin{array}{ll}
\ q.canonize\ (k.put\ (k.get\ s)\ s) & \\
= q.canonize\ (q.choose\ (l.put\ (k.get\ s)\ u)) & \text{by definition } k.put \text{ and } u \\
\sim_U l.put\ (k.get\ s)\ u & \text{by ReCanonize for } q \\
= l.put\ (l.get\ u)\ u & \text{by definition } k.get \text{ and } u \\
\sim_U u & \text{by GetPut for } l \\
= q.canonize\ s & \text{by definition } u
\end{array}$$

and obtain the required equivalence.

▶ **PutGet**: Let $v \in V$ and $s \in S$ be strings. To shorten the proof, abbreviate $q.canonize\ s$ as u. By ReCanonize for q we have:

$$q.canonize\ (q.choose\ (l.put\ v\ u)) \sim_U l.put\ v\ u$$

Using this fact, we calculate as follows

$$\begin{array}{ll}
\ k.get\ (k.put\ v\ s) & \\
= k.get\ (q.choose\ (l.put\ v\ u)) & \text{by definition } k.put \text{ and } u \\
= l.get\ (q.canonize\ (q.choose\ (l.put\ v\ u))) & \text{by definition } k.get \\
\sim_V l.get\ (l.put\ v\ u) & \text{by ReCanonize for } q \\
& \text{and GetEquiv for } l \\
\sim_V v & \text{by PutGet for } l
\end{array}$$

and obtain the required equivalence.

▶ **CreateGet**: Similar to the proof of PutGet. □

$$\boxed{\frac{\begin{array}{c} l \in S/{\sim_S} \Longleftrightarrow U/{\sim_U} \\ q \in V \leftarrow\!\!\!\!\twoheadrightarrow U/{\sim_U} \\ \sim_V \triangleq \{(v, v') \in V \times V \mid q.canonize\ v \sim_U q.canonize\ v'\} \end{array}}{rquot\ l\ q \in S/{\sim_S} \Longleftrightarrow V/{\sim_V}}}$$

3.2.4 Lemma: Let l be a quotient lens in $S/{\sim_S} \Longleftrightarrow U/{\sim_U}$ and q a canonizer in $V \leftarrow\!\!\!\!\twoheadrightarrow U/{\sim_U}$. Then $rquot\ l\ q$ is a quotient lens in $S/{\sim_S} \Longleftrightarrow V/{\sim_V}$ where $v {\sim_V} v'$ if and only if $q.canonize\ v \sim_U q.canonize\ v'$.

Proof: We prove each quotient lens law separately. To lighten the notation, we will abbreviate $rquot\ l\ q$ as k.

▶ **GetEquiv**: Let $s \in S$ and $s' \in S$ be strings such that $s \sim_S s'$. We will prove that

$$k.get\ s \sim_V k.get\ s'$$

by showing that:

$$q.\mathit{canonize}\ (k.\mathit{get}\ s) \sim_U q.\mathit{canonize}\ (k.\mathit{get}\ s')$$

We calculate as follows

$$\begin{array}{ll}
\quad q.\mathit{canonize}\ (k.\mathit{get}\ s) & \\
= q.\mathit{canonize}\ (q.\mathit{choose}\ (l.\mathit{get}\ s)) & \text{by definition } k.\mathit{get} \\
\sim_U l.\mathit{get}\ s & \text{by ReCanonize for } q \\
\sim_U l.\mathit{get}\ s' & \text{by GetEquiv for } l \\
\sim_U q.\mathit{canonize}\ (q.\mathit{choose}\ (l.\mathit{get}\ s')) & \text{by ReCanonize for } q \\
= q.\mathit{canonize}\ (k.\mathit{get}\ s') & \text{by definition } k.\mathit{get}
\end{array}$$

and obtain the required equivalence.

▶ **PutEquiv**: Let $v \in V$ and $v' \in V$ be strings such that $v \sim_V v'$ and similarly, let $s \in S$ and $s' \in S$ be strings such that $s \sim_S s'$. By the definition of $\sim_V$, we have that:

$$q.\mathit{canonize}\ v \sim_U q.\mathit{canonize}\ v'$$

Using this equivalence, we calculate as follows

$$\begin{array}{ll}
\quad k.\mathit{put}\ v\ s & \\
= l.\mathit{put}\ (q.\mathit{canonize}\ v)\ s & \text{by definition } k.\mathit{put} \\
\sim_S l.\mathit{put}\ (q.\mathit{canonize}\ v')\ s' & \text{by PutEquiv for } l \\
= k.\mathit{put}\ v'\ s' & \text{by definition } k.\mathit{put}
\end{array}$$

and obtain the required equivalence.

▶ **CreateEquiv**: Similar to the proof of PutGet.

▶ **GetPut**: Let $s \in S$. We calculate as follows

$$\begin{array}{ll}
\quad k.\mathit{put}\ (k.\mathit{get}\ s)\ s & \\
= l.\mathit{put}\ (q.\mathit{canonize}\ (q.\mathit{choose}\ (l.\mathit{get}\ s)))\ s & \text{by definition } k.\mathit{get} \text{ and } k.\mathit{put} \\
\sim_S l.\mathit{put}\ (l.\mathit{get}\ s)\ s & \text{by ReCanonize for } q \\
 & \text{and PutEquiv for } l \\
\sim_S s & \text{by GetPut for } l
\end{array}$$

and obtain the required equivalence.

▶ **PutGet**: Let $v \in V$ and $s \in S$. We will show that

$$k.\mathit{get}\ (k.\mathit{put}\ v\ s) \sim_V v$$

by showing that:

$$q.\mathit{canonize}\ (k.\mathit{get}\ (k.\mathit{put}\ v\ s)) \sim_U q.\mathit{canonize}\ v$$

We calculate as follows

$$\begin{array}{ll}
\quad q.\mathit{canonize}\ (k.\mathit{get}\ (k.\mathit{put}\ v\ s)) & \\
= q.\mathit{canonize}\ (q.\mathit{choose}\ (l.\mathit{get}\ (k.\mathit{put}\ v\ s))) & \text{by definition } k.\mathit{get} \\
\sim_U l.\mathit{get}\ (k.\mathit{put}\ v\ s) & \text{by ReCanonize for } q \\
= l.\mathit{get}\ (l.\mathit{put}\ (q.\mathit{canonize}\ v)\ s) & \text{by definition } k.\mathit{put} \\
\sim_U (q.\mathit{canonize}\ v) & \text{By PutGet for } l
\end{array}$$

and obtain the desired equivalence.

▶ **CreateGet**: Similar to the proof of PutGet. □

$$\boxed{\begin{array}{c} \sim_U \text{ is a refinement of } \sim_{U'} \\ q \in V \mathrel{\leftarrow\!\!\boxminus\!\!\rightarrow} U/\!\sim_U \\ \hline q \in V \mathrel{\leftarrow\!\!\boxminus\!\!\rightarrow} U/\!\sim_{U'} \end{array}}$$

3.2.5 Lemma: Let $q \in V \mathrel{\leftarrow\!\!\boxminus\!\!\rightarrow} U/\!\sim_U$ be a canonizer and let $\sim_{U'}$ be an equivalence relation on U such that $\sim_{U'}$ is a refinement of $\sim_U$. Then q is also a canonizer in $V \mathrel{\leftarrow\!\!\boxminus\!\!\rightarrow} U/\!\sim_{U'}$.

Proof: We prove the canonizer law directly.

▶ **ReCanoniz:** Let $u \in U$ be a string. As $q \in V \mathrel{\leftarrow\!\!\boxminus\!\!\rightarrow} U/\!\sim_U$ we have $q.\mathit{canonize}\ (q.\mathit{choose}\ u) \sim_U u$. Since $\sim_U$ refines $\sim_{U'}$ we immediately have that $q.\mathit{canonize}\ (q.\mathit{choose}\ u) \sim_{U'} u$, as required. □

$$\boxed{\begin{array}{c} l_1 \in S/\!\sim_S \Longleftrightarrow U/\!\sim_U \\ l_2 \in U/\!\sim_U \Longleftrightarrow V/\!\sim_V \\ \hline l_1;l_2 \in S/\!\sim_S \Longleftrightarrow V/\!\sim_V \end{array}}$$

3.2.6 Lemma: Let $l_1 \in S/\!\sim_S \Longleftrightarrow U/\!\sim_U$ and $l_2 \in U/\!\sim_U \Longleftrightarrow V/\!\sim_V$ be quotient lenses. Then $l_1;l_2$ is a quotient lens in $S/\!\sim_S \Longleftrightarrow V/\!\sim_V$.

Proof: We prove each quotient lens law separately.

▶ **GetEquiv:** Let s and s' be strings in S such that $s \sim_S s'$. We calculate as follows

$$\begin{array}{rll} & (l_1;l_2).\mathit{get}\ s & \\ = & l_2.\mathit{get}\ (l_1.\mathit{get}\ s) & \text{by definition } (l_1;l_2).\mathit{get} \\ \sim_V & l_2.\mathit{get}\ (l_1.\mathit{get}\ s') & \text{by } \textsc{GetEquiv} \text{ for } l_1 \text{ and } l_2 \\ = & (l_1;l_2).\mathit{get}\ s' & \text{by definition } (l_1;l_2).\mathit{get} \end{array}$$

and obtain the required equivalence.

▶ **PutEquiv:** Let v and v' be strings in V such that $v \sim_V v'$ and likewise let s and s' be strings in S such that $s \sim_S s'$. We calculate as follows

$$\begin{array}{rll} & (l_1;l_2).\mathit{put}\ v\ s & \\ = & l_1.\mathit{put}\ (l_2.\mathit{put}\ s\ (l_1.\mathit{get}\ s))\ s & \text{by definition } (l_1;l_2).\mathit{put} \\ \sim_S & l_1.\mathit{put}\ (l_2.\mathit{put}\ v'\ (l_1.\mathit{get}\ s'))\ s' & \text{by } \textsc{GetEquiv} \text{ for } l_1 \\ & & \text{and } \textsc{PutEquiv} \text{ for } l_1 \text{ and } l_2 \\ = & (l_1;l_2).\mathit{put}\ v'\ s' & \text{by definition } (l_1;l_2).\mathit{put} \end{array}$$

and obtain the required equivalence.

▶ **CreateEquiv:** Similar to the proof of PutGet.

▶ **GetPut:** Let S be a string in S. We calculate as follows

$$\begin{array}{rll} & (l_1;l_2).\mathit{put}\ ((l_1;l_2).\mathit{get}\ s)\ s & \\ = & l_1.\mathit{put}\ (l_2.\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (l_1.\mathit{get}\ s))\ s & \text{by definition } (l_1;l_2).\mathit{get} \\ & & \text{and } (l_1;l_2).\mathit{put} \\ \sim_S & l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ s & \text{by } \textsc{GetPut} \text{ for } l_2 \\ & & \text{and } \textsc{PutEquiv} \text{ for } l_1 \\ \sim_S & s & \text{by } \textsc{GetPut} \text{ for } l \end{array}$$

and obtain the required equivalence.

▶ **PutGet:** Let v be a string in V and let s be a string in S. We calculate as follows

$$\begin{array}{lll} & (l_1;l_2).\mathit{get}\ ((l_1;l_2).\mathit{put}\ v\ s) & \\ = & l_2.\mathit{get}\ (l_1.\mathit{get}\ (l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s)) & \text{by definition } (l_1;l_2).\mathit{get} \\ & & \text{and } (l_1;l_2).\mathit{put} \\ \sim_V & l_2.\mathit{get}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s)) & \text{by PUTGET for } l_1 \\ & & \text{and GETEQUIV for } l_2 \\ \sim_V & v & \text{by PUTGET for } l_2 \end{array}$$

and obtain the required equivalence.

▶ **CreateGet:** Similar to the proof of PUTGET. □

$$\boxed{\dfrac{l \in S/{\sim_S} \Longleftrightarrow U/{\sim_U}}{\mathit{canonizer\_of\_lens}\ l \in S \mathrel{\leftrightarrow\!\!\!\!\!\rhd} U/{\sim_U}}}$$

3.2.7 Lemma: Let l be a quotient lens in $S/{\sim_S} \Longleftrightarrow U/{\sim_U}$. Then the canonizer $\mathit{canonizer\_of\_lens}\ l$ is in $S \mathrel{\leftrightarrow\!\!\!\!\!\rhd} U/{\sim_U}$.

Proof: We prove the canonizer law directly. To shorten the proof, we will abbreviate the canonizer $\mathit{canonizer\_of\_lens}\ l$ as q.

▶ **ReCanonize:**
Let $u \in U$ be a string. We calculate as follows

$$\begin{array}{lll} & q.\mathit{canonize}\ (q.\mathit{choose}\ u) & \\ = & l.\mathit{get}\ (l.\mathit{create}\ u) & \text{by definition } q.\mathit{canonize} \text{ and } q.\mathit{choose} \\ \sim_U & u & \text{by CREATEGET for } l \end{array}$$

and obtain the required equivalence. □

$$\boxed{\dfrac{\begin{array}{ll} l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}} & S_1 \cdot^{!} S_2 \\ l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}} & V_1 \cdot^{!} V_2 \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2)/({\sim_{S_1}} \cdot {\sim_{S_2}}) \Longleftrightarrow (V_1 \cdot V_2)/({\sim_{V_1}} \cdot {\sim_{V_2}})}}$$

3.2.9 Lemma: Let $l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}}$ and $l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}}$ be quotient lenses such that $S_1 \cdot^{!} S_2$ and $V_1 \cdot^{!} V_2$. Then $l_1 \cdot l_2$ is a quotient lens in $(S_1 \cdot S_2)/({\sim_{S_1}} \cdot {\sim_{S_2}}) \Longleftrightarrow (V_1 \cdot V_2)/({\sim_{V_1}} \cdot {\sim_{V_2}})$.

Proof: We prove each quotient lens law separately. To shorten the proof, abbreviate ${\sim_{S_1}} \cdot {\sim_{S_2}}$ as $\sim_S$ and ${\sim_{V_1}} \cdot {\sim_{V_2}}$ as $\sim_V$.

▶ **GetEquiv:** Let $s = s_1 \cdot s_2$ and $s' = s'_1 \cdot s'_2$ be strings in $S_1 \cdot S_2$. By the definition of $\sim_S$ we have that $s_1 \sim_{S_1} s'_1$ and $s_2 \sim_{S_2} s'_2$. Using these facts, we calculate as follows

$$\begin{array}{lll} & (l_1 \cdot l_2).\mathit{get}\ (s_1 \cdot s_2) & \\ = & (l_1.\mathit{get}\ s_1) \cdot (l_2.\mathit{get}\ s_2) & \text{by definition } (l_1 \cdot l_2).\mathit{get} \\ \sim_V & (l_1.\mathit{get}\ s'_1) \cdot (l_2.\mathit{get}\ s'_2) & \text{by GETEQUIV for } l_1 \text{ and } l_2 \\ & & \text{and definition } \sim_V \\ = & (l_1 \cdot l_2).\mathit{get}\ (s'_1 \cdot s'_2) & \text{by definition } (l_1 \cdot l_2).\mathit{get} \end{array}$$

and obtain the required equivalence.

▶ **PutEquiv:** Let $v = v_1{\cdot}v_2$ and $v' = v'_1{\cdot}v'_2$ be strings in $V_1{\cdot}V_2$ and let $s = s_1{\cdot}s_2$ and $s' = s'_1{\cdot}s'_2$ be strings in $S_1{\cdot}S_2$. By the definitions of $\sim_V$ and $\sim_S$ we have $v_1 \sim_{V_1} v'_1$ and $v_2 \sim_{V_2} v'_2$ and $s_1 \sim_{S_1} s'_1$ and $s_2 \sim_{S_2} s'_2$. Using these facts, we calculate as follows

$$\begin{aligned}
& (l_1{\cdot}l_2).\mathit{put}\ (v_1{\cdot}v_2)\ (s_1{\cdot}s_2) & \\
= {} & (l_1.\mathit{put}\ v_1\ s_1)\cdot(l_2.\mathit{put}\ v_2\ s_2) & \text{by definition } (l_1{\cdot}l_2).\mathit{put} \\
\sim_S {} & (l_1.\mathit{put}\ v'_1\ s'_1)\cdot(l_2.\mathit{put}\ v'_2\ s'_2) & \text{by PutEquiv for } l_1 \text{ and } l_2 \\
& & \text{and definition } \sim_S \\
= {} & (l_1{\cdot}l_2).\mathit{put}\ (v'_1{\cdot}v'_2)\ (s'_1{\cdot}s'_2) & \text{by definition } (l_1{\cdot}l_2).\mathit{put}
\end{aligned}$$

and obtain the required equivalence.

▶ **CreateEquiv:** Similar to the proof of PutEquiv.

▶ **GetPut:** Let $s = s_1{\cdot}s_2$ be a string in $S_1{\cdot}S_2$. We calculate as follows

$$\begin{aligned}
& (l_1{\cdot}l_2).\mathit{put}\ ((l_1{\cdot}l_2).\mathit{get}\ (s_1{\cdot}s_2))\ (s_1{\cdot}s_2) & \\
= {} & (l_1{\cdot}l_2).\mathit{put}\ ((l_1.\mathit{get}\ s_1){\cdot}(l_2.\mathit{get}\ s_2))\ (s_1{\cdot}s_2) & \text{by definition } (l_1{\cdot}l_2).\mathit{get} \\
= {} & (l_1.\mathit{put}\ (l_1.\mathit{get}\ s_1)\ s_1){\cdot}(l_2.\mathit{put}\ (l_2.\mathit{get}\ s_2)\ s_2) & \text{by definition } (l_1{\cdot}l_2).\mathit{put} \\
& \quad \text{with } V_1{\cdot}^!V_2 \text{ and } \mathrm{cod}(l_1.\mathit{get}) = V_1 \text{ and } \mathrm{cod}(l_2.\mathit{get}) = V_2 & \\
\sim_S {} & s_1 \cdot s_2 & \text{By GetPut for } l_1 \text{ and } l_2 \\
& & \text{and definition } \sim_S
\end{aligned}$$

and obtain the required equivalence.

▶ **PutGet:** Let $v = v_1{\cdot}v_2$ be a string in $V_1{\cdot}V_2$ and let $s = s_1{\cdot}s_2$ be a string in $S_1{\cdot}S_2$. We calculate as follows

$$\begin{aligned}
& (l_1{\cdot}l_2).\mathit{get}\ ((l_1{\cdot}l_2).\mathit{put}\ (v_1{\cdot}v_2)\ (s_1{\cdot}s_2)) & \\
= {} & (l_1{\cdot}l_2).\mathit{get}\ ((l_1.\mathit{put}\ v_1\ s_1)\cdot(l_2.\mathit{put}\ v_2\ s_2)) & \text{by definition } (l_1{\cdot}l_2).\mathit{put} \\
= {} & (l_1.\mathit{get}\ (l_1.\mathit{put}\ v_1\ s_1))\cdot(l_2.\mathit{get}\ (l_2.\mathit{put}\ v_2\ s_2)) & \text{by definition } (l_1{\cdot}l_2).\mathit{get} \\
& \quad \text{with } S_1{\cdot}^!S_2 \text{ and } \mathrm{cod}(l_1.\mathit{put}) = S_1 \text{ and } \mathrm{cod}(l_2.\mathit{put}) = S_2 & \\
\sim_V {} & v_1 \cdot v_2 & \text{By PutGet for } l_1 \text{ and } l_2 \\
& & \text{and definition } \sim_V
\end{aligned}$$

and obtain the required equivalence.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{S^{!*} \qquad V^{!*} \\ l \in S/{\sim_S} \Longleftrightarrow V/{\sim_V}}{l^* \in S^*/{\sim_S}^* \Longleftrightarrow V^*/{\sim_V}^*}}$$

3.2.11 Lemma: Let $l \in S/{\sim_S} \Longleftrightarrow V/{\sim_V}$ be a quotient lens such that $S^{!*}$ and $V^{!*}$. Then l^* is a quotient lens in $S^*/{\sim_S}^* \Longleftrightarrow V^*/{\sim_V}^*$.

Proof: We prove each quotient lens law separately.

▶ **GetEquiv:** Let $s = s_1 \cdots s_n$ and $s' = s'_1 \cdots s'_m$ be strings in S^* such that $s \sim_S{}^* s'$. By the definition of $\sim_S{}^*$ we have $n = m$ and $s_i \sim_S s'_i$ for i from 1 to n. Using these facts, we calculate as follows

$$\begin{aligned}
& l^*.\mathit{get}\ (s_1 \cdots s_n) & \\
= {} & (l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n) & \text{by definition } l^*.\mathit{get} \\
\sim_V{}^* {} & (l.\mathit{get}\ s'_1) \cdots (l.\mathit{get}\ s'_n) & \text{by GetEquiv for } l \\
= {} & l^*.\mathit{get}\ (s'_1 \cdots s'_n) & \text{by definition } l^*.\mathit{get}
\end{aligned}$$

and obtain the required equivalence.

▶ **PutEquiv:** Let $v = v_1 \cdots v_n$ and $v' = v'_1 \cdots v'_m$ be strings in V^* such that $v \sim_V{}^* v'$ and let $s = s_1 \cdots s_o$ and $s' = s'_1 \cdots s_p$ be strings in S^* such that $s \sim_S{}^* s'$. By the definition of $\sim_V{}^*$ we have $m = n$ and $v_i \sim_V v'_i$ for i from 1 to n and also $o = p$ and $s_j \sim_S s'_j$ for j from 1 to o. Using these facts, we calculate as follows

$$\begin{array}{ll}
= l^*.\textit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_o) & \\
= s''_1 \cdots s''_n & \text{by definition } l^*.\textit{put} \\
\quad \text{where } s''_i = \begin{cases} l.\textit{put}\ v_i\ s_i \text{ for } i \in \{1, \ldots, \max(n, o)\} \\ l.\textit{create}\ v_i \text{ for } i \in \{o+1, \ldots, n\} \end{cases} & \\
\sim_S{}^*\ s'''_1 \cdots s'''_n & \text{by } \textsc{PutEquiv} \text{ for } l \text{ } (n \text{ times}) \\
\quad \text{where } s'''_i = \begin{cases} l.\textit{put}\ v'_i\ s'_i \text{ for } i \in \{1, \ldots, \max(n, o)\} \\ l.\textit{create}\ v'_i \text{ for } i \in \{o+1, \ldots, n\} \end{cases} & \\
= l^*.\textit{put}\ (v'_1 \cdots v'_n)\ (s'_1 \cdots s'_o) & \text{by definition } l^*.\textit{put}
\end{array}$$

and obtain the required equivalence.

▶ **CreateEquiv:** Similar to the proof of PutEquiv.

▶ **GetPut:** Let $s = s_1 \cdots s_n$ be a string in $\in S^*$. We calculate as follows

$$\begin{array}{ll}
\quad l^*.\textit{put}\ (l^*.\textit{get}\ (s_1 \cdots s_n))\ (s_1 \cdots s_n) & \\
= l^*.\textit{put}\ ((l.\textit{get}\ s_1) \cdots (l.\textit{get}\ s_n))\ (s_1 \cdots s_n) & \text{by definition } l^*.\textit{get} \\
= (l.\textit{put}\ (l.\textit{get}\ s_1)\ s_1) \cdots (l.\textit{put}\ (l.\textit{get}\ s_n)\ s_n) & \text{by definition } l^*.\textit{put} \\
\quad \text{with } V^{!*} \text{ and } \mathrm{cod}(l.\textit{get}) = V & \\
\sim_S{}^*\ s_1 \cdots s_n & \text{by } \textsc{GetPut} \text{ for } l
\end{array}$$

and obtain the required equivalence.

▶ **PutGet:** Let $v = v_1 \cdots v_n$ be a string in V^* and let $s = s_1 \cdots s_m$ be a string in S^*. We calculate as follows

$$\begin{array}{ll}
\quad l^*.\textit{get}\ (l^*.\textit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) & \\
= l^*.\textit{get}\ (s'_1 \cdots s'_n) & \text{by definition } l^*.\textit{put} \\
\quad \text{where } s'_i = \begin{cases} l.\textit{put}\ v_i\ s_i \text{ for } i \in \{1, \ldots, \max(m, n)\} \\ l.\textit{create}\ v_i \text{ for } i \in \{m+1, \ldots, n\} \end{cases} & \\
= (l.\textit{get}\ s'_1) \cdots (l.\textit{get}\ s'_n) & \text{by definition } l^*.\textit{get} \\
\quad \text{with } S^{!*} \text{ and } \mathrm{cod}(l.\textit{put}) = S & \\
\sim_V{}^*\ v_1 \cdots v_n & \text{by } \textsc{PutGet} \text{ and } \textsc{CreateGet} \text{ for } l
\end{array}$$

and obtain the required equivalence.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{\begin{array}{c} S_1 \cap S_2 = \emptyset \\ l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}} \\ l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}} \\ \forall v, v' \in V_1 \cap V_2.\ v \sim_{V_1} v' \text{ if and only if } v \sim_{V_2} v' \end{array}}{l_1 \,|\, l_2 \in (S_1 \cup S_2)/({\sim_{S_1}} \cup {\sim_{S_2}}) \Longleftrightarrow (V_1 \cup V_2)/({\sim_{V_1}} \cup {\sim_{V_2}})}}$$

3.2.12 Lemma: Let $l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}}$ and $l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}}$ be quotient lenses such that $S_1 \cap S_2 = \emptyset$ and for every v and v' in $V_1 \cap V_2$ we have $v \sim_{V_1} v'$ if and only if $v \sim_{V_2} v'$. Then the quotient lens $l_1 \,|\, l_2$ is in $(S_1 \cup S_2)/({\sim_{S_1}} \cup {\sim_{S_2}}) \Longleftrightarrow (V_1 \cup V_2)/({\sim_{V_1}} \cup {\sim_{V_2}})$.

Proof: We prove each quotient lens law separately. To shorten the proof abbreviate $\sim_{S_1} \cup \sim_{S_2}$ as $\sim_S$ and $\sim_{V_1} \cup \sim_{V_2}$ as $\sim_V$.

▶ **GetEquiv:** Let s and s' be strings in $S_1 \cup S_2$ such that $s \sim_S s'$. As $S_1 \cap S_2 = \emptyset$, we either have $s \in S_1$ and $s' \in S_1$ and $s \sim_{S_1} s'$ or $s \in S_2$ and $s' \in S_2$ and $s \sim_{S_2} s'$. We analyze each case separately.

Case s and s' in S_1: We calculate as follows

$$\begin{aligned}
&(l_1 \,|\, l_2).\mathit{get}\ s && \\
&= l_1.\mathit{get}\ s && \text{by definition } (l_1 \,|\, l_2).\mathit{get} \\
&\sim_V l_1.\mathit{get}\ s' && \text{by GetEquiv for } l_1 \\
& && \text{and definition } \sim_V \\
&= (l_1 \,|\, l_2).\mathit{get}\ s' && \text{by definition } (l_1 \,|\, l_2).\mathit{get}
\end{aligned}$$

and obtain the required equivalence.

Case s and s' in S_2: Symmetric to the previous case.

▶ **PutEquiv:** Let v and v' be strings in $V_1 \cup V_2$ such that $v \sim_V v'$ and let s and s' be strings in $S_1 \cup S_2$ such that $s \sim_V s'$. By the conditions on $\sim_{V_1}$ and $\sim_{V_2}$ we either have

- v and v' in $V_1 \cap V_2$ with $v \sim_{V_1} v'$ and $v \sim_{V_2} v'$ or
- v and v' in $V_1 - V_2$ with $v \sim_{V_1} v'$ or
- v and v' in $V_2 - V_1$ with $v \sim_{V_2} v'$.

Similarly, as $(S_1 \cap S_2) = \emptyset$, we either have

- s and s' in S_1 with $s \sim_{S_1} s'$ or
- s and s' in S_2 with $s \sim_{S_2} s'$.

We analyze several cases.

Case v and v' in V_1 and s and s' in S_1: We calculate as follows

$$\begin{aligned}
&(l_1 \,|\, l_2).\mathit{put}\ v\ s && \\
&= l_1.\mathit{put}\ v\ s && \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
&\sim_S l_1.\mathit{put}\ v'\ s' && \text{by PutEquiv for } l_1 \\
& && \text{and definition } \sim_S \\
&= (l_1 \,|\, l_2).\mathit{put}\ v'\ s' && \text{by definition } (l_1 \,|\, l_2).\mathit{put}
\end{aligned}$$

and obtain the required equivalence.

Case v and v' in V_2 and s and s' in S_2: Symmetric to the previous case.

Case v and v' in $V_1 - V_2$ and s and s' in S_2: We calculate as follows

$$\begin{aligned}
&(l_1 \,|\, l_2).\mathit{put}\ v\ s && \\
&= l_1.\mathit{create}\ v\ s && \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
&\sim_S l_1.\mathit{create}\ v'\ s' && \text{by CreateEquiv for } l_1 \\
& && \text{and definition } \sim_S \\
&= (l_1 \,|\, l_2).\mathit{put}\ v'\ s' && \text{by definition } (l_1 \,|\, l_2).\mathit{put}
\end{aligned}$$

and obtain the required equivalence.

Case v and v' in $V_2 - V_1$ and s and s' in S_1: Symmetric to the previous case.

▶ **CreateEquiv:** Similar to the proof of PutEquiv.

▶ **GetPut:** Let s be a string in $S_1 \cup S_2$. We analyze several cases.

Case $s \in S_1$: We calculate as follows

$$\begin{array}{ll}
\quad (l_1 \mid l_2).\mathit{put}\ ((l_1 \mid l_2).\mathit{get}\ s)\ s & \\
= (l_1 \mid l_2).\mathit{put}\ (l_1.\mathit{get}\ s)\ s & \text{by definition } (l_1 \mid l_2).\mathit{get} \\
= l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ s & \text{by definition } (l_1 \mid l_2).\mathit{put} \\
\qquad \text{with } \mathrm{cod}(l_1.\mathit{get}) = V_1 & \\
\sim_S s & \text{by \textsc{GetPut} for } l_1 \\
 & \text{and definition } \sim_S
\end{array}$$

and obtain the required equivalence.

Case $s \in S_2$: Symmetric to the previous case.

▶ **PutGet:** Let v be a string in $V_1 \cup V_2$ and let s be a string in $S_1 \cup S_2$. We analyze several cases.

Case $v \in V_1$ and $s \in S_1$: We calculate as follows

$$\begin{array}{ll}
\quad (l_1 \mid l_2).\mathit{get}\ ((l_1 \mid l_2).\mathit{put}\ v\ s) & \\
= (l_1 \mid l_2).\mathit{get}\ (l_1.\mathit{put}\ v\ s) & \text{by definition } (l_1 \mid l_2).\mathit{put} \\
= l_1.\mathit{get}\ (l_1.\mathit{put}\ v\ s) & \text{by definition } (l_1 \mid l_2).\mathit{get} \\
\qquad \text{with } \mathrm{cod}(l_1.\mathit{put}) = S_1 & \\
\sim_V v & \text{by \textsc{PutGet} for } l_1 \\
 & \text{and definition } \sim_V
\end{array}$$

and obtain the required equivalence.

Case $v \in V_2$ and $s \in S_2$: Symmetric to the previous case.

Case $v \in V_1 - V_2$ and $s \in S_2$: We calculate as follows

$$\begin{array}{ll}
\quad (l_1 \mid l_2).\mathit{get}\ ((l_1 \mid l_2).\mathit{put}\ v\ s) & \\
= (l_1 \mid l_2).\mathit{get}\ (l_1.\mathit{create}\ v) & \text{by definition } (l_1 \mid l_2).\mathit{put} \\
= l_1.\mathit{get}\ (l_1.\mathit{create}\ v) & \text{by definition } (l_1 \mid l_2).\mathit{get} \\
\qquad \text{with } \mathrm{cod}(l_1.\mathit{create}) = S_1 & \\
\sim_V v & \text{by \textsc{CreateGet} for } l_1 \\
 & \text{and definition } \sim_V
\end{array}$$

and obtain the required equivalence.

Case $V \in V_2 - V_1$ and $s \in S_1$: Symmetric to the previous case.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{\begin{array}{ll} l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}} & S_1 \cdot^{!} S_2 \\ l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}} & V_2 \cdot^{!} V_1 \end{array}}{l_1 \sim l_2 \in (S_1 \cdot S_2)/({\sim_{S_1}} \cdot {\sim_{S_2}}) \Longleftrightarrow (V_2 \cdot V_1)/({\sim_{V_2}} \cdot {\sim_{V_1}})}}$$

3.2.13 Lemma: Let $l_1 \in S_1/{\sim_{S_1}} \Longleftrightarrow V_1/{\sim_{V_1}}$ and $l_2 \in S_2/{\sim_{S_2}} \Longleftrightarrow V_2/{\sim_{V_2}}$ be quotient lenses such that $S_1 \cdot^! S_2$ and $V_2 \cdot^! V_1$. Then $l_1 \sim l_2$ is a quotient lens in $(S_1 \cdot S_2)/({\sim_{S_1}} \cdot {\sim_{S_2}}) \Longleftrightarrow (V_2 \cdot V_1)/({\sim_{V_2}} \cdot {\sim_{V_1}})$.

Proof: We prove each quotient lens law separately. To shorten the proof, abbreviate ${\sim_{S_1}} \cdot {\sim_{S_2}}$ as $\sim_S$ and ${\sim_{V_2}} \cdot {\sim_{V_1}}$ as $\sim_V$.

▶ **GetEquiv:** Let $s = s_1 \cdot s_2$ and $s' = s'_1 \cdot s'_2$ be strings in $S_1 \cdot S_2$. By the definition of $\sim_S$ we have that $s_1 \sim_{S_1} s'_1$ and $s_2 \sim_{S_2} s'_2$. Using these facts, we calculate as follows

$$\begin{array}{rll}
 & (l_1 \sim l_2).\textit{get}\ (s_1 \cdot s_2) & \\
= & (l_2.\textit{get}\ s_2) \cdot (l_1.\textit{get}\ s_1) & \text{by definition } (l_1 \sim l_2).\textit{get} \\
\sim_V & (l_2.\textit{get}\ s'_2) \cdot (l_2.\textit{get}\ s'_1) & \text{by GetEquiv for } l_2 \text{ and } l_1 \\
 & & \text{and definition } \sim_V \\
= & (l_1 \sim l_2).\textit{get}\ (s'_1 \cdot s'_2) & \text{by definition } (l_1 \sim l_2).\textit{get}
\end{array}$$

and obtain the required equivalence.

▶ **PutEquiv:** Let $v = v_2 \cdot v_1$ and $v' = v'_2 \cdot v'_1$ be strings in $V_2 \cdot V_1$ and let $s = s_1 \cdot s_2$ and $s' = s'_1 \cdot s'_2$ be strings in $S_1 \cdot S_2$. By the definitions of $\sim_V$ and $\sim_S$ we have $v_2 \sim_{V_2} v'_2$ and $v_1 \sim_{V_1} v'_1$ and $s_1 \sim_{S_1} s'_1$ and $s_2 \sim_{S_2} s'_2$. Using these facts, we calculate as follows

$$\begin{array}{rll}
 & (l_1 \sim l_2).\textit{put}\ (v_2 \cdot v_1)\ (s_1 \cdot s_2) & \\
= & (l_1.\textit{put}\ v_1\ s_1) \cdot (l_2.\textit{put}\ v_2\ s_2) & \text{by definition } (l_1 \sim l_2).\textit{put} \\
\sim_S & (l_1.\textit{put}\ v'_1\ s'_1) \cdot (l_2.\textit{put}\ v'_2\ s'_2) & \text{by PutEquiv for } l_1 \text{ and } l_2 \\
 & & \text{and definition } \sim_S \\
= & (l_1 \sim l_2).\textit{put}\ (v'_2 \cdot v'_1)\ (s'_1 \cdot s'_2) & \text{by definition } (l_1 \sim l_2).\textit{put}
\end{array}$$

and obtain the required equivalence.

▶ **CreateEquiv:** Similar to the proof of PutEquiv.

▶ **GetPut:** Let $s = s_1 \cdot s_2$ be a string in $S_1 \cdot S_2$. We calculate as follows

$$\begin{array}{rll}
 & (l_1 \sim l_2).\textit{put}\ ((l_1 \sim l_2).\textit{get}\ (s_1 \cdot s_2))\ (s_1 \cdot s_2) & \\
= & (l_1 \sim l_2).\textit{put}\ ((l_2.\textit{get}\ s_2) \cdot (l_1.\textit{get}\ s_1))\ (s_1 \cdot s_2) & \text{by definition } (l_1 \sim l_2).\textit{get} \\
= & (l_1.\textit{put}\ (l_1.\textit{get}\ s_1)\ s_1) \cdot (l_2.\textit{put}\ (l_2.\textit{get}\ s_2)\ s_2) & \text{by definition } (l_1 \sim l_2).\textit{put} \\
 & \quad \text{with } V_2 \cdot^! V_1 \text{ and } \mathrm{cod}(l_2.\textit{get}) = V_2 \text{ and } \mathrm{cod}(l_1.\textit{get}) = V_1 & \\
\sim_S & s_1 \cdot s_2 & \text{By GetPut for } l_1 \text{ and } l_2 \\
 & & \text{and definition of } \sim_S
\end{array}$$

and obtain the required equivalence.

▶ **PutGet:** Let $v = v_2 \cdot v_1$ be a string in $V_2 \cdot V_1$ and let $s = s_1 \cdot s_2$ be a string in $S_1 \cdot S_2$. We calculate as follows

$$\begin{array}{rll}
 & (l_1 \sim l_2).\textit{get}\ ((l_1 \sim l_2).\textit{put}\ (v_2 \cdot v_1)\ (s_1 \cdot s_2)) & \\
= & (l_1 \sim l_2).\textit{get}\ ((l_1.\textit{put}\ v_1\ s_1) \cdot (l_2.\textit{put}\ v_2\ s_2)) & \text{by definition } (l_1 \sim l_2).\textit{put} \\
= & (l_2.\textit{get}\ (l_2.\textit{put}\ v_2\ s_2)) \cdot (l_1.\textit{get}\ (l_1.\textit{put}\ v_1\ s_1)) & \text{by definition } (l_1 \sim l_2).\textit{get} \\
 & \quad \text{with } S_1 \cdot^! S_2 \text{ and } \mathrm{cod}(l_1.\textit{put}) = S_1 \text{ and } \mathrm{cod}(l_2.\textit{put}) = S_2 & \\
\sim_V & v_2 \cdot v_1 & \text{By PutGet for } l_1 \text{ and } l_2 \\
 & & \text{and definition } \sim_V
\end{array}$$

and obtain the required equivalence.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{S_1 \cdot^! S_2 \qquad p \in \Pi u : (U_1 \cdot U_2).\ \{(u_1, u_2) \in U_1 \times U_2 \mid u_1 \cdot u_2 = u\} \\ q_1 \in S_1 \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} U_1/{\sim_{U_1}} \\ q_2 \in S_2 \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} U_2/{\sim_{U_2}}}{q_1 \cdot_p q_2 \in (S_1 \cdot S_2) \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} (U_1 \cdot U_2)/\mathsf{TrClose}(\sim_{U_1} \cdot \sim_{U_2})}}$$

3.2.14 Lemma: Let $q_1 \in S_1 \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} U_1/{\sim_{U_1}}$ and $q_2 \in S_2 \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} U_2/{\sim_{U_2}}$ be canonizers such that $S_1 \cdot^! S_2$ and let p be a function in:

$$\Pi u : (U_1 \cdot U_2).\ \{(u_1, u_2) \in U_1 \times U_2 \mid u_1 \cdot u_2 = u\}$$

Then $q_1 \cdot_p q_2$ is a canonizer in $(S_1 \cdot S_2) \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} (U_1 \cdot U_2)/\mathsf{TrClose}(\sim_{U_1} \cdot \sim_{U_2})$.

Proof: We prove the canonizer law directly. To shorten the proof, we will abbreviate $\sim_{U_1} \cdot \sim_{U_2}$ as $\sim_U$.

▶ **ReCanonize:** Let u be a string in $U_1 \cdot U_2$ and let u_1 be a string in U_1 and u_2 a string in U_2 with $p\ u = (u_1, u_2)$. We calculate as follows

$$\begin{aligned}
& (q_1 \cdot_p q_2).\mathit{canonize}\ ((q_1 \cdot_p q_2).\mathit{choose}\ u) \\
= {} & (q_1 \cdot_p q_2).\mathit{canonize} \\
& \quad ((q_1.\mathit{choose}\ u_1) \cdot (q_2.\mathit{choose}\ u_2)) && \text{by definition } (q_1 \cdot_p q_2).\mathit{choose} \\
= {} & (q_1.\mathit{canonize}\ (q_1.\mathit{choose}\ u_1)) \cdot && \text{by definition } (q_1 \cdot_p q_2).\mathit{canonize} \\
& (q_2.\mathit{canonize}\ (q_2.\mathit{choose}\ u_2)) \\
& \quad \text{with } S_1 \cdot^! S_2 \text{ and } \mathrm{cod}(q_1.\mathit{choose}) = S_1 \text{ and } \mathrm{cod}(q_2.\mathit{choose}) = S_2 \\
\sim_U {} & u_1 \cdot u_2 && \text{by } \textsc{ReCanonize} \text{ for } q_1 \text{ and } q_2 \\
= {} & u && \text{by type of } p
\end{aligned}$$

and obtain the required equivalence. □

$$\boxed{\dfrac{S^{!*} \qquad p \in \Pi u : U^*.\ \{[u_1, \ldots, u_n] \in U\ \mathit{list} \mid u_1 \cdots u_n = u\} \\ q \in S \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} U/{\sim_U}}{q^{*p} \in S^* \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} U^*/\mathsf{TrClose}({\sim_U}^*)}}$$

3.2.16 Lemma: Let $q \in S \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} U/{\sim_U}$ be a canonizer such that $S^{!*}$. Also let p be a function in:

$$\Pi u : U^*.\ \{[u_1, \ldots, u_n] \in U\ \mathit{list} \mid u_1 \cdots u_n = u\}$$

Then q^{*p} is a canonizer in $S^* \mathrel{\leftrightarrow\!\!\!\!\!\!\mapsto} U^*/\mathsf{TrClose}({\sim_U}^*)$.

Proof: We prove the canonizer law directly.

▶ **ReCanonize:** Let u be a string in U^* and let u_1 to u_n be strings in U with $p\ u = [u_1, \ldots, u_n]$. We calculate as follows

$$\begin{aligned}
& q^{*p}.\mathit{canonize}\ (q^{*p}.\mathit{choose}\ u) \\
= {} & q^{*p}.\mathit{canonize} \\
& \quad ((q.\mathit{choose}\ u_1) \cdots (q.\mathit{choose}\ u_n)) && \text{by definition } q^{*p}.\mathit{choose} \\
= {} & (q.\mathit{canonize}\ (q.\mathit{choose}\ u_1)) \cdots && \text{by definition } q^{*p}.\mathit{canonize} \\
& (q.\mathit{canonize}\ (q.\mathit{choose}\ u_n)) \\
& \quad \text{with } S^{!*} \text{ and } \mathrm{cod}(q.\mathit{choose}) = S \\
{\sim_U}^* {} & u_1 \cdots u_n && \text{by } \textsc{ReCanonize} \text{ for } q \\
= {} & u && \text{by type of } p
\end{aligned}$$

and obtain the required equivalence. □

$$\boxed{\dfrac{\begin{array}{c} S_1 \cap S_2 = \emptyset \\ q_1 \in S_1 \leftrightarrow U_1/{\sim_{U_1}} \\ q_2 \in S_2 \leftrightarrow U_2/{\sim_{U_2}} \end{array}}{q_1 \,|\, q_2 \in (S_1 \cup S_2) \leftrightarrow (U_1 \cup U_2)/\mathsf{TrClose}({\sim_{U_1}} \cup {\sim_{U_2}})}}$$

3.2.17 Lemma: Let $q_1 \in S_1 \leftrightarrow U_1/{\sim_{U_1}}$ and $q_2 \in S_2 \leftrightarrow U_2/{\sim_{U_2}}$ be canonizers such that the intersection $S_1 \cap S_2$ of the source types is empty. Then $q_1 \,|\, q_2$ is a canonizer in $S_1 \cup S_2 \leftrightarrow (U_1 \cup U_2)/\mathsf{TrClose}({\sim_{U_1}} \cup {\sim_{U_2}})$.

Proof: We prove the canonizer law directly.

▶ **ReCanonize:** Let $u \in U_1 \cup U_2$. We analyze two cases.

Case $u \in U_1$**:** We calculate as follows

$$\begin{array}{lll} & (q_1 \,|\, q_2).\mathit{canonize} & \\ & \quad ((q_1 \,|\, q_2).\mathit{choose}\ u) & \\ = & (q_1 \,|\, q_2).\mathit{canonize}\ (q_1.\mathit{choose}\ u) & \text{by definition } (q_1 \,|\, q_2).\mathit{choose} \\ = & q_1.\mathit{canonize}\ (q_1.\mathit{choose}\ u) & \text{by definition } (q_1 \,|\, q_2).\mathit{canonize} \\ & \quad \text{with } \mathrm{cod}(q_1.\mathit{choose}) = S_1 & \\ \sim_U & u & \text{by } \textsc{ReCanonize} \text{ for } q_1 \end{array}$$

and obtain the required equivalence.

Case $u \in U_1$**:** Symmetric to the previous case. □

$$\boxed{\dfrac{\begin{array}{c} V_1 \cdot^{!} V_2 \\ l \in S/{\sim_S} \Longleftrightarrow V_1/{\sim_{V_1}} \\ f \in S \to V_2 \end{array}}{\mathit{dup}_1\ l\ f \in S/{\sim_S} \Longleftrightarrow (V_1 \cdot V_2)/({\sim_{V_1}} \cdot \mathsf{Tot}(V_2))}}$$

3.2.18 Lemma: Let l be a quotient lens in $S/{\sim_S} \Longleftrightarrow V_1/{\sim_{V_1}}$ and let f be a function from S to V_2 such that $V_1 \cdot^{!} V_2$. Then $\mathit{dup}_1\ l\ f$ is a quotient lens in $S/{\sim_S} \Longleftrightarrow (V_1 \cdot V_2)/({\sim_{V_1}} \cdot \mathsf{Tot}(V_2))$.

Proof: We prove each quotient lens law separately. To shorten the proof, abbreviate ${\sim_{V_1}} \cdot \mathsf{Tot}(V_2)$ as $\sim_V$.

▶ **GetEquiv:** Let $s \in S$ and $s' \in S$ be strings such that $s \sim_S s'$. We calculate as follows

$$\begin{array}{lll} & (\mathit{dup}_1\ l\ f).\mathit{get}\ s & \\ = & (l.\mathit{get}\ s) \cdot (f\ s) & \text{by definition } (\mathit{dup}_1\ l\ f).\mathit{get} \\ \sim_V & (l.\mathit{get}\ s') \cdot (f\ s') & \text{by } \textsc{GetEquiv} \text{ for } l \text{ and definition of } \sim_V \\ = & (\mathit{dup}_1\ l\ f).\mathit{get}\ s' & \end{array}$$

and obtain the required equivalence.

▶ **PutEquiv:** Let $v = v_1 \cdot v_2$ and $v' = v_1' \cdot v_2'$ be strings in $V_1 \cdot V_2$ such that $v \sim_V v'$ and let s and s' be strings in S such that $s \sim_S s'$. By the definition of $\sim_V$ we have $v_1 \sim_{V_1} v_1'$. Using these facts and definitions, we calculate as follows

$$\begin{array}{ll}
\quad (\mathit{dup}_1\ l\ f).\mathit{put}\ (v_1{\cdot}v_2)\ s & \\
= l.\mathit{put}\ v_1\ s & \text{by definition } (\mathit{dup}_1\ l\ f).\mathit{put} \\
\sim_S l.\mathit{put}\ v_1'\ s' & \text{by PutEquiv for } l \\
= (\mathit{dup}_1\ l\ f).\mathit{put}\ (v_1'{\cdot}v_2')\ s' & \text{by definition } (\mathit{dup}_1\ l\ f).\mathit{put}
\end{array}$$

and obtain the required equivalence.

▶ **CreateEquiv:** Similar to the proof of PutEquiv.

▶ **GetPut:** Let $s \in S$ be a string. We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{dup}_1\ l\ f).\mathit{put}\ ((\mathit{dup}_1\ l\ f).\mathit{get}\ s)\ s & \\
= (\mathit{dup}_1\ l\ f).\mathit{put}\ ((l.\mathit{get}\ s){\cdot}(f\ s))\ s & \text{by definition } (\mathit{dup}_1\ l\ f).\mathit{get} \\
= l.\mathit{put}\ (l.\mathit{get}\ s) & \text{by definition } (\mathit{dup}_1\ l\ f).\mathit{put} \\
\quad\quad \text{with } V_1{\cdot}^{!}V_2 \text{ and } \mathrm{cod}(l.\mathit{get}) = V_1 & \text{and } \mathrm{cod}(f) = V_2 \\
\sim_S s & \text{by GetPut for } l
\end{array}$$

and obtain the required equivalence.

▶ **PutGet:** Let $v = v_1{\cdot}v_2$ be a string in V and let s be a string in S. We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{dup}_1\ l\ f).\mathit{get}\ ((\mathit{dup}_1\ l\ f).\mathit{put}\ (v_1{\cdot}v_2)\ s) & \\
= (\mathit{dup}_1\ l\ f).\mathit{get}\ (l.\mathit{put}\ v_1\ s) & \text{by definition } (\mathit{dup}_1\ l\ f).\mathit{put} \\
= (l.\mathit{get}\ (l.\mathit{put}\ v_1\ s)){\cdot}(f\ (l.\mathit{put}\ v_1\ s)) & \text{by definition } (\mathit{dup}_1\ l\ f).\mathit{get} \\
\sim_V v_1{\cdot}v_2 & \text{by PutGet for } l \\
 & \text{and definition of } \sim_V
\end{array}$$

and obtain the required equivalence.

▶ **CreateGet:** Similar to the proof of PutGet. □

$$\boxed{\dfrac{S_0 \subseteq S \qquad \forall s \in S_0.\ f\ s = s \qquad f \in S \to S_0}{\mathit{normalize}\ f \in S \mathrel{\leftarrow\!\!\!\!\!\mapsto} S_0/{=}}}$$

3.2.19 Lemma: Let S and S_0 be sets such that $S_0 \subseteq S$. Also let $f \in S \to S_0$ be a function from S to S_0 such that $f\ s = s$ for every s in S_0. Then *normalize* f is a canonizer in $S \mathrel{\leftarrow\!\!\!\!\!\mapsto} S_0/{=}$.

Proof: We prove the canonizer law directly.

▶ **ReCanonize:** Let $s \in S_0$. We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{normalize}\ f).\mathit{canonize} & \\
\quad\quad ((\mathit{normalize}\ f).\mathit{choose}\ s) & \\
= (\mathit{normalize}\ f).\mathit{canonize}\ s & \text{by definition } (\mathit{normalize}\ f).\mathit{choose} \\
= f\ s & \text{by definition } (\mathit{normalize}\ f).\mathit{canonize} \\
= s & \text{as } s \in S_0
\end{array}$$

and obtain the required equality. □

$$\frac{n \in \mathbb{N} \qquad sp \in \Sigma^* \qquad nl \in \Sigma^* \qquad (\Sigma^* \cdot nl \cdot \Sigma^*) \cap S_0 = \emptyset}{unwrap\ n\ S_0\ sp\ nl \in ([(sp \cup nl)/sp]S_0) \leftrightarrow\!\!\!\!\!\!\rhd\ S_0/{=}}$$

3.2.20 Lemma: Let n be a natural number, S_0 a language, and *sp* and *nl* strings such that for every string u in S_0 the string *nl* does not occur in u. Then $unwrap\ n\ S_0\ sp\ nl$ is a canonizer in $([(sp \cup nl)/sp]S_0) \leftrightarrow\!\!\!\!\!\!\rhd\ S_0/{=}$.

Proof: We prove the canonizer law directly.

▶ **ReCanonize**: Let s be a string in S_0. The required equality,

$$(unwrap\ n\ S_0\ sp\ nl).canonize\ (unwrap\ n\ S_0\ sp\ nl).choose\ s = s$$

is immediate as *nl* does not occur in *s*, $(unwrap\ n\ S_0\ sp\ nl).choose$ replaces some occurrences of *s* with *nl*, and $(unwrap\ n\ S_0\ sp\ nl).canonize$ replaces every occurrence of *nl* with *s*. □

Resourceful Lens Proofs

4.1.3 Lemma [PutChunks]: Let l be a resourceful lens in $S \overset{C,k}{\Longleftrightarrow} V$, let $v \in V$ be a view, let $c \in C$ be a rigid complement, and let $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ be a resource. Then $locs(l.put\ v\ (c,r)) = locs(v)$.

Proof: We calculate as follows

$$\begin{array}{ll}
\ locs(l.put\ v\ (c,r)) & \\
= locs(l.get\ (l.put\ v\ (c,r))) & \text{by GetChunks for } l \\
= locs(v) & \text{by PutGet for } l
\end{array}$$

and obtain the required equality. □

4.1.4 Lemma [CreateChunks]: Let l be a resourceful lens in $S \overset{C,k}{\Longleftrightarrow} V$, let $v \in V$ be a view, and let $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ be a resource. Then $locs(l.create\ v\ r) = locs(v)$.

Proof: Similar to the proof of Lemma 4.1.3. □

4.1.5 Lemma [ReorderPut]: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens, let $v \in V$ be a view, let $c \in C$ be a rigid complement, let $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ be a resource, and let $q \in \mathsf{Perms}(v)$ be a permutation on the chunks of v. Then we have $\circlearrowleft\!\!\!\!\!\!{\scriptstyle q}\;\ (l.put\ v\ (c,r)) = l.put\ (\circlearrowleft\!\!\!\!\!\!{\scriptstyle q}\;\ v)\ (c, r \circ q^{-1})$.

Proof: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens, let $v \in V$ be a view, let $c \in C$ be a rigid complement, let $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ be a resource, and let $q \in \mathsf{Perms}(v)$ be a permutation on the chunks in v such that $\mathrm{dom}(r) = (locs(v))$. To shorten the proof, define structures s_1 and s_2 as follows:

$$s_1 \triangleq \circlearrowleft\!\!\!\!\!\!{\scriptstyle q}\;\ (l.put\ v\ (c,r)) \qquad s_2 \triangleq l.put\ (\circlearrowleft\!\!\!\!\!\!{\scriptstyle q}\;\ v)\ (c, r \circ q^{-1})$$

First, we demonstrate that the sets of locations in s_1 and s_2 are identical, by calculating as follows:

$$\begin{array}{ll}
\ locs(s_1) & \\
= locs(\circlearrowleft\!\!\!\!\!\!{\scriptstyle q}\;\ (l.put\ v\ (c,r))) & \text{by definition } s_1 \\
= locs(l.put\ v\ (c,r)) & \text{by definition } locs \text{ and } \circlearrowleft \\
= locs(v) & \text{by Lemma 4.1.3 for } l \\
= locs(\circlearrowleft\!\!\!\!\!\!{\scriptstyle q}\;\ v) & \text{by definition } locs \text{ and } \circlearrowleft \\
= locs(l.put\ (\circlearrowleft\!\!\!\!\!\!{\scriptstyle q}\;\ v)\ (c, r \circ q^{-1})) & \text{by Lemma 4.1.3 for } l \\
= locs(s_2) & \text{by definition of } s_2
\end{array}$$

Next, we show that for every location $x \in \mathit{locs}(s_1)$ the chunk at x in s_1 is identical to the chunk at x in s_2. Let $x \in \mathit{locs}(s_1)$ be a location. We analyze two cases.

Case $q^{-1}(x) \in \mathrm{dom}(r)$: We calculate as follows

$$\begin{aligned}
&s_1[x] \\
&= (\circlearrowleft_{q}\ (l.\mathit{put}\ v\ (c,r)))[x] && \text{by definition } s_1 \\
&= (l.\mathit{put}\ v\ (c,r))[q^{-1}(x)] && \text{by definition } \circlearrowleft \text{ and } [\cdot] \\
&= k.\mathit{put}\ (v[q^{-1}(x)])\ (r(q^{-1}(x))) && \text{by } \textsc{ChunkPut} \text{ for } l \\
&= k.\mathit{put}\ ((\circlearrowleft_{q}\ v)[x])\ ((r \circ q^{-1})(x)) && \text{by definition } \circlearrowleft \text{ and } [\cdot] \\
&= l.\mathit{put}\ (\circlearrowleft_{q}\ v)\ (c, r \circ q^{-1})[x] && \text{by } \textsc{ChunkPut} \text{ for } l \\
&= s_2[x] && \text{by definition } s_2
\end{aligned}$$

and obtain the required equality.

Case $q^{-1}(x) \notin \mathrm{dom}(r)$: Similar to the previous case, using NoChunkPut instead of ChunkPut.

Finally, we prove that $\mathit{skel}(s_1) = \mathit{skel}(s_2)$. Observe that $\mathit{skel}(v) = \mathit{skel}(\circlearrowleft_{q}\ v)$. Using this fact, we calculate as follows:

$$\begin{aligned}
&\mathit{skel}(s_1) \\
&= \mathit{skel}(\circlearrowleft_{q}\ (l.\mathit{put}\ v\ (c,r))) && \text{by definition } s_1 \\
&= \mathit{skel}(l.\mathit{put}\ v\ (c,r)) && \text{by definition } \mathit{skel} \text{ and } \circlearrowleft \\
&= \mathit{skel}(l.\mathit{put}\ (\circlearrowleft_{q}\ v)\ (c, r \circ q^{-1})) && \text{by } \textsc{SkelPut} \text{ for } l \\
&= \mathit{skel}(s_2) && \text{by definition } s_2
\end{aligned}$$

Putting all these facts together we have $s_1 = s_2$, which completes the proof. □

4.1.6 Lemma [ReorderCreate]: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens, let $v \in V$ be a view, let $r \in \{|\mathbb{N} \mapsto k.C|\}$ be a resource, and let $q \in \mathsf{Perms}(v)$ be a permutation on the chunks of v. Then we have $\circlearrowleft_{q}\ (l.\mathit{create}\ v\ r) = l.\mathit{create}\ (\circlearrowleft_{q}\ v)\ (r \circ q^{-1})$.

Proof: Similar to the proof of Lemma 4.1.5. □

$$\boxed{\dfrac{l \in S \overset{C,k}{\Longleftrightarrow} V}{\lfloor l \rfloor \in S \overset{S}{\Longleftrightarrow} V}}$$

4.1.7 Lemma: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens. Then $\lfloor l \rfloor$ is a basic lens in $S \overset{S}{\Longleftrightarrow} V$.

Proof: We prove each basic lens law separately.
► **GetPut:** Let $s \in S$. We calculate as follows

$$\begin{aligned}
&\lfloor l \rfloor.\mathit{put}\ (\lfloor l \rfloor.\mathit{get}\ s)\ (\lfloor l \rfloor.\mathit{res}\ s) \\
&= \lfloor l \rfloor.\mathit{put}\ (l.\mathit{get}\ s)\ s && \text{by definition } \lfloor l \rfloor.\mathit{get} \text{ and } \lfloor l \rfloor.\mathit{res} \\
&= l.\mathit{put}\ (l.\mathit{get}\ s)\ (c, r \circ g) && \text{by definition } \lfloor l \rfloor.\mathit{put} \\
&\qquad \text{where } c, r = l.\mathit{res}\ s \\
&\qquad\quad \text{and } g = \mathit{align}(l.\mathit{get}\ s, l.\mathit{get}\ s) \\
&= l.\mathit{put}\ (l.\mathit{get}\ s)\ (c, r) && \text{by } \textsc{GetChunks} \text{ and } \textsc{ResChunks} \\
& && \text{and as } \mathit{align}(l.\mathit{get}\ s, l.\mathit{get}\ s) = \mathit{id} \\
&= l.\mathit{put}\ (l.\mathit{get}\ s)\ (l.\mathit{res}\ s) && \text{by definition } (c, r) \\
&= s && \text{by } \textsc{GetPut} \text{ for } l
\end{aligned}$$

and obtain the required equality.

▶ **PutGet:** Let $v \in V$ and $s \in S$. We calculate as follows

$$\begin{aligned}
& \lfloor l \rfloor.\mathit{get}\ (\lfloor l \rfloor.\mathit{put}\ v\ s) & \\
={} & \lfloor l \rfloor.\mathit{get}\ (l.\mathit{put}\ v\ (c, r \circ g)) & \text{by definition } \lfloor l \rfloor.\mathit{put} \\
& \qquad \text{where } c, r = l.\mathit{res}\ s & \\
& \qquad\quad \text{and } g = \mathit{align}(v, l.\mathit{get}\ s) & \\
={} & l.\mathit{get}\ (l.\mathit{put}\ v\ (c, r \circ g)) & \text{by definition } \lfloor l \rfloor.\mathit{get} \\
={} & v & \text{by } \textsc{PutGet} \text{ for } l
\end{aligned}$$

and obtain the required equality.

▶ **CreateGet:** Let $v \in V$. We calculate as follows

$$\begin{aligned}
& \lfloor l \rfloor.\mathit{get}\ (\lfloor l \rfloor.\mathit{create}\ v) & \\
={} & \lfloor l \rfloor.\mathit{get}\ (l.\mathit{create}\ v\ \{\!|\,|\!\}) & \text{by definition } \lfloor l \rfloor.\mathit{create} \\
={} & l.\mathit{get}\ (l.\mathit{create}\ v\ \{\!|\,|\!\}) & \text{by definition } \lfloor l \rfloor.\mathit{put} \\
={} & v & \text{by } \textsc{CreateGet} \text{ for } l
\end{aligned}$$

and obtain the required equality. □

$$\boxed{\dfrac{\begin{array}{c} k' \in S' \overset{C'}{\Longleftrightarrow} V' \\ k \in S \overset{C}{\Longleftrightarrow} V \end{array}}{\widehat{k} \in S \overset{C,k'}{\Longleftrightarrow} V}}$$

4.2.1 Lemma: Let $k \in S \overset{C}{\Longleftrightarrow} V$ and $k' \in S' \overset{C'}{\Longleftrightarrow} V'$ be basic lenses. Then $\widehat{k}$ is a resourceful lens in $S \overset{C,k'}{\Longleftrightarrow} V$.

Proof: We prove each resourceful lens law separately.

▶ **GetPut:** Let $s \in S$ be a string. We calculate as follows

$$\begin{aligned}
& \widehat{k}.\mathit{put}\ (\widehat{k}.\mathit{get}\ s)\ (\widehat{k}.\mathit{res}\ s) & \\
={} & \widehat{k}.\mathit{put}\ (k.\mathit{get}\ s)\ (k.\mathit{res}\ s, \{\!|\,|\!\}) & \text{by definition of } \widehat{k}.\mathit{get} \text{ and } \widehat{k}.\mathit{res} \\
={} & k.\mathit{put}\ (k.\mathit{get}\ s)\ (k.\mathit{res}\ s) & \text{by definition of } \widehat{k}.\mathit{put} \\
={} & s & \text{by } \textsc{GetPut} \text{ for } k
\end{aligned}$$

and obtain the required equality.

▶ **PutGet:** Let $v \in V$ be a string, $c \in C$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k'.C|\!\}$ a resource. We calculate as follows

$$\begin{aligned}
& \widehat{k}.\mathit{get}\ (\widehat{k}.\mathit{put}\ v\ (c, r)) & \\
={} & \widehat{k}.\mathit{get}\ (k.\mathit{put}\ v\ c) & \text{by definition } \widehat{k}.\mathit{put} \\
={} & k.\mathit{get}\ (k.\mathit{put}\ v\ c) & \text{by definition } \widehat{k}.\mathit{get} \\
={} & v & \text{by } \textsc{PutGet} \text{ for } k
\end{aligned}$$

and obtain the required equality.

▶ **CreateGet:** Let $v \in V$ be a string and $r \in \{\!|\mathbb{N} \mapsto k'.C|\!\}$ a resource. We calculate as follows

$$\begin{aligned}
& \widehat{k}.\mathit{get}\ (\widehat{k}.\mathit{create}\ v\ r) & \\
={} & \widehat{k}.\mathit{get}\ (k.\mathit{create}\ v) & \text{by definition } \widehat{k}.\mathit{create} \\
={} & k.\mathit{get}\ (k.\mathit{create}\ v) & \text{by definition } \widehat{k}.\mathit{get} \\
={} & v & \text{by } \textsc{CreateGet} \text{ for } k
\end{aligned}$$

and obtain the required equality.

▶ **GetChunks**: Let $s \in S$. We calculate as follows

$$\begin{aligned} \mathit{locs}(s) &= \emptyset && \text{as } S \text{ is a language of ordinary strings} \\ &= \mathit{locs}(\widehat{k}.\mathit{get}\ s) && \text{as } V \text{ is a language of ordinary strings} \end{aligned}$$

and obtain the required equality.

▶ **ResChunks**: Let $s \in S$ be a string, $c \in C$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k'.C|\!\}$ a resource such that $(c, r) = \widehat{k}.\mathit{res}\ s$. By the definition of $\widehat{k}.\mathit{res}$ we have that $r = \{\!||\!\}$. Using this fact, we calculate as follows

$$\begin{aligned} \mathit{locs}(s) &= \emptyset && \text{as } S \text{ is a language of ordinary strings} \\ &= \mathrm{dom}(r) && \text{as } r = \{\!||\!\} \end{aligned}$$

and obtain the required equality.

▶ **ChunkPut**: Vacuously holds. Suppose, for a contradiction, that there exists a string $v \in \lfloor V \rfloor$, a resource $r \in \{\!|\mathbb{N} \mapsto k'.C|\!\}$, and a location $x \in (\mathit{locs}(v) \cap \mathrm{dom}(r))$. As V is a language of ordinary strings, we have $\mathit{locs}(v) = \emptyset$, which contradicts $x \in \mathit{locs}(v)$.

▶ **ChunkCreate**: Vacuously holds by the same argument.

▶ **NoChunkPut**: Vacuously holds by the same argument.

▶ **NoChunkCreate**: Vacuously holds by the same argument.

▶ **SkelPut**: Let v and v' be strings in V, let c be a rigid complement in C, and let r and r' be resources in $\{\!|\mathbb{N} \mapsto k'.C|\!\}$ such that $\mathit{skel}(v) = \mathit{skel}(v')$. As V is a language of ordinary strings, we have that $v = v'$. Using this fact, we calculate as follows:

$$\begin{aligned} &\mathit{skel}(\widehat{k}.\mathit{put}\ v\ (c, r)) \\ =\ &\mathit{skel}(k.\mathit{put}\ v\ c) && \text{by definition } \widehat{k}.\mathit{put} \\ =\ &\mathit{skel}(k.\mathit{put}\ v'\ c) && \text{as } v = v' \\ =\ &\mathit{skel}(\widehat{k}.\mathit{put}\ v'\ (c, r')) && \text{by definition } \widehat{k}.\mathit{put} \end{aligned}$$

and obtain the required equality.

▶ **SkelCreate**: Similar to the proof of SkelPut. □

$$\boxed{\dfrac{k \in S \overset{C}{\Longleftrightarrow} V}{\langle k \rangle \in \langle S \rangle \overset{\{\square\},k}{\Longleftrightarrow} \langle V \rangle}}$$

4.2.2 Lemma: Let $k \in S \overset{C}{\Longleftrightarrow} V$ be a basic lens. Then $\langle k \rangle$ is a resourceful lens in $\langle S \rangle \overset{\{\square\},k}{\Longleftrightarrow} \langle V \rangle$.

Proof: We prove each resourceful lens law separately.

▶ **GetPut**: Let $s \in \langle S \rangle$ be a string. We calculate as follows

$$\begin{aligned} &\langle k \rangle.\mathit{put}\ (\langle k \rangle.\mathit{get}\ s)\ (\langle k \rangle.\mathit{res}\ s) \\ =\ &\langle k \rangle.\mathit{put}\ (k.\mathit{get}\ s)\ (\square, \{\!|1 \mapsto k.\mathit{res}\ s|\!\}) && \text{by definition } \langle k \rangle.\mathit{get} \text{ and } \langle k \rangle.\mathit{res} \\ =\ &k.\mathit{put}\ (k.\mathit{get}\ s)\ (\{\!|1 \mapsto k.\mathit{res}\ s|\!\}(1)) && \text{by definition } \langle k \rangle.\mathit{put} \\ &\quad \text{with } 1 \in \mathrm{dom}(\{\!|1 \mapsto k.\mathit{res}\ s|\!\}) \\ =\ &k.\mathit{put}\ (k.\mathit{get}\ s)\ (k.\mathit{res}\ s) && \text{by definition application} \\ =\ &s && \text{by GetPut for } k \end{aligned}$$

and obtain the required equality.

▶ **PutGet**: Let $v \in \langle V \rangle$ be a string, $\square \in \{\square\}$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource. We calculate as follows

$$\begin{array}{lll} & \langle k\rangle.\mathit{get}\ (\langle k\rangle.\mathit{put}\ v\ (\square, r)) & \\ = & \begin{cases} k.\mathit{get}\ (k.\mathit{put}\ v\ r(1)) & \text{if } 1 \in \mathrm{dom}(r) \\ k.\mathit{get}\ (k.\mathit{create}\ v) & \text{otherwise} \end{cases} & \begin{array}{l} \text{by the definition of } \langle k\rangle.\mathit{get} \\ \text{and } \langle k\rangle.\mathit{put} \end{array} \\ = & v & \text{by PutGet and CreateGet for } l \end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof of PutGet.

▶ **GetChunks**: Let $s \in \langle S \rangle$ be a string. We calculate as follows

$$\begin{array}{rll} \mathit{locs}(s) & = \{1\} & \text{by definition } \mathit{locs} \\ & = \mathit{locs}(\langle k\rangle.\mathit{get}\ s) & \text{by definition } \mathit{locs} \end{array}$$

and obtain the required equality.

▶ **ResChunks**: Let $s \in \lfloor\langle S \rangle\rfloor$ be a string, $\square \in \{\square\}$ a rigid complement, and $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource such that $(\square, r) = \langle k\rangle.\mathit{res}\ s$. We calculate as follows

$$\begin{array}{rll} \mathrm{dom}(r) & = \mathrm{dom}(\{\!|1 \mapsto k.\mathit{res}\ s|\!\}) & \text{by definition } \langle k\rangle.\mathit{res} \\ & = \{1\} & \text{by definition dom} \\ & = \mathit{locs}(s) & \text{by definition } \mathit{locs} \end{array}$$

and obtain the required equality.

▶ **ChunkPut**: Let $v \in \langle V \rangle$ be a string, $\square \in \{\square\}$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource, and $x \in (\mathit{locs}(v) \cap \mathrm{dom}(r))$ a location. As $\mathit{locs}(v) = \{1\}$ we must have $x = 1$ and $1 \in \mathrm{dom}(r)$. We calculate as follows

$$\begin{array}{lll} & (\langle k\rangle.\mathit{put}\ v\ (\square, r))[x] & \\ = & \langle k\rangle.\mathit{put}\ v\ (c, r) & \text{by definition } [\cdot] \text{ and } x = 1 \\ = & k.\mathit{put}\ v\ (r(1)) & \text{by definition } \langle k\rangle.\mathit{put} \\ & \quad \text{with } 1 \in \mathrm{dom}(r) & \\ = & k.\mathit{put}\ (v[x])\ (r(x)) & \text{by definition } [\cdot] \text{ and } x = 1 \end{array}$$

and obtain the required equality.

▶ **ChunkCreate**: Similar to the proof of ChunkPut.

▶ **NoChunkPut**: Let $v \in \langle V \rangle$ be a string, $\square \in \{\square\}$ a rigid complement, $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ a resource, and $x \in (\mathit{locs}(v) - \mathrm{dom}(r))$ a location. As $\mathit{locs}(v) = \{1\}$ we must have $x = 1$ and $1 \notin \mathrm{dom}(r)$. Using these facts and definitions, we calculate as follows

$$\begin{array}{lll} & (\langle k\rangle.\mathit{put}\ v\ (\square, r))[x] & \\ = & \langle k\rangle.\mathit{put}\ v\ (c, r) & \text{by definition } [\cdot] \text{ and } x = 1 \\ = & k.\mathit{create}\ v & \text{by definition } \langle k\rangle.\mathit{put} \\ & \quad \text{with } 1 \notin \mathrm{dom}(r) & \\ = & k.\mathit{create}\ (v[x]) & \text{by definition } [\cdot] \text{ and } x = 1 \end{array}$$

and obtain the required equality.

▶ **NoChunkCreate**: Similar to the proof of NoChunkPut.

▶ **SkelPut:** Let v and v' be strings in $\langle V\rangle$, let $\square \in \{\square\}$ be a rigid complement, and let r and r' be resources in $\{\!|\mathbb{N} \mapsto k.C|\!\}$ such that $skel(v) = skel(v)$. We calculate as follows

$$\begin{array}{ll}
\quad skel(\langle k\rangle.put\ v\ (\square, r)) & \\
= \square & \text{by definition } skel \\
= skel(\langle k\rangle.put\ v'\ (\square, r')) & \text{by definition } skel
\end{array}$$

and obtain the required equality.

▶ **SkelCreate:** Similar to the proof of SkelPut. □

$$\boxed{\dfrac{\begin{array}{ll} l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 & \lfloor S_1\rfloor \cdot^! \lfloor S_2\rfloor \\ l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 & \lfloor V_1\rfloor \cdot^! \lfloor V_2\rfloor \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2) \overset{(C_1 \times C_2),k}{\Longleftrightarrow} (V_1 \cdot V_2)}}$$

4.2.3 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be resourceful lenses such that $\lfloor S_1\rfloor \cdot^! \lfloor S_2\rfloor$ and $\lfloor V_1\rfloor \cdot^! \lfloor V_2\rfloor$. Then $l_1 \cdot l_2$ is a resourceful lens in $(S_1 \cdot S_2) \overset{(C_1 \times C_2),k}{\Longleftrightarrow} (V_1 \cdot V_2)$.

Proof: We prove each resourceful lens law separately.
▶ **GetPut:** Let s be a string in $s_1 \cdot s_2 \in S_1 \cdot S_2$. We calculate as follows

$$\begin{array}{ll}
\quad (l_1 \cdot l_2).put & \\
\qquad ((l_1 \cdot l_2).get\ (s_1 \cdot s_2))\ ((l_1 \cdot l_2).res\ (s_1 \cdot s_2)) & \\
= (l_1 \cdot l_2).put & \\
\qquad ((l_1.get\ s_1) \cdot (l_2.get\ s_2))\ ((l_1 \cdot l_2).res\ (s_1 \cdot s_2)) & \text{by definition } (l_1 \cdot l_2).get \\
= (l_1 \cdot l_2).put & \\
\qquad ((l_1.get\ s_1) \cdot (l_2.get\ s_2))\ ((c_1, c_2), r_1 +\!\!+\ r_2) & \text{by definition } (l_1 \cdot l_2).res \\
\qquad \text{where } c_1, r_1 = l_1.res\ s_1 & \\
\qquad \text{and } c_2, r_2 = l_2.res\ s_2 & \\
= (l_1.put\ (l_1.get\ s_1)\ (c_1, r_1')) \cdot & \\
\quad (l_2.put\ (l_2.get\ s_2)\ (c_2, r_2')) & \text{by definition } (l_1 \cdot l_2).put \\
\qquad \text{where } r_1', r_2' = split(|l.get\ s_1|, r_1 +\!\!+\ r_2) & \\
\qquad \text{with } V_1 \cdot^! V_2 \text{ and } \mathrm{cod}(l_1.get) = V_1 \text{ and } \mathrm{cod}(l_2.get) = V_2 & \\
= (l_1.put\ (l_1.get\ s_1)\ (c_1, r_1)) \cdot & \\
\quad (l_2.put\ (l_2.get\ s_2)\ (c_2, r_2)) & \text{by GetChunks} \\
\qquad \text{with ResChunks for } l_1 \text{ and definition } split & \\
= s_1 \cdot s_2 & \text{by GetPut for } l_1 \text{ and } l_2
\end{array}$$

and obtain the required equality.

▶ **PutGet:** Let $v = v_1 \cdot v_2$ be a string in $V_1 \cdot V_2$, let (c_1, c_2) be a rigid complement in $C_1 \times C_2$, and let r be a resource in $\{|\mathbb{N} \mapsto k.C|\}$. We calculate as follows

$$\begin{array}{ll}
\quad (l_1{\cdot}l_2).\mathit{get} & \\
\qquad ((l_1{\cdot}l_2).\mathit{put}\ (v_1{\cdot}v_2)\ ((c_1,c_2),r)) & \\
= (l_1{\cdot}l_2).\mathit{get} & \\
\qquad ((l_1.\mathit{put}\ v_1\ (c_1,r_1)){\cdot}(l_2.\mathit{put}\ v_2\ (c_2,r_2))) & \text{by definition } (l_1{\cdot}l_2).\mathit{put} \\
\qquad \text{where } r_1,r_2 = \mathit{split}(|v_1|,r) & \\
= (l_1.\mathit{get}\ (l_1.\mathit{put}\ v_1\ (c_1,r_1))){\cdot} & \\
\quad (l_2.\mathit{get}\ (l_2.\mathit{put}\ v_2\ (c_2,r_2))) & \text{by definition } (l_1{\cdot}l_2).\mathit{get} \\
\qquad \text{with } \lfloor S_1\rfloor {\cdot}^{!} \lfloor S_2\rfloor \text{ and } \mathrm{cod}(l_1.\mathit{put}) = S_1 \text{ and } \mathrm{cod}(l_2.\mathit{put}) = S_2 & \\
= v_1{\cdot}v_2 & \text{by PutGet for } l_1 \text{ and } l_2
\end{array}$$

and obtain the required equality.

▶ **CreateGet:** Similar to the proof of PutGet.

▶ **GetChunks:** Let $s = s_1{\cdot}s_2$ be a string in $S_1{\cdot}S_2$. We calculate as follows

$$\begin{array}{ll}
\quad \mathit{locs}(s_1{\cdot}s_2) & \\
= \{1,\ldots,|s_1|+|s_2|\} & \text{by definition } \mathit{locs} \\
= \{1,\ldots,|l_1.\mathit{get}\ s_1|+|l_2.\mathit{get}\ s_2|\} & \text{by GetChunks for } l_1 \text{ and } l_2 \\
= \mathit{locs}((l_1.\mathit{get}\ s_1){\cdot}(l_2.\mathit{get}\ s_2)) & \text{by definition } \mathit{locs} \\
= \mathit{locs}((l_1{\cdot}l_2).\mathit{get}\ (s_1{\cdot}s_2)) & \text{by definition } (l_1{\cdot}l_2).\mathit{get}
\end{array}$$

and obtain the required equality.

▶ **ResChunks:** Let $s = s_1{\cdot}s_2$ be a string in $S_1{\cdot}S_2$, (c_1,c_2) a rigid complement in $C_1 \times C_2$, and r a resource in $\{|\mathbb{N} \mapsto k.C|\}$ with $((c_1,c_2),r) = (l_1{\cdot}l_2).\mathit{res}\ s$. We calculate as follows

$$\begin{array}{ll}
\quad \mathrm{dom}(r) & \\
= \mathrm{dom}(r_1 \mathbin{++} r_2) & \text{by definition } (l_1{\cdot}l_2).\mathit{res} \\
\qquad \text{where } r_1,c_1 = l_1.\mathit{res}\ s_1 \text{ and } r_2,c_2 = l_2.\mathit{res}\ s_2 & \\
= \mathrm{dom}(r_1) \cup \{i+|r_1| \mid i \in \mathrm{dom}(r_2)\} & \text{by definition } (\mathbin{++}) \text{ and dom} \\
= \mathit{locs}(s_1) \cup \{i+|s_1| \mid i \in \mathit{locs}(s_2)\} & \text{by ResChunks for } l_1 \text{ and } l_2 \\
= \mathit{locs}(s_1{\cdot}s_2) & \text{by definition } \mathit{locs}
\end{array}$$

and obtain the required equality.

▶ **ChunkPut:** Let $v = v_1{\cdot}v_2$ be a string in $V_1{\cdot}V_2$, let (c_1,c_2) be a rigid complement in $C_1 \times C_2$, let r be a resource in $\{|\mathbb{N} \mapsto k.C|\}$, and let x be a location in $\mathit{locs}(v) \cap \mathrm{dom}(r)$. We analyze two cases.

Case $x \in \mathit{locs}(v_1)$: We calculate as follows

$$\begin{array}{ll}
\quad ((l_1{\cdot}l_2).\mathit{put}\ (v_1{\cdot}v_2)\ ((c_1,c_2),r))[x] & \\
= ((l_1.\mathit{put}\ v_1\ (c_1,r_1)){\cdot}(l_2.\mathit{put}\ v_2\ (c_2,r_2)))[x] & \text{by definition } (l_1{\cdot}l_2).\mathit{put} \\
\qquad \text{where } r_1,r_2 = \mathit{split}(|v_1|,r) & \\
= (l_1.\mathit{put}\ v_1\ (c_1,r_1))[x] & \text{by Lemma 4.1.3} \\
 & \text{and definition } [\cdot] \\
= k.\mathit{put}\ (v_1[x])\ (r_1(x)) & \text{by ChunkPut for } l_1 \\
= k.\mathit{put}\ ((v_1{\cdot}v_2)[x])\ ((r_1 \mathbin{++} r_2)(x)) & \text{by definition } [\cdot] \text{ and } (\mathbin{++}) \\
= k.\mathit{put}\ ((v_1{\cdot}v_2)[x])\ (r(x)) & \text{by definition } \mathit{split}, \\
 & r_1, \text{ and } r_2
\end{array}$$

and obtain the required equality.

Case $x \notin (locs(v_1))$: Similar to the previous case.

▶ **ChunkCreate**: Similar to the proof of ChunkPut.

▶ **NoChunkPut**: Similar to the proof of ChunkPut.

▶ **NoChunkCreate**: Similar to the proof of ChunkPut.

▶ **SkelPut**: Let $v = v_1 \cdot v_2$ and $v' = v'_1 \cdot v'_2$ be strings in $V_1 \cdot V_2$, let (c_1, c_2) be a rigid complement in $C_1 \times C_2$, and let r and r' be resources in $\{|\mathbb{N} \mapsto k.C|\}$ such that $skel(v) = skel(v')$. By $\lfloor V_1 \rfloor \cdot^! \lfloor V_2 \rfloor$ and as $V_1 \cdot V_2$ is chunk unambiguous, we have $skel(v_1) = skel(v'_1)$ and $skel(v_2) = skel(v'_2)$. Using these facts and definitions, we calculate as follows

$$\begin{array}{ll}
\quad skel((l_1 \cdot l_2).put\ (v_1 \cdot v_2)\ ((c_1, c_2), r)) & \\
= skel(l_1.put\ v_1\ (c_1, r_1)) \cdot (l_2.put\ v_2\ (c_2, r_2)) & \text{by definition } (l_1 \cdot l_2).put \\
\qquad \text{where } r_1, r_2 = split(|v_1|, r) & \\
= skel(l_1.put\ v_1\ (c_1, r_1)) \cdot skel(l_2.put\ v_2\ (c_2, r_2)) & \text{by definition } skel \\
= skel(l_1.put\ v'_1\ (c_1, r'_1)) \cdot skel(l_2.put\ v'_2\ (c_2, r'_2)) & \text{by SkelPut for } l_1 \text{ and } l_2 \\
\qquad \text{where } r'_1, r'_2 = split(|v'_1|, r') & \\
= skel(l_1.put\ v'_1\ (c_1, r'_1)) \cdot (l_2.put\ v'_2\ (c_2, r'_2)) & \text{by definition } skel \\
= skel((l_1 \cdot l_2).put\ (v'_1 \cdot v'_2)\ ((c_1, c_2), r')) & \text{by definition } (l_1 \cdot l_2).put, \\
& r'_1 \text{ and } r'_2
\end{array}$$

and obtain the required equality.

▶ **SkelCreate**: Similar to the proof of SkelPut. □

$$\boxed{\dfrac{\lfloor S \rfloor^{!*} \qquad \lfloor V \rfloor^{!*} \\ l \in S \overset{C,k}{\Longleftrightarrow} V}{l^* \in S^* \overset{(C\ \text{list}),k}{\Longleftrightarrow} V\ ^*}}$$

4.2.4 Lemma: Let $l \in S \overset{C,k}{\Longleftrightarrow} V$ be a resourceful lens such that $\lfloor S \rfloor^{!*}$ and $\lfloor V \rfloor^{!*}$. Then l^* is a resourceful lens in $S^* \overset{K\ list,R}{\Longleftrightarrow} V^*$.

Proof: We prove each resourceful lens law separately.

▶ **GetPut:** Let $s = s_1 \cdots s_n$ be a string in S^*. To shorten the proof, let $(c_i, r_i) = l.res\ s_i$ and $(r'_i, r''_i) = split(|l.get\ s_i|, r''_i)$ for i from 1 to n where $r''_0 = \pi_2(l^*.res\ s)$. Using these facts and definitions, we calculate as follows

$$\begin{array}{ll}
\quad l^*.put\ (l^*.get(s_1 \cdots s_n))\ (l^*.res\ (s_1 \cdots s_n)) & \\
= l^*.put\ ((l.get\ s_1) \cdots (l.get\ s_n))\ (l^*.res\ (s_1 \cdots s_n)) & \text{by definition } l^*.get \\
= l^*.put & \\
\qquad ((l.get\ s_1) \cdots (l.get\ s_n)) & \\
\qquad ([c_1, \ldots, c_n], r_1 +\!\!+ \ldots +\!\!+ r_n) & \text{by definition } l^*.res \\
= (l.put\ (l.get\ s_1)\ (c_1, r'_1)) \cdots (l.put\ (l.get\ s_n)\ (c_n, r'_n)) & \text{by definition } l^*.put \\
\qquad \text{with } V^{!*} \text{ and } \text{cod}(l.get) = \lfloor V \rfloor \text{ and definition } r'_1 \text{ to } r'_n & \\
= (l.put\ (l.get\ s_1)\ (c_1, r_1)) \cdots (l.put\ (l.get\ s_n)\ (c_n, r_n)) & \text{by GetChunks for } l \\
\qquad \text{with ResChunks and definition } split & \\
= s_1 \cdots s_n & \text{by GetPut for } l
\end{array}$$

and obtain the required equality.

▶ **PutGet:** Let $v = v_1 \cdots v_n$ be a string in V^*, let $[c_1, \ldots, c_m]$ be a rigid complement in C list, and let $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ be a resource. To shorten the proof, let $r'_0 = r$ and $(r_i, r'_i) = \mathit{split}(|v_i|, r'_{(i-1)})$ for i from 1 to n. We calculate as follows

$$\begin{aligned}
& l^*.\mathit{get}\ (l^*.\mathit{put}\ (v_1 \cdots v_n)\ ([c_1, \ldots, c_m], r)) && \\
&= l^*.\mathit{get}\ (s'_1 \cdots s'_n) && \text{by definition } l^*.\mathit{put} \\
&\quad \text{where } s'_i = \begin{cases} l.\mathit{put}\, v_i\, (c_i, r_i) & i \in \{1, \ldots, \min(n, m)\} \\ l.\mathit{create}\, v_i\, r_i & i \in \{m+1, \ldots, n\} \end{cases} && \\
&= (l.\mathit{get}\ s'_1) \cdots (l.\mathit{get}\ s'_n) && \text{by definition } l^*.\mathit{get} \\
&\quad \text{with } \lfloor S \rfloor^{!*} \text{ and } \operatorname{cod}(l.\mathit{put}) = S = \operatorname{cod}(l.\mathit{create}) && \\
&= v_1 \cdots v_n && \text{by CreateGet} \\
& && \text{and PutGet for } l
\end{aligned}$$

and obtain the required equality.

▶ **CreateGet:** Similar to the proof of PutGet.

▶ **GetChunks:** Let $s = s_1 \cdots s_n$ be a string in S^*. We calculate as follows

$$\begin{aligned}
& \mathit{locs}(s_1 \cdots s_n) && \\
&= \{1, \ldots, \textstyle\sum_{i=1}^{n} |s_i|\} && \text{by definition } \mathit{locs} \\
&= \{1, \ldots, \textstyle\sum_{i=1}^{n} |l.\mathit{get}\ s_i|\} && \text{by GetChunks for } l \\
&= \mathit{locs}((l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n)) && \text{by definition } \mathit{locs} \\
&\quad \text{with } \lfloor V \rfloor^{!*} \text{ and } \operatorname{cod}(l.\mathit{get}) = V && \\
&= \mathit{locs}(l^*.\mathit{get}\ (s_1 \cdots s_n)) && \text{by definition } l^*.\mathit{get}
\end{aligned}$$

and obtain the required equality.

▶ **ResChunks:** Let $s = s_1 \cdots s_n$ be a string in S^*, let c be a rigid complement in C *list*, and let r be a resource in $\{\!|\mathbb{N} \mapsto k.C|\!\}$ such that $(c, r) = l^*.\mathit{res}\ s$. To shorten the proof, let $(c_i, r_i) = l.\mathit{res}\ s_i$ for i from 1 to n. Using these fact and definitions, we calculate as follows

$$\begin{aligned}
& \operatorname{dom}(r) && \\
&= \operatorname{dom}(r_1 \mathbin{++} \ldots \mathbin{++} r_n) && \text{by definition } l^*.\mathit{res} \\
&= \textstyle\bigcup_{i=1}^{n} \{j + \sum_{k=1}^{(i-1)} |r_k| \mid j \in \operatorname{dom}(r_i)\} && \text{by definition } (\mathbin{++}) \\
&= \textstyle\bigcup_{i=1}^{n} \{j + \sum_{k=1}^{(i-1)} |l.\mathit{get}\ s_k| \mid j \in \mathit{locs}(l.\mathit{get}\ s_i)\} && \text{by ResChunks for } l \\
&= \{1, \ldots, \textstyle\sum_{i=1}^{n} |l.\mathit{get}\ s_i|\} && \text{by definition } |\cdot| \\
&= \mathit{locs}((l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n)) && \text{by definition } \mathit{locs} \\
&= \mathit{locs}(l^*.\mathit{get}\ (s_1 \cdots s_n)) && \text{by definition } l^*.\mathit{get}
\end{aligned}$$

and obtain the required equality.

▶ **ChunkPut:** Let $v = v_1 \cdots v_n$ be a string in V^*, let $[c_1, \ldots, c_m]$ be a rigid complement in C *list*, let r be a resource in $\{\!|\mathbb{N} \mapsto k.C|\!\}$, and let x be a location in $\mathit{locs}(v) \cap \operatorname{dom}(r)$ a location. To shorten the proof, let $r'_0 = r$ and $(r_i, r'_i) = \mathit{split}(|v_i|, r'_{(i-1)})$ and

$$s'_i = \begin{cases} l.\mathit{put}\, v_i\, (c_i, r_i) & i \in \{1, \ldots, \min(n, m)\} \\ l.\mathit{create}\, v_i\, r_i & i \in \{m+1, \ldots, n\} \end{cases}$$

for $i \in \{1, \ldots, n\}$. We analyze several cases.

Case $x \in \mathit{locs}(v_1)$: We calculate as follows

$$\begin{array}{ll}
\quad (l^*.\mathit{put}\ (v_1 \cdots v_n)\ ([c_1, \ldots, c_m], r))[x] & \\
= (s'_1 \cdots s'_n)[x] & \text{by definition } l^*.\mathit{put}, \\
= s'_1[x] & \text{by definition } [\cdot] \\
\quad\ \text{with Lemmas 4.1.3 and 4.1.4} & \\
= k.\mathit{put}\ v_1\ (r_1(x)) & \text{by } \textsc{ChunkCreate} \\
 & \text{and } \textsc{ChunkPut} \text{ for } l \\
= k.\mathit{put}\ (v_1 \cdots v_n)[x]\ ((r_1 +\!\!+ \ldots +\!\!+ r_n)(x)) & \text{by definition } [\cdot] \\
\quad\ \text{with definition } (+\!\!+) \text{ and } r_1 \text{ to } r_n & \\
= k.\mathit{put}\ (v_1 \cdots v_n)[x]\ (r(x)) & \text{by definition application}
\end{array}$$

and obtain the required equality.

Case $x \notin \mathit{locs}(v_1)$: Similar to the previous case.

▶ **ChunkCreate**: Similar to the proof of ChunkPut.

▶ **NoChunkPut**: Similar to the proof of ChunkPut.

▶ **NoChunkCreate**: Similar to the proof of ChunkPut.

▶ **SkelPut**: Let $v = v_1 \cdots v_n$ and $v' = v_1 \cdots v_m$ be strings in V^*, let $[c_1, \ldots, c_o]$ be a rigid complement in $C\ \mathit{list}$, and let r and r' be resources in $\{|\mathbb{N} \mapsto k.C|\}$ such that $\mathit{skel}(v) = \mathit{skel}(v')$. By $\lfloor V \rfloor^{!*}$ and as V^* is chunk unambiguous, we have $m = n$ and $\mathit{skel}(v_i) = \mathit{skel}(v'_i)$ for i from 1 to n. To shorten the proof, let

$$\begin{array}{ll}
r'_0 = r & (r_i, r'_i) = \mathit{split}(|v_i|, r'_{(i-1)}) \\
r''_0 = r' & (r''_i, r'''_i) = \mathit{split}(|v'_i|, r'''_{(i-1)})
\end{array}$$

and

$$s'_i = \begin{cases} l.\mathit{put}\, v_i\, (c_i, r_i) & i \in \{1, \ldots, \min(n, o)\} \\ l.\mathit{create}\, v_i\, r_i & i \in \{o+1, \ldots, n\} \end{cases}$$

and

$$s''_i = \begin{cases} l.\mathit{put}\, v'_i\, (c_i, r'_i) & i \in \{1, \ldots, \min(n, o)\} \\ l.\mathit{create}\, v'_i\, r'_i & i \in \{o+1, \ldots, n\} \end{cases}$$

for i from 1 to n. Using these facts and definitions, we calculate as follows

$$\begin{array}{ll}
\quad \mathit{skel}(l^*.\mathit{put}\ (v_1 \cdots v_n)\ ([c_1, \ldots, c_o], r)) & \\
= \mathit{skel}(s'_1 \cdots s'_n) & \text{by the definition of } l^*.\mathit{put} \\
= (\mathit{skel}(s'_1)) \cdots (\mathit{skel}(s'_n)) & \text{by the definition of } \mathit{skel} \\
= (\mathit{skel}(s''_1)) \cdots (\mathit{skel}(s''_n)) & \text{by } \textsc{SkelCreate} \\
 & \text{and } \textsc{SkelPut} \text{ for } l \\
= \mathit{skel}(s''_1 \cdots s''_n) & \text{by the definition of } \mathit{skel} \\
= \mathit{skel}(l^*.\mathit{put}\ (v'_1 \cdots v'_n)\ ([c_1, \ldots, c_o], r')) & \text{by definition } l^*.\mathit{put}
\end{array}$$

and obtain the required equality.

▶ **SkelCreate**: Similar to the proof of SkelPut. □

$$\boxed{\dfrac{\begin{array}{c} \lfloor S_1 \rfloor \cap \lfloor S_2 \rfloor = \emptyset \qquad \lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor \subseteq \lfloor V_1 \cap V_2 \rfloor \\ l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 \\ l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 \end{array}}{l_1 \mid l_2 \in (S_1 \cup S_2) \overset{(C_1+C_2),k}{\Longleftrightarrow} (V_1 \cup V_2)}}$$

4.2.5 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be resourceful lenses such that $\lfloor S_1 \rfloor \cap \lfloor S_2 \rfloor = \emptyset$ and $\lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor \subseteq \lfloor V_1 \cap V_2 \rfloor$. Then $l_1 \,|\, l_2$ is a resourceful lens in $(S_1 \cup S_2) \overset{(C_1+C_2),k}{\Longleftrightarrow} (V_1 \cup V_2)$.

Proof: We prove each resourceful lens law separately.
▶ **GetPut**: Let s be a string in $S_1 \cup S_2$. We analyze two cases.

Case $s \in S_1$**:** We calculate as follows

$$
\begin{array}{ll}
\quad (l_1 \,|\, l_2).\mathit{put}\ ((l_1 \,|\, l_2).\mathit{get}\ s)\ ((l_1 \,|\, l_2).\mathit{res}\ s) & \\
= (l_1 \,|\, l_2).\mathit{put}\ (l_1.\mathit{get}\ s)\ ((l_1 \,|\, l_2).\mathit{res}\ s) & \text{by definition } (l_1 \,|\, l_2).\mathit{get} \\
\qquad \text{with } s \in S_1 & \\
= (l_1 \,|\, l_2).\mathit{put}\ (l_1.\mathit{get}\ s)\ (\mathit{Inl}(c_1), r) & \text{by definition } (l_1 \,|\, l_2).\mathit{res} \\
\qquad \text{with } s \in S_1 & \\
\qquad \text{where } (c_1, r) = l_1.\mathit{res}\ s & \\
= l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ (c_1, r) & \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
\qquad \text{with } \mathrm{cod}(l.\mathit{get}) = V_1 & \\
= l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ (l_1.\mathit{res}\ s) & \text{by definition } (c_1, r) \\
= s & \text{by } \textsc{PutGet} \text{ for } l_1
\end{array}
$$

and obtain the required equality.

Case $s \in S_2$**:** Symmetric to the previous case.

▶ **PutGet**: Let v be a string in $V_1 \cup V_2$, let c be a rigid complement in $C_1 + C_2$, and let r be a resource in $\{\!|\mathbb{N} \mapsto k.C|\!\}$. We analyze several cases.

Case $v \in V_1$ **and** $c = \mathit{Inl}(c_1)$**:** We calculate as follows

$$
\begin{array}{ll}
\quad (l_1 \,|\, l_2).\mathit{get}\ ((l_1 \,|\, l_2).\mathit{put}\ v\ (c, r)) & \\
= (l_1 \,|\, l_2).\mathit{get}\ (l_1.\mathit{put}\ v\ (c_1, r)) & \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
\qquad \text{with } v \in V_1 \text{ and } c = \mathit{Inl}(c_1) & \\
= l_1.\mathit{get}\ (l_1.\mathit{put}\ v\ (c_1, r)) & \text{by definition } (l_1 \,|\, l_2).\mathit{get} \\
\qquad \text{with } \mathrm{cod}(l_1.\mathit{put}) = S_1 & \\
= v & \text{by } \textsc{PutGet} \text{ for } l
\end{array}
$$

and obtain the required equality.

Case $v \in V_2$ **and** $c = \mathit{Inr}(c_2)$**:** Symmetric to the previous case.

Case $v \notin V_2$ **and** $c = \mathit{Inr}(c_2)$**:** We calculate as follows

$$
\begin{array}{ll}
\quad (l_1 \,|\, l_2).\mathit{get}\ ((l_1 \,|\, l_2).\mathit{put}\ v\ (c, r)) & \\
= (l_1 \,|\, l_2).\mathit{get}\ (l_1.\mathit{create}\ v\ r) & \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
\qquad \text{with } v \notin V_2 \text{ and } c = \mathit{Inr}(c_2) & \\
= l_1.\mathit{get}\ (l_1.\mathit{create}\ v\ r) & \text{by definition } (l_1 \,|\, l_2).\mathit{get} \\
\qquad \text{with } \mathrm{cod}(l_1.\mathit{create}) = S_1 & \\
= v & \text{by } \textsc{CreateGet} \text{ for } l
\end{array}
$$

and obtain the required equality.

Case $v \notin V_1$ **and** $c = \mathit{Inl}(c_1)$**:** Symmetric to the previous case.

▶ **CreateGet**: Similar to the proof of PUTGET.

▶ **GetChunks**: Let s be a string in $S_1 \cup S_2$. As $\lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor \subseteq \lfloor V_1 \cap V_2 \rfloor$, for every $v \in V_1 \cap V_2$ the chunks of v as identified by V_1 and by V_2 are identical. We analyze two cases.

Case $s \in S_1$**:** We calculate as follows

$$\begin{array}{rll} locs(s) & = locs(l_1.get\ s) & \text{by GETCHUNKS for } l_1 \\ & = locs((l_1 \mid l_2).get\ s) & \text{by definition } (l_1 \mid l_2).get \\ & \quad \text{with } s \in S_1 & \end{array}$$

and obtain the required equality.

Case $s \in S_2$**:** Symmetric to the previous case.

▶ **ResChunks**: Let s be a string in $S_1 \cup S_2$, let c be a rigid complement in $C_1 + C_2$, and let r be a resource in $\{|\mathbb{N} \mapsto k.C|\}$ with $(c, r) = (l_1 \mid l_2).res\ s$. We analyze two cases.

Case $s \in S_1$**:** By the assumption of the case and the definition of $(l_1 \mid l_2).res$ we have $c = Inl(c_1)$ where $c_1, r = l_1.res\ s$. The required equality, $locs(s) = \mathrm{dom}(r)$, is immediate by RESCHUNKS for l_1.

Case $s \in S_2$**:** Symmetric to the previous case.

▶ **ChunkPut**: Let v be a string in $V_1 \cup V_2$, let c be a rigid complement in $C_1 + C_2$, let r be a resource in $\{|\mathbb{N} \mapsto k.C|\}$, and let x be a location in $locs(v) \cap \mathrm{dom}(r)$. We analyze several cases.

Case $v \in V_1$ **and** $c = Inl(c_1)$**:** As $\lfloor V_1 \rfloor \cap \lfloor V_2 \rfloor \subseteq \lfloor V_1 \cap V_2 \rfloor$, for every $v \in V_1 \cap V_2$ we have that if x is a location of a chunk in v as specified by V_1, then it is also a chunk in v as specified by V_2, and vice versa. Using this fact, we calculate as follows

$$\begin{array}{ll} \quad (l_1 \mid l_2).put\ v\ (c, r)[x] & \\ = l_1.put\ v\ (c_1, r)[x] & \text{by definition } (l_1 \mid l_2).put \\ \quad \text{with } v \in V_1 \text{ and } c = Inl(c_1) & \\ = k.put\ (v[x])\ (r(x)) & \text{by CHUNKPUT for } l_1 \end{array}$$

and obtain the required equality.

Case $v \in V_2$ **and** $c = Inr(c_2)$**:** Symmetric to the previous case.

Case $v \notin V_2$ **and** $c = Inr(c_2)$**:** We calculate as follows.

$$\begin{array}{ll} \quad (l_1 \mid l_2).put\ v\ (c, r)[x] & \\ = l_1.create\ v\ r[x] & \text{by definition } (l_1 \mid l_2).put \\ \quad \text{with } v \notin V_2 \text{ and } c = Inr(c_2) & \\ = k.put\ (v[x])\ (r(x)) & \text{by CHUNKCREATE for } l_1 \end{array}$$

and obtain the required equality.

Case $v \notin V_1$ **and** $c = Inl(c_1)$**:** Symmetric to the previous case.

▶ **ChunkCreate**: Similar to the proof of CHUNKPUT.

▶ **NoChunkPut**: Similar to the proof of CHUNKPUT.

▶ **NoChunkCreate**: Similar to the proof of CHUNKPUT.

▶ **SkelPut**: Let v and v' be strings in $V_1 \cup V_2$, let c be a rigid complement in $C_1 + C_2$, and let r and r' be resources in $\{|\mathbb{N} \mapsto k.C|\}$ such that $skel(v) = skel(v')$. We analyze several cases.

Case $v \in V_1$ and $v' \in V_1$ and $c = \mathit{Inl}(c_1)$: We calculate as follows

$$\begin{array}{ll}
\quad \mathit{skel}((l_1 \,|\, l_2).\mathit{put}\ v\ (c, r)) & \\
= \mathit{skel}(l_1.\mathit{put}\ v\ (c_1, r)) & \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
\qquad \text{with } v \in V_1 \text{ and } c = \mathit{Inl}(c_1) & \\
= \mathit{skel}(l_1.\mathit{put}\ v'\ (c_1, r')) & \text{by S}{\small\text{KEL}}\text{P}{\small\text{UT}} \text{ for } l_1 \\
= \mathit{skel}((l_1 \,|\, l_2).\mathit{put}\ v'\ (c, r')) & \text{by definition } (l_1 \,|\, l_2).\mathit{put} \\
\qquad \text{with } v' \in V_1 \text{ and } c = \mathit{Inl}(c_1) &
\end{array}$$

and obtain the required equality.

Case $v \in V_2$ and $v' \in V_2$ and $c = \mathit{Inr}(c_2)$: Similar to the previous case.

Case $v \in V_1$ and $v' \in V_1$ and $c = \mathit{Inr}(c_2)$: Similar to the first case.

Case $v \in V_2$ and $v' \in V_2$ and $c = \mathit{Inl}(c_1)$: Similar to the first case.

Case $v \in V_1$ and $v' \notin V_1$: Can't happen. As $\mathit{skel}(v) = \mathit{skel}(v')$, we have the sets of locations $\mathit{locs}(v)$ and $\mathit{locs}(v')$ are identical. Let v'' be the string obtained from v by setting the chunk at every location in $\mathit{locs}(v)$ to the corresponding chunk in v'. By construction, we have $v' = v'$. By chunk compatibility we also have $v'' \in V_1$. However, by the assumptions of the case, we have $v' \notin V_1$, which is a contradiction.

Case $v \in V_2$ and $v' \notin V_2$: Symmetric to the previous case.

► **SkelCreate:** Similar to the proof of SKELPUT. □

$$\boxed{\dfrac{\begin{array}{c} l_1 \in S \overset{C_1,k_1}{\Longleftrightarrow} U \\ l_2 \in U \overset{C_2,k_2}{\Longleftrightarrow} V \end{array}}{l_1;l_2 \in S \overset{(C_1 \otimes C_2),(k_1;k_2)}{\Longleftrightarrow} V}}$$

4.2.6 Lemma: Let $l_1 \in S \overset{C_1,k_1}{\Longleftrightarrow} U$ and $l_2 \in U \overset{C_2,k_2}{\Longleftrightarrow} V$ be resourceful lenses. Then $(l_1;l_2)$ is a resourceful lens in $S \overset{(C_1 \otimes C_2),(k_1;k_2)}{\Longleftrightarrow} V$.

Proof: We prove each resourceful lens law separately.
► **GetPut:** Let $s \in S$. We calculate as follows

$$\begin{array}{ll}
\quad (l_1;l_2).\mathit{put}\ ((l_1;l_2).\mathit{get}\ s)\ ((l_1;l_2).\mathit{res}\ s) & \\
= (l_1;l_2).\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (\langle c_1, c_2 \rangle, \mathit{zip}\ r_1\ r_2) & \text{by definition } (l_1;l_2).\mathit{get} \\
\qquad \text{where } c_1, r_1 = l_1.\mathit{res}\ s & \text{and } (l_1;l_2).\mathit{res} \\
\qquad\quad \text{and } c_2, r_2 = l_2.\mathit{res}\ (l_1.\mathit{get}\ s) & \\
= l_1.\mathit{put}\ (l_2.\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (c_2, r_2'))\ (c_1, r_1') & \text{by definition } (l_1;l_2).\mathit{put} \\
\qquad \text{where } r_1', r_2' = \mathit{unzip}\ (\mathit{zip}\ r_1\ r_2) & \\
= l_1.\mathit{put} & \\
\qquad (l_2.\mathit{put}\ (l_2.\mathit{get}\ (l_1.\mathit{get}\ s))\ (l_2.\mathit{res}\ (l_1.\mathit{get}\ s))) & \\
\qquad (l_1.\mathit{res}\ s) & \text{by definition } (c_1, r_1) \\
\qquad \text{with } \mathit{unzip}(\mathit{zip}\ r_1\ r_2) = r_1, r_2 & \text{and } (c_2, r_2) \\
= l_1.\mathit{put}\ (l_1.\mathit{get}\ s)\ (l_1.\mathit{res}\ s) & \text{by G}{\small\text{ET}}\text{P}{\small\text{UT}} \text{ for } l_2 \\
= s & \text{by G}{\small\text{ET}}\text{P}{\small\text{UT}} \text{ for } l_1
\end{array}$$

and obtain the required equality.

▶ **PutGet**: Let v be a view in V, let $\langle c_1, c_2\rangle$ be a rigid complement in $C_1 \otimes C_2$, and let r be a resource in $\{|\mathbb{N} \mapsto (k_1;k_2).C|\}$. We calculate as follows

$$\begin{array}{ll}
\quad (l_1;l_2).\textit{get}\ ((l_1;l_2).\textit{put}\ v\ (\langle c_1, c_2\rangle, r)) & \\
= (l_1;l_2).\textit{get}\ (l_1.\textit{put}\ (l_2.\textit{put}\ v\ (c_2, r_2))\ (c_1, r_1)) & \text{by definition } (l_1;l_2).\textit{put} \\
\qquad \text{where } r_1, r_2 = \textit{unzip}\ r & \\
= l_2.\textit{get}\ (l_1.\textit{get}\ (l_1.\textit{put}\ (l_2.\textit{put}\ v\ (c_2, r_2))\ (c_1, r_1))) & \text{by definition } (l_1;l_2).\textit{get} \\
= l_2.\textit{get}\ (l_2.\textit{put}\ v\ (c_2, r_2)) & \text{by PUTGET for } l_1 \\
= v & \text{by PUTGET for } l_2
\end{array}$$

and obtain the required equality.

▶ **CreateGet**: Similar to the proof of PUTGET.

▶ **GetChunks**: Let s be a string in S. We calculate as follows

$$\begin{array}{rll}
\textit{locs}(s) & = \textit{locs}(l_1.\textit{get}\ s) & \text{by GETCHUNKS for } l_1 \\
& = \textit{locs}(l_2.\textit{get}\ (l_1.\textit{get}\ s)) & \text{by GETCHUNKS for } l_2 \\
& = \textit{locs}((l_1;l_2).\textit{get}\ s) & \text{by definition } (l_1;l_2).\textit{get}
\end{array}$$

and obtain the required equality.

▶ **ResChunks**: Let s be a string in S, let $\langle c_1, c_2\rangle$ be a rigid complement in $C_1 \otimes C_2$, and let r be a resource in $\{|\mathbb{N} \mapsto (k_1;k_2).C|\}$ a resource with $(c, r) = (l_1;l_2).\textit{res}\ s$. The proof goes in three steps.

First, we show that the set of locations in s is equal to the domain of the resource computed from s using $l_1.\textit{res}$,

$$\textit{locs}(s) = \text{dom}(r_1) \quad \text{by RESCHUNKS for } l_1$$

where $c_1, r_1 = l_1.\textit{res}\ s$.

Next, we show that the set of locations in s is equal to the domain of the resource computed from $l_1.\textit{get}\ s$ using $l_2.\textit{res}$,

$$\begin{array}{rll}
\textit{locs}(s) & = \textit{locs}(l_1.\textit{get}\ s) & \text{by GETCHUNKS for } l_1 \\
& \text{dom}(r_2) & \text{by RESCHUNKS for } l_2
\end{array}$$

where $c_2, r_2 = l_2.\textit{res}\ (l_1.\textit{get}\ s)$.

Finally, using both of these facts, we calculate as follows

$$\begin{array}{ll}
\quad \textit{locs}(s) & \\
= \text{dom}(\textit{zip}\ r_1\ r_2) & \text{by definition } \textit{zip} \\
\qquad \text{with } \text{dom}(r_1) = \text{dom}(r_2) & \\
= \text{dom}(r) & \text{by definition } (l_1;l_2).\textit{res}
\end{array}$$

and obtain the required equality.

▶ **ChunkPut**: Let v be a string in $\lfloor V \rfloor$, let $\langle c_1, c_2\rangle$ be a rigid complement in $C_1 \otimes C_2$ be a rigid complement, let r be a resource in $\{|\mathbb{N} \mapsto (k_1;k_2).C|\}$, and let x be a location in $\textit{locs}(v) \cap \text{dom}(r)$. We calculate as follows

$$\begin{array}{ll}
\quad ((l_1;l_2).\textit{put}\ v\ (\langle c_1, c_2\rangle, r))[x] & \\
= (l_1.\textit{put}\ (l_2.\textit{put}\ (c_2, r_2))\ (c_1, r_1))[x] & \text{by definition } (l_1;l_2).\textit{put} \\
\quad \text{where } r_1, r_2 = \textit{unzip}\ r & \\
= k_1.\textit{put}\ ((l_2.\textit{put}\ v\ (c_2, r_2))[x])\ (r_1(x)) & \text{by CHUNKPUT for } l_1 \\
= k_1.\textit{put}\ (k_2.\textit{put}\ (v[x])\ (r_2(x)))\ (r_1(x)) & \text{by CHUNKPUT for } l_2 \\
= (k_1;k_2).\textit{put}\ (v[x])\ \langle r_1(x), r_2(x)\rangle & \text{by definition of } (k_1;k_2).\textit{put} \\
= (k_1;k_2).\textit{put}\ (v[x])\ (r(x)) & \text{by definition } (r_1, r_2) \text{ and } \textit{unzip}
\end{array}$$

and obtain the required equality.

▶ **ChunkCreate:** Similar to the proof of ChunkPut.

▶ **NoChunkPut:** Similar to the proof of ChunkPut.

▶ **NoChunkCreate:** Similar to the proof of ChunkPut.

▶ **SkelPut:** Let v and v' be strings in V, let $\langle c_1, c_2\rangle \in C_1 \otimes C_2$ be a rigid complement, and let r and r' be resources in $\{\!|\mathbb{N} \mapsto k.C|\!\}$ with $skel(v) = skel(v')$. To shorten the proof, let r_1 and r_2 and r'_1 and r'_2 be resources and let u and u' be strings defined as follows:

$$\begin{array}{rcl} r_1, r_2 & = & unzip\ r \\ r'_1, r'_2 & = & unzip\ r' \\ u & = & l_2.\boldsymbol{put}\ v\ (c_2, r_2) \\ u' & = & l_2.\boldsymbol{put}\ v'\ (c_2, r'_2) \end{array}$$

Observe that $skel(u) = skel(u')$ by SkelPut for l_2. Using these facts and definitions, we calculate as follows

$$\begin{array}{ll} \quad skel((l_1;l_2).\boldsymbol{put}\ v\ (\langle c_1, c_2\rangle, r)) & \\ = skel(l_1.\boldsymbol{put}\ (l_2.\boldsymbol{put}\ v\ (c_2, r_2))\ (c_1, r_1)) & \text{by definition } (l_1;l_2).\boldsymbol{put} \\ = skel(l_1.\boldsymbol{put}\ u\ (c_1, r_1)) & \text{by definition } u \\ = skel(l_1.\boldsymbol{put}\ u'\ (c_1, r'_1)) & \text{by SkelPut for } l_1 \\ = skel(l_1.\boldsymbol{put}\ (l_2.\boldsymbol{put}\ v'\ (c_2, r'_2))\ (c_1, r'_1)) & \text{by definition } u' \\ = skel((l_1;l_2).\boldsymbol{put}\ v'\ (\langle c_2, c_2\rangle, r')) & \text{by definition } (l_1;l_2).\boldsymbol{put} \end{array}$$

and obtain the required equality.

▶ **SkelCreate:** Similar to the proof of SkelPut. □

$$\boxed{\dfrac{\begin{array}{ll} l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 & \lfloor S_1\rfloor \cdot^{!} \lfloor S_2\rfloor \\ l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 & \lfloor V_2\rfloor \cdot^{!} \lfloor V_1\rfloor \end{array}}{l_1 \sim l_2 \in (S_1 \cdot S_2) \overset{(C_2 \times C_1),k}{\Longleftrightarrow} (V_2 \cdot V_1)}}$$

4.4.1 Lemma: Let $l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2$ be resourceful lenses such that $\lfloor S_1\rfloor \cdot^{!} \lfloor S_2\rfloor$ and $\lfloor V_1\rfloor \cdot^{!} \lfloor V_2\rfloor$. Then $l_1 \sim l_2$ is a resourceful lens in $(S_1 \cdot S_2) \overset{(C_2 \times C_1),k}{\Longleftrightarrow} (V_2 \cdot V_1)$.

Proof: We prove each resourceful lens law separately.

▶ **GetPut:** Let $s = s_1 \cdot s_2$ be a string in $S_1 \cdot S_2$. We calculate as follows

$$\begin{aligned}
& (l_1 \sim l_2).\mathit{put} \\
& \quad ((l_1 \sim l_2).\mathit{get}\ (s_1 \cdot s_2))\ ((l_1 \sim l_2).\mathit{res}\ (s_1 \cdot s_2)) \\
&= (l_1 \sim l_2).\mathit{put} \\
& \quad ((l_2.\mathit{get}\ s_2) \cdot (l_1.\mathit{get}\ s_1))\ ((l_1 \sim l_2).\mathit{res}\ (s_1 \cdot s_2)) && \text{by definition } (l_1 \sim l_2).\mathit{get} \\
&= (l_1 \sim l_2).\mathit{put} \\
& \quad ((l_2.\mathit{get}\ s_2) \cdot (l_1.\mathit{get}\ s_1))\ ((c_2, c_1), r_2 +\!\!+\, r_1) && \text{by definition } (l_1 \sim l_2).\mathit{res} \\
& \quad \text{where } c_1, r_1 = l_1.\mathit{res}\ s_1 \\
& \quad \quad \text{and } c_2, r_2 = l_2.\mathit{res}\ s_2 \\
&= (l_1.\mathit{put}\ (l_1.\mathit{get}\ s_1)\ (c_1, r'_1)) \cdot && \text{by definition } (l_1 \sim l_2).\mathit{put} \\
& \quad (l_2.\mathit{put}\ (l_2.\mathit{get}\ s_2)\ (c_2, r'_2)) \\
& \quad \text{with } \lfloor V_2 \rfloor \cdot^{!} \lfloor V_1 \rfloor \text{ and } \mathrm{cod}(l_2.\mathit{get}) = V_2 \text{ and } \mathrm{cod}(l_1.\mathit{get}) = V_1 \\
& \quad \text{where } r'_2, r'_1 = \mathit{split}(|l_2.\mathit{get}\ s_2|, r_2 +\!\!+\, r_1) \\
&= (l_1.\mathit{put}\ (l_1.\mathit{get}\ s_1)\ (c_1, r_1)) \cdot && \text{by } \textsc{GetChunks} \text{ for } l_1 \\
& \quad (l_2.\mathit{put}\ (l_2.\mathit{get}\ s_2)\ (c_2, r_2)) && \text{and } l_2 \\
& \quad \text{with } \textsc{ResChunks} \text{ for } l_1 \text{ and definition } \mathit{split} \\
&= s_1 \cdot s_2 && \text{by } \textsc{GetPut} \text{ for } l_1 \text{ and } l_2
\end{aligned}$$

and obtain the required equality.

▶ **PutGet:** Let $v = v_1 \cdot v_2$ be a string in $V_1 \cdot V_2$, let (c_2, c_1) be a rigid complement in $C_2 \times C_1$, and let r be a resource in $\{\!|\mathbb{N} \mapsto k.C|\!\}$. We calculate as follows

$$\begin{aligned}
& (l_1 \sim l_2).\mathit{get} \\
& \quad ((l_1 \sim l_2).\mathit{put}\ (v_2 \cdot v_1)\ ((c_2, c_1), r)) \\
&= (l_1 \sim l_2).\mathit{get} \\
& \quad ((l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\ v_2\ (c_2, r_2))) && \text{by definition } (l_1 \sim l_2).\mathit{put} \\
& \quad \text{where } r_2, r_1 = \mathit{split}(|v_2|, r) \\
&= (l_2.\mathit{get}\ (l_2.\mathit{put}\ v_2\ (c_2, r_2))) \cdot \\
& \quad (l_1.\mathit{get}\ (l_1.\mathit{put}\ v_1\ (c_1, r_1))) && \text{by definition } (l_1 \sim l_2).\mathit{get} \\
& \quad \text{with } \lfloor S_1 \rfloor \cdot^{!} \lfloor S_2 \rfloor \text{ and } \mathrm{cod}(l_1.\mathit{put}) = S_1 \text{ and } \mathrm{cod}(l_2.\mathit{put}) = S_2 \\
&= v_2 \cdot v_1 && \text{by } \textsc{PutGet} \text{ for } l_2 \text{ and } l_1
\end{aligned}$$

and obtain the required equality.

▶ **CreateGet:** Similar to the proof of PutGet.

▶ **GetChunks:** Let $s = s_1 \cdot s_2$ be a string in $S_1 \cdot S_2$. We calculate as follows

$$\begin{aligned}
&= \mathit{locs}(s_1 \cdot s_2) \\
&= \{1, \ldots, |s_1| + |s_2|\} && \text{by definition } \mathit{locs} \\
&= \{1, \ldots, |l_1.\mathit{get}\ s_1| + |l_2.\mathit{get}\ s_2|\} && \text{by } \textsc{GetChunks} \text{ for } l_1 \text{ and } l_2 \\
&= \mathit{locs}((l_2.\mathit{get}\ s_2) \cdot (l_1.\mathit{get}\ s_1)) && \text{by definition } \mathit{locs} \\
&= \mathit{locs}((l_1 \sim l_2).\mathit{get}\ (s_1 \cdot s_2)) && \text{by definition } (l_1 \sim l_2).\mathit{get}
\end{aligned}$$

and obtain the required equality.

▶ **ResChunks:** Let $s = s_1 \cdot s_2$ be a string in $S_1 \cdot S_2$, let (c_2, c_1) be a rigid complement in $C_2 \times C_1$, and let

r be a resource in $\{\!|\mathbb{N} \mapsto k.C|\!\}$ with $((c_1, c_2), r) = (l_1 \sim l_2).\mathit{res}\ s$. We calculate as follows

$$
\begin{array}{ll}
\quad \mathrm{dom}(r) & \\
= \mathrm{dom}(r_2 \mathbin{++} r_1) & \text{by definition } (l_1 \sim l_2).\mathit{res} \\
\qquad \text{where } r_1, c_1 = l_1.\mathit{res}\ s_1 & \\
\qquad \text{and } r_2, c_2 = l_2.\mathit{res}\ s_2 & \\
= \mathrm{dom}(r_2) \cup \{i + |r_2| \mid i \in \mathrm{dom}(r_1)\} & \text{by definition } (\mathbin{++}) \text{ and dom} \\
= \mathit{locs}(s_2) \cup \{i + |s_2| \mid i \in \mathit{locs}(s_1)\} & \text{by } \textsc{ResChunks} \text{ for } l_2 \text{ and } l_1 \\
= \mathit{locs}(s_1 \cdot s_2) & \text{by definition } \mathit{locs}
\end{array}
$$

and obtain the required equality.

▶ **ChunkPut**: Let $v = v_1 \cdot v_2$ be a string in $V_2 \cdot V_1$, let (c_2, c_1) be a rigid complement in $C_2 \times C_1$, let r be a resource in $\{\!|\mathbb{N} \mapsto k.C|\!\}$, and let x be a location in $\mathit{locs}(v) \cap \mathrm{dom}(r)$. To shorten the proof, define the following resources and permutations:

$$
\begin{array}{rclcrcl}
q_2 & = & l_2.\mathit{perm}\ (l_2.\mathit{put}\ v_2\ (c_2, r_2)) & \qquad & r_2, r_1 & = & \mathit{split}(|v_2|, r) \\
q_1 & = & l_1.\mathit{perm}\ (l_1.\mathit{put}\ v_1\ (c_1, r_1)) & & q & = & (q_2 \ast\ast\, q_1)
\end{array}
$$

We analyze two cases.

Case $x \in \mathit{locs}(v_2)$: Let y be the unique location satisfying $q(y) = x$. From $x \in \mathit{locs}(v_2)$ we have that $y > n_1$ and $q(y) = q_2(y - n_1)$ where $n_1 = |l.\mathit{put}\ v_1\ (c_1, r_1)|$. Using these facts, we calculate as follows

$$
\begin{array}{ll}
\quad ((l_1 \sim l_2).\mathit{put}\ (v_2 \cdot v_1)\ ((c_2, c_1), r))[y] & \\
= ((l_1.\mathit{put}\, v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\, v_2\ (c_2, r_2)))[y] & \text{by definition } (l_1 \sim l_2).\mathit{put} \\
\qquad \text{where } r_2, r_1 = \mathit{split}(|v_2|, r) & \\
= l_2.\mathit{put}\ v_2\ (c_2, r_2)[y - n_1] & \text{by definition } [\cdot] \\
= k.\mathit{put}\ v_2[x]\ (r_2(x)) & \text{by } \textsc{ChunkPut} \text{ for } l_2 \\
= k.\mathit{put}\ (v_1 \cdot v_2)[x]\ ((r_2 \mathbin{++} r_1)(x)) & \text{by definition } [\cdot] \\
= k.\mathit{put}\ (v_1 \cdot v_2)[x]\ (r(x)) & \text{by definition } \mathit{split}, \\
 & r_1, \text{ and } r_2
\end{array}
$$

and obtain the required equality.

Case $x \notin \mathit{locs}(v_2)$: Similar to the previous case.

▶ **ChunkCreate**: Similar to the proof of ChunkPut.

▶ **SkelPut**: Let $v = v_2 \cdot v_1$ and $v' = v'_2 \cdot v'_1$ be strings in $V_2 \cdot V_1$, let (c_2, c_1) be a rigid complement in $C_2 \times C_1$, and let r and r' be resources in $r \in \{\!|\mathbb{N} \mapsto k.C|\!\}$ such that $\mathit{skel}(v) = \mathit{skel}(v')$. By $\lfloor V_2 \rfloor \cdot^{!} \lfloor V_1 \rfloor$ and as $V_2 \cdot V_1$ is chunk unambiguous, we have $\mathit{skel}(v_2) = \mathit{skel}(v'_2)$ and $\mathit{skel}(v_1) = \mathit{skel}(v'_1)$. Using these facts and definitions, we calculate as follows

$$
\begin{array}{ll}
\quad \mathit{skel}((l_1 \sim l_2).\mathit{put}\ (v_2 \cdot v_1)\ ((c_1, c_2), r)) & \\
= \mathit{skel}(l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot (l_2.\mathit{put}\ v_2\ (c_2, r_2)) & \text{by definition } (l_1 \sim l_2).\mathit{put} \\
\qquad \text{where } r_2, r_1 = \mathit{split}(|v_2|, r) & \\
= \mathit{skel}(l_1.\mathit{put}\ v_1\ (c_1, r_1)) \cdot \mathit{skel}(l_2.\mathit{put}\ v_2\ (c_2, r_2)) & \text{by definition } \mathit{skel} \\
= \mathit{skel}(l_1.\mathit{put}\ v'_1\ (c_1, r'_1)) \cdot \mathit{skel}(l_2.\mathit{put}\ v'_2\ (c_2, r'_2)) & \text{by } \textsc{SkelPut} \text{ for } l_1 \text{ and } l_2 \\
\qquad \text{where } r'_2, r'_1 = \mathit{split}(|v'_2|, r') & \\
= \mathit{skel}(l_1.\mathit{put}\ v'_1\ (c_1, r'_1)) \cdot (l_2.\mathit{put}\ v'_2\ (c_2, r'_2)) & \text{by definition } \mathit{skel} \\
= \mathit{skel}((l_1 \sim l_2).\mathit{put}\ (v'_2 \cdot v'_1)\ ((c_1, c_2), r')) & \text{by definition } (l_1 \sim l_2).\mathit{put}, \\
 & r'_1, \text{ and } r'_2
\end{array}
$$

and obtain the required equality.

▶ **SkelCreate:** Similar to the proof of SkelPut. □

Secure Lens Proofs

5.1.1 Lemma: The `redact` lens is a secure lens at the following type:

$$\begin{array}{rl} & (\,(\mathtt{SPACE}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}):\mathsf{Tainted} \\ & \mid(\mathtt{ASTERISK}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}):\mathsf{Trusted})^{*} \\ \overset{\text{🔒}}{\Longleftrightarrow} & (\,(\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{NEWLINE}):\mathsf{Tainted} \\ & \mid(\mathtt{TIME}\cdot\mathtt{BUSY}\cdot\mathtt{NEWLINE}):\mathsf{Trusted})^{*} \end{array}$$

Proof: Suppose that we have annotated some of the regular expressions in the `redact` lens with security labels indicating that the data handled by the `public` lens is tainted:

```
let public : lens =
    del (SPACE:Tainted)
  . copy ((TIME . DESC):Tainted)
  . del (LOCATION:Tainted)
  . copy (NEWLINE:Tainted)

let private : lens =
    del ASTERISK
  . copy TIME
  . (DESC . LOCATION) <-> "BUSY"
  . copy NEWLINE

let redact : lens =
  public* . ( private . public* )*
```

We do not explicitly add annotations for Trusted data since every regular expression R is equivalent to R:Trusted in the two-point integrity lattice.

By the typing rules for `del`, `copy`, `<->`, and concatenation we have:

$$\begin{array}{rl} \mathtt{public} \in & \\ & (\mathtt{SPACE}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}):\mathsf{Tainted} \\ \overset{\text{🔒}}{\Longleftrightarrow} & (\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{NEWLINE}):\mathsf{Tainted} \\ \mathtt{private} \in & \\ & (\mathtt{ASTERISK}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}) \\ \overset{\text{🔒}}{\Longleftrightarrow} & (\mathtt{TIME}\cdot\mathtt{BUSY}\cdot\mathtt{NEWLINE}) \end{array}$$

The syntactic type that would be computed mechanically using our typing rules is slightly more complicated but semantically equivalent. We use such equivalences throughout this proof.

By the typing rule for iteration we have:

$$\begin{array}{rl} \mathtt{public*} \in & \\ & ((\mathtt{SPACE}\cdot\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{LOCATION}\cdot\mathtt{NEWLINE}):\mathsf{Tainted})^{*} \\ \overset{\text{🔒}}{\Longleftrightarrow} & ((\mathtt{TIME}\cdot\mathtt{DESC}\cdot\mathtt{NEWLINE}):\mathsf{Tainted})^{*} \end{array}$$

Next, by the typing rule for concatenation, and as Trusted observes the unambiguous concatenability of the types in the view, we have:

$$\begin{array}{l}
\texttt{private . public*} \in \\
\quad (\texttt{ASTERISK}\cdot\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{LOCATION}\cdot\texttt{NEWLINE})\cdot \\
\quad ((\texttt{SPACE}\cdot\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{LOCATION}\cdot\texttt{NEWLINE})\colon \mathsf{Tainted})^* \\
\overset{🔒}{\Longleftrightarrow} (\texttt{TIME}\cdot\texttt{BUSY}\cdot\texttt{NEWLINE})\cdot((\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{NEWLINE})\colon \mathsf{Tainted})^*
\end{array}$$

Then, by the typing rule for iteration, as Trusted observes the unambiguous iterability of the types in the view, we have:

$$\begin{array}{l}
\texttt{(private . public*)*} \in \\
\quad ((\texttt{ASTERISK}\cdot\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{LOCATION}\cdot\texttt{NEWLINE})\cdot \\
\quad ((\texttt{SPACE}\cdot\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{LOCATION}\cdot\texttt{NEWLINE})\colon \mathsf{Tainted})^*)^* \\
\overset{🔒}{\Longleftrightarrow} ((\texttt{TIME}\cdot\texttt{BUSY}\cdot\texttt{NEWLINE})\cdot((\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{NEWLINE})\colon \mathsf{Tainted})^*)^*
\end{array}$$

Finally, by the typing rule for concatenation, and again as Trusted observes the unambiguous concatenability of the types in the view, we have:

$$\begin{array}{l}
\texttt{public* . (private . public*)*} \in \\
\quad (((\texttt{SPACE}\cdot\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{LOCATION}\cdot\texttt{NEWLINE})^*)\colon \mathsf{Tainted})^*\cdot \\
\quad (\texttt{ASTERISK}\cdot\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{LOCATION}\cdot\texttt{NEWLINE})\cdot \\
\quad (((\texttt{SPACE}\cdot\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{LOCATION}\cdot\texttt{NEWLINE})^*)\colon \mathsf{Tainted})^* \\
\overset{🔒}{\Longleftrightarrow} (((\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{NEWLINE})^*)\colon \mathsf{Tainted})^*\cdot \\
\quad (\texttt{TIME}\cdot\texttt{BUSY}\cdot\texttt{NEWLINE})\cdot \\
\quad (((\texttt{TIME}\cdot\texttt{DESC}\cdot\texttt{NEWLINE})^*)\colon \mathsf{Tainted})^*
\end{array}$$

The equivalent type stated in the lemma can be obtained using the equivalence between

$$(R\colon \mathsf{Tainted})^*\cdot(S\cdot(R\colon \mathsf{Tainted})^*)^*$$

and

$$((R\colon \mathsf{Tainted}) \mid S)^*$$

which holds when R and S are disjoint and unambiguously iterable. □

5.2.2 Lemma: Secure lenses admit the following inference rule:

$$\frac{v' \approx_k l.get\ s \approx_k v}{l.put\ v'\ (l.put\ v\ s) \approx_k l.put\ v'\ s} \qquad \text{(PutPutTrusted)}$$

Proof: Let $k \in \mathcal{Q}$ be a label, and let $v, v' \in V$ and $s \in S$ be strings such that $v \approx_k (l.get\ s)$ and $v' \approx_k (l.get\ s)$. By PutGet for l and the reflexivity of $\approx_k$ we have:

$$l.get\ (l.put\ v\ s) \approx_k v \approx_k v'$$

Using this fact, we calculate as follows:

$$\begin{array}{lll}
& l.put\ v'\ (l.put\ v\ s) & \\
\approx_k & l.put\ v\ s & \text{by GetPut for } l \\
\approx_k & s & \text{by GetPut for } l
\end{array}$$

Also by GetPut for l we have

$$l.put\ v'\ s \approx_k s$$

The required equivalence follows from the transitivity of $\approx_k$. □

$$\boxed{\frac{E \text{ well-formed}}{\mathit{copy}\ E \in E \overset{\text{🔒}}{\Longleftrightarrow} E}}$$

5.4.1 Lemma: Let $E \in \mathcal{R}$ be a well-formed security-annotated regular expression. Then *copy* E is a secure lens in $E \overset{\text{🔒}}{\Longleftrightarrow} E$.

Proof: We prove the secure lens laws separately. In this chapter, we will omit the proofs of PutGet and CreateGet because they are identical to the basic lens proofs.
▶ **GetPut:** Let k be a label in $\mathcal{Q}$ and let e and e' be strings in E such that $e' \approx_k (\mathit{copy}\ E).\mathit{get}\ e$. We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{copy}\ E).\mathit{put}\ e'\ e & \\
= e' & \text{by definition of } (\mathit{copy}\ E).\mathit{put} \\
\approx_k (\mathit{copy}\ E).\mathit{get}\ e & \text{by assumption} \\
= e & \text{by definition of } (\mathit{copy}\ E).\mathit{get}
\end{array}$$

and obtain the required equivalence.
▶ **GetNoLeak:** Let j be a label in $\mathcal{P}$ and let e and e' be strings in E such that $e' \sim_j e$. We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{copy}\ E).\mathit{get}\ e & \\
= e & \text{by definition of } (\mathit{copy}\ E).\mathit{get} \\
\sim_j e' & \text{by assumption} \\
= (\mathit{copy}\ E).\mathit{get}\ e' & \text{by definition of } (\mathit{copy}\ E).\mathit{get}
\end{array}$$

and obtain the required equivalence. □

$$\boxed{\frac{E, F \text{ well-formed} \qquad F = \{u\}}{\mathit{const}\ E\ F \in E \overset{\text{🔒}}{\Longleftrightarrow} F}}$$

5.4.2 Lemma: Let E and F be well-formed security-annotated regular expressions such that $F = \{u\}$ for some string u. Then *const* E F is a secure lens in $E \overset{\text{🔒}}{\Longleftrightarrow} F$.

Proof: We prove each secure lens law separately.
▶ **GetPut:** Let k be a label in $\mathcal{Q}$, let e be a string in E, and let u be a string in F such that:

$$u \approx_k (\mathit{const}\ E\ F).\mathit{get}\ e$$

We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{const}\ E\ F).\mathit{put}\ u\ e & \\
= e & \text{by definition } (\mathit{const}\ E\ F).\mathit{put} \\
\approx_k e & \text{by reflexivity}
\end{array}$$

and obtain the required equivalence.
▶ **GetNoLeak:** Let j be a label in $\mathcal{P}$ and let e and e' be strings in E such that $e \sim_j e'$. Let u be the unique element of F.

$$\begin{array}{ll}
\quad (\mathit{const}\ E\ F).\mathit{get}\ e & \\
= u & \text{by definition } (\mathit{const}\ E\ F).\mathit{get} \\
= (\mathit{const}\ E\ F).\mathit{get}\ e' & \text{by definition } (\mathit{const}\ E\ F).\mathit{get}
\end{array}$$

and obtain the required equivalence by the reflexivity of $\sim_j$. □

$$\frac{\begin{array}{c} S_1 \cap S_2 = \emptyset \\ l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1 \\ l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2 \\ q = \bigvee\{k \mid k \text{ min obs. } V_1 \neq V_2 \wedge V_1 \;\&\; V_2 \text{ agree}\} \\ p = \bigvee\{k \mid k \text{ min obs. } S_1 \cap S_2 = \emptyset\} \end{array}}{l_1 \,|\, l_2 \in (S_1 \,|\, S_2){:}\,q \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \,|\, V_2){:}\,p}$$

5.4.3 Lemma: Let $l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2$ be secure lenses such that $S_1 \cap S_2 = \emptyset$. Then $l_1 \,|\, l_2$ is a secure lens in $(S_1 \,|\, S_2){:}\,q \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \,|\, V_2){:}\,p$ where the label q is $\bigvee\{k \mid k \text{ min obs. } V_1 \neq V_2 \wedge V_1 \;\&\; V_2 \text{ agree}\}$ and the label p is $\bigvee\{k \mid k \text{ min obs. } S_1 \cap S_2 = \emptyset\}$.

Proof: We prove each secure lens law separately.
▶ **GetPut:** Let k be a label in $\mathcal{Q}$, let s be a string in $(S_1 \,|\, S_2){:}\,q$, and let v be a string in $(V_1 \,|\, V_2){:}\,p$ such that $v \approx_k \textit{get } s$. We analyze two cases.

Case $k \not\sqsupseteq q$: Then the equivalence $\approx_k$ is the total relation on $(S_1 \,|\, S_2){:}\,q$ and $(l_1 \,|\, l_2).\textit{put } v\ s \approx_k s$ trivially.

Case $k \sqsupseteq q$ and $v \in V_1$ and $s \in S_1$: From

$$v \in V_1 \qquad v \approx_k^{(V_1 \,|\, V_2):p} (l_1 \,|\, l_2).\textit{get } s \qquad k \text{ observes } V_1 \;\&\; V_2 \text{ agree}$$

we have $v \approx_k^{V_1} l_1.\textit{get } s$. Using this fact, we calculate as follows

$$\begin{array}{rll} & (l_1 \,|\, l_2).\textit{put } v\ s & \\ = & l_1.\textit{put } v\ s & \text{by definition of } (l_1 \,|\, l_2).\textit{get} \\ \approx_k & s & \text{by GetPut for } l_1 \end{array}$$

and obtain the required equivalence.

Case $k \sqsupseteq q$ and $v \in V_2$ and $s \in S_2$: Symmetric to the previous case.

Case $k \sqsupseteq q$ and $v \in V_2 - V_1$ and $s \in S_1$: Can't happen. The assumptions k observes $V_1 \neq V_2$ and $v \approx_k (l_1 \,|\, l_2).\textit{get } s$ lead to a contradiction.

Case $k \sqsupseteq q$ and $v \in V_1 - V_2$ and $s \in S_2$: Symmetric to the previous case.

▶ **GetNoLeak:** Let j be a label in $\mathcal{P}$ and let s and s' be strings in $(S_1 \,|\, S_2) : q$ with $s \sim_j s'$. We analyze two cases.

Case $j \not\sqsupseteq p$: Then the equivalence $\sim_j$ is the total relation on $(V_1 \,|\, V_2){:}\,p$ and $(l_1 \,|\, l_2).\textit{get } s \approx_j (l_1 \,|\, l_2).\textit{get } s'$ trivially.

Case $j \sqsupseteq p$ and $s \in S_1$: From

$$s \in S_1 \qquad s{\sim_j}s' \qquad k \text{ observes } S_1 \cap S_2 = \emptyset$$

we have $s' \in S_1$ and $s \sim_k^{S_1} s'$. Using these facts, we calculate as follows

$$\begin{array}{rll} & (l_1 \,|\, l_2).\textit{get } s & \\ = & l_1.\textit{get } s & \text{by definition of } (l_1 \,|\, l_2).\textit{get} \\ \sim_k & l_1.\textit{get } s' & \text{by GetNoLeak for } l_1 \\ = & (l_1 \,|\, l_2).\textit{get } s' & \text{by definition of } (l_1 \,|\, l_2).\textit{get} \end{array}$$

and obtain the required equivalence.

Case $j \sqsupseteq p$ and $s \in s_2$: Symmetric to the previous case. □

$$\boxed{\frac{\begin{array}{c} l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1 \qquad S_1 \cdot^{!} S_2 \\ l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2 \qquad V_1 \cdot^{!} V_2 \\ q = \bigvee\{k \mid k \text{ min obs. } V_1 \cdot^{!} V_2\} \\ p = \bigvee\{k \mid k \text{ min obs. } S_1 \cdot^{!} S_2\} \end{array}}{l_1 \cdot l_2 \in (S_1 \cdot S_2) \colon q \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \cdot V_2) \colon p}}$$

5.4.4 Lemma: Let $l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2$ be secure lenses such that $S_1 \cdot^{!} S_2$ and $V_1 \cdot^{!} V_2$. Then $(l_1 \cdot l_2)$ is a secure lens in $(S_1 \cdot S_2) \colon q \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \cdot V_2) \colon p$ where the label q is $\bigvee\{k \mid k \text{ min obs. } V_1 \cdot^{!} V_2\}$ and the label p is $\bigvee\{k \mid k \text{ min obs. } S_1 \cdot^{!} S_2\}$.

Proof: We prove each secure lens law separately.
▶ **GetPut**: Let k be a label in $\mathcal{Q}$, let $s_1 \cdot s_2$ be a string in $(S_1 \cdot S_2) \colon q$, and let $v_1 \cdot v_2$ be a string in $(V_1 \cdot V_2) \colon p$ such that $(v_1 \cdot v_2) \approx_k (l_1 \cdot l_2).\mathit{get}\ (s_1 \cdot s_2)$. We analyze two cases.

Case $k \not\sqsupseteq q$: Then $\approx_k$ is the total relation on $(S_1 \cdot S_2) \colon q$ and so

$$(l_1 \cdot l_2).\mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) \approx_k (s_1 \cdot s_2)$$

trivially.

Case $k \sqsupseteq q$: From k observes $V_1 \cdot^{!} V_2$ we also have:

$$v_1 \approx_k^{V_1} l_1.\mathit{get}\ s_1 \qquad v_2 \approx_k^{V_2} l_2.\mathit{get}\ s_2$$

Using these equivalences, we calculate as follows

$$\begin{array}{rll} & (l_1 \cdot l_2).\mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) & \\ = & (l_1.\mathit{put}\ v_1\ s_1) \cdot (l_2.\mathit{put}\ v_2\ s_2) & \text{by definition } (l_1 \cdot l_2).\mathit{put} \\ \approx_k & s_1 \cdot s_2 & \text{by GETPUT for } l_1 \text{ and } l_2 \end{array}$$

and obtain the required equivalence.

▶ **GetNoLeak**: Let j be a label in $\mathcal{P}$ and let $s_1 \cdot s_2$ and $s_1' \cdot s_2'$ be strings in $(S_1 \cdot S_2) \colon q$ such that:

$$s_1 \cdot s_2 \sim_j s_1' \cdot s_2'$$

We analyze two cases.

Case $j \not\sqsupseteq p$: Then $\sim_j$ is the total relation on $(V_1 \cdot V_2) \colon p$ and so

$$(l_1 \cdot l_2).\mathit{get}\ (s_1 \cdot s_2) \sim_j (l_1 \cdot l_2).\mathit{get}\ (s_1' \cdot s_2')$$

trivially.

Case $j \sqsupseteq p$: From j observes $S_1 \cdot^{!} S_2$ we also have:

$$s_1 \sim_j^{S_1} s_1' \qquad s_2 \sim_j^{S_2} s_2'$$

Using these equivalences, we calculate as follows

$$\begin{array}{rll} & (l_1 \cdot l_2).\mathit{get}\ (s_1 \cdot s_2) & \\ = & (l_1.\mathit{get}\ s_1) \cdot (l_2.\mathit{get}\ s_2) & \text{by definition of } (l_1 \cdot l_2).\mathit{get} \\ \sim_j & (l_1.\mathit{get}\ s'_1) \cdot (l_2.\mathit{get}\ s'_2) & \text{by GetNoLeak for } l_1 \text{ and } l_2 \\ = & (l_1 \cdot l_2).\mathit{get}\ (s'_1 \cdot s'_2) & \text{by definition of } (l_1 \cdot l_2).\mathit{get} \end{array}$$

and obtain the required equivalence. □

$$\frac{\begin{array}{c} S^{!*} \qquad V^{!*} \\ l \in S \stackrel{\blacksquare}{\Longleftrightarrow} V \\ q = \bigvee \{k \mid k \text{ min obs. } V^{!*}\} \\ p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\} \end{array}}{l^* \in (S^*){:}\,q \stackrel{\blacksquare}{\Longleftrightarrow} (V^*){:}\,p}$$

5.4.5 Lemma: Let $l \in S \stackrel{\blacksquare}{\Longleftrightarrow} V$ be a secure lens such that $S^{!*}$ and $V^{!*}$. Then l^* is a secure lens in $(S^*){:}\,q \stackrel{\blacksquare}{\Longleftrightarrow} (V^*){:}\,p$ where $q = \bigvee \{k \mid k \text{ min obs. } V^{!*}\}$ and $p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\}$.

Proof: We prove each secure lens law separately.
▶ **GetPut**: Let k be a label in $\mathcal{Q}$, let $s_1 \cdots s_m$ be a string in $(S^*){:}\,q$, and let $v_1 \cdots v_n$ be a string in $(V^*){:}\,p$ such that $(v_1 \cdots v_n) \approx_k l^*.\mathit{get}\ (s_1 \cdots s_m)$. We analyze two cases:

Case $k \not\sqsupseteq q$: Then $\approx_k$ is the total relation on $(S^*){:}\,q$ and so

$$l^*.\mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) \approx_k (s_1 \cdots s_m)$$

trivially.

Case $k \sqsupseteq q$: From

$$v_1 \cdots v_n \approx_k l^*.\mathit{get}\ (s_1 \cdots s_m) \qquad k \text{ observes } V^{!*}$$

we have:

$$m = n \qquad v_i \approx_k l.\mathit{get}\ s_i \text{ for } i \in \{1, \ldots, n\}$$

Using these facts, we calculate as follows

$$\begin{array}{rll} & l^*.\mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_n) & \\ = & (l.\mathit{put}\ v_1\ s_1) \cdots (l.\mathit{put}\ v_n\ s_n) & \text{by definition } l^*.\mathit{put} \\ \approx_k & s_1 \cdots s_n & \text{by GetPut for } l \end{array}$$

and obtain the required equivalence.

▶ **GetNoLeak**: Let j be a label in $\mathcal{P}$ and let $s_1 \cdots s_m$ and $s'_1 \cdots s'_n$ be strings in $(S^*) : q$ such that $(s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)$. We analyze two cases.

Case $j \not\sqsupseteq p$: Then $\sim_j$ is the total relation on $(V^*){:}\,p$ and so

$$l^*.\mathit{get}\ (s_1 \cdots s_n) \sim_j l^*.\mathit{get}\ (s'_1 \cdots c'_n)$$

trivially.

Case $j \sqsupseteq p$: From

$$(s_1 \cdots s_m) {\sim_j} (s'_1 \cdots s'_n) \qquad j \text{ observes } S^{!*}$$

we have

$$m = n \qquad s_i \sim^S_j s'_i \text{ for } i \in \{1, \ldots, n\}$$

Using these facts, calculate as follows

$$\begin{array}{rll} & l^*.\mathit{get}\ (s_1 \cdots s_n) & \\ = & (l.\mathit{get}\ s_1) \cdots (l.\mathit{get}\ s_n) & \text{by definition } l^*.\mathit{get} \\ \sim_j & (l.\mathit{get}\ s'_1) \cdots (l.\mathit{get}\ s'_n) & \text{by G{\small ET}N{\small O}L{\small EAK} for } l \\ = & l^*.\mathit{get}\ (s'_1 \cdots s'_n) & \text{by definition } l^*.\mathit{get} \end{array}$$

and obtain the required equivalence. □

$$\boxed{\dfrac{\begin{array}{c} l_1 \in S \overset{🔒}{\Longleftrightarrow} U \\ l_2 \in U \overset{🔒}{\Longleftrightarrow} V \end{array}}{l_1;l_2 \in S \overset{🔒}{\Longleftrightarrow} V}}$$

5.4.6 Lemma: Let $l_1 \in S \overset{🔒}{\Longleftrightarrow} U$ and $l_2 \in U \overset{🔒}{\Longleftrightarrow} V$ be secure lenses. Then $l_1;l_2$ is a secure lens in $S \overset{🔒}{\Longleftrightarrow} V$.

Proof: We prove each secure lens law separately.

▶ **GetPut**: Let k be a label in $\mathcal{Q}$, let s be a string in S, and let v be a string in V such that $v \approx_k (l_1;l_2).\mathit{get}\ s$. By the definition of $(l_1;l_2).\mathit{get}$ we have:

$$v \approx_k l_2.\mathit{get}\ (l_1.\mathit{get}\ s)$$

By the GetPut law for l_2 we also have

$$l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s) \approx_k l_1.\mathit{get}\ s$$

Using this fact and the GetPut law for l_1 we obtain

$$(l_1;l_2).\mathit{put}\ v\ s = l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s \approx_k s$$

as required.

▶ **GetNoLeak**: Let j be a label in $\mathcal{P}$, and let s and s' strings in S such that $s \sim_j s'$. We calculate as follows:

$$\begin{array}{rll} & (l_1;l_2).\mathit{get}\ s & \\ = & l_2.\mathit{get}\ (l_1.\mathit{get}\ s) & \text{by definition of } (l_1;l_2).\mathit{get} \\ \sim_j & l_2.\mathit{get}\ (l_1.\mathit{get}\ s') & \text{by G{\small ET}N{\small O}L{\small EAK} for } l_1 \text{ and } l_2 \\ = & (l_1;l_2).\mathit{get}\ s' & \text{by definition of } (l_1;l_2).\mathit{get} \end{array}$$

and obtain the required equivalence. □

$$\boxed{\dfrac{\begin{array}{c} E, F \text{ well-formed} \qquad E \cap F = \emptyset \qquad (E \mid F)^{!*} \\ q = \bigvee \{k \mid k \text{ min obs. } E^{!*}\} \\ p \sqsupseteq \bigvee \{k \mid k \text{ observes } E {\cdot^!} F \text{ and } F {\cdot^!} E\} \end{array}}{\mathit{filter}\ E\ F \in (E{:}q \mid F{:}p)^* \overset{🔒}{\Longleftrightarrow} E^*}}$$

5.4.7 Lemma: Let E and F be well-formed security-annotated regular expressions such that $E \cap F = \emptyset$ and $(E \mid F)^{!*}$. Then for every label p such that $p \sqsupseteq \bigvee\{k \mid k \text{ observes } E \cdot^{!} F \text{ and } F \cdot^{!} E\}$ the secure lens *filter* E F is in $(E{:}\,q \mid F{:}\,p)^* \overset{\text{🔒}}{\Longleftrightarrow} E^*$ where $q = \bigvee \{k \mid k \text{ min obs. } E^{!*}\}$.

Proof: We prove each secure lens law separately. To shorten the proof, we will abbreviate $filter\ E\ F$ as l.

▶ **GetPut:** Let k be a label in $\mathcal{Q}$, let $s_1 \cdots s_m$ be a string in $(E{:}\,q \mid F{:}\,p)^*$, and let $v_1 \cdots v_n$ be a string in E^* such that

$$v_1 \cdots v_n \approx_k (l.\mathit{get}\ (s_1 \cdots s_m))$$

We consider several cases.

Case $k \not\sqsupseteq q$**:** We will prove that for all $v_1 \cdots v_n$ in E^* we have

$$l.\mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) \approx_k (s_1 \cdots s_m)$$

by induction on n. Note that we use a strengthened induction hypothesis that does not assume

$$v_1 \cdots v_n \approx_k l.\mathit{get}\ (s_1 \cdots s_m).$$

We analyze several subcases:

Subcase $m = 0$**:** We calculate as follows:

$$\begin{array}{ll} \quad l.\mathit{put}\ (v_1 \cdots v_n)\ \epsilon & \\ = \mathsf{str\_unfilter}\ F\ (v_1 \cdots v_n)\ \epsilon & \text{by definition } l.\mathit{put} \\ = v_1 \cdots v_n & \text{by definition } \mathsf{str\_unfilter} \end{array}$$

As $k \not\sqsupseteq q$ we have $hide_k(v_1 \cdots v_n) = hide_k(\epsilon)$, as required.

Subcase $m > 0$ **and** $s_1 \in F$**:** We calculate as follows:

$$\begin{array}{ll} \quad l.\mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) & \\ = \mathsf{str\_unfilter}\ F\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) & \text{by definition } l.\mathit{put} \\ = s_1 \cdot (\mathsf{str\_unfilter}\, F (v_1 \cdots v_n)(s_2 \cdots s_m)) & \text{by definition } \mathsf{str\_unfilter} \\ \qquad \text{with } s_1 \in F & \\ = s_1 \cdot (l.\mathit{put}\ (v_1 \cdots v_n)\ (s_2 \cdots s_m)) & \text{by definition } l.\mathit{put} \end{array}$$

The required equivalence follows by the induction hypothesis and the definition of $\approx_k$ with $k \not\sqsupseteq q$.

Subcase $m > 0$ **and** $s_1 \in E$**:** Similar to the previous subcase.

Case $k \sqsupseteq q$**:** Similar to the previous case.

▶ **GetNoLeak:** Let j be a label in $\mathcal{P}$ and let $s_1 \cdots s_m$ and $s'_1, \cdots s'_n$ be strings in $(E{:}\,q \mid F{:}\,p)^*$ such that $(s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)$. We analyze two cases

Case $j \not\sqsupseteq p$**:** Let $[e_1, \ldots, e_i]$ and $[e'_1, \ldots, e'_j]$ be the sequences of substrings of $s_1 \cdots s_m$ and $s_1 \cdots s'_n$ that belong to E. As $hide_j$ function maps each string in F to ϵ we have:

$$hide_j(e_1 \cdots e_i) = hide_j(e'_1 \cdots e'_j)$$

We calculate as follows

$$\begin{aligned}
& \mathit{hide}_j(e_1 \cdots e_i) = \mathit{hide}_j(e'_1 \cdots e'_j) \\
i.e., \ & \mathit{hide}_j(\mathsf{str\_filter}\ E\ (s_1 \cdots s_m)) = \mathit{hide}_j(\mathsf{str\_filter}\ E\ (s'_1 \cdots s'_n)) \\
& \quad \text{by definition of } \mathsf{str\_filter} \\
i.e., \ & \mathit{hide}_j(l.\mathit{get}\ (s_1 \cdots s_m)) = \mathit{hide}_j(l.\mathit{get}\ (s'_1 \cdots s'_n)) \\
& \quad \text{by definition } l.\mathit{get} \\
i.e., \ & l.\mathit{get}\ (s_1 \cdots s_m) \sim_j l.\mathit{get}\ (s'_1 \cdots s'_n) \\
& \quad \text{by definition of } \sim_j
\end{aligned}$$

and obtain the required equivalence.

Case $j \sqsupseteq p$**:** Let $[\vec{e_1}, \ldots, \vec{e_o}]$ and $[\vec{e_1}', \ldots, \vec{e_p}']$ be the sequences of *contiguous* elements of E in $s_1 \cdots s_m$ and $s'_1 \cdots s'_n$—i.e, elements not separated by an F. As j observes $E \cdot^! F$ and $F \cdot^! E$ we have $o = p$ and $\mathit{hide}_j(\vec{e_i}) = \mathit{hide}_j(\vec{e_i}')$ for i from 1 to o. We calculate as follows

$$\begin{aligned}
& \mathit{hide}_j(\vec{e_1}) \cdots \mathit{hide}_j(\vec{e_o}) = \mathit{hide}_j(\vec{e_1}') \cdots \mathit{hide}_j(\vec{e_p}') \\
i.e., \ & \mathit{hide}_j(\mathsf{str\_filter}\ E\ (s_1 \cdots s_m)) = \mathit{hide}_j(\mathsf{str\_filter}\ E\ (s'_1 \cdots s'_n)) \\
& \quad \text{by definition of } \mathsf{str\_filter} \\
i.e., \ & \mathit{hide}_j(l.\mathit{get}\ (s_1 \cdots s_m)) = \mathit{hide}_j(l.\mathit{get}\ (s'_1 \cdots s'_n)) \\
& \quad \text{by definition of } l.\mathit{get} \\
i.e., \ & (l.\mathit{get}\ (s_1 \cdots s_m)) \sim_j (l.\mathit{get}\ (s'_1 \cdots s'_n)) \\
& \quad \text{by definition of } \sim_j
\end{aligned}$$

and obtain the required equivalence. □

$$\boxed{\dfrac{q \in \mathcal{Q} \qquad p \in \mathcal{P} \\ l \in S \overset{\text{🔒}}{\Longleftrightarrow} V}{l \in S{:}q \overset{\text{🔒}}{\Longleftrightarrow} V{:}p}}$$

5.4.8 Lemma: Let $l \in S \overset{\text{🔒}}{\Longleftrightarrow} V$ be a secure lens and let q be a label in $\mathcal{Q}$ and p a label in $\mathcal{P}$ be labels. Then l is also a secure lens in $S{:}q \overset{\text{🔒}}{\Longleftrightarrow} V{:}p$.

Proof: We prove each secure lens law separately.
▶ **GetPut**: Let k be a label in $\mathcal{Q}$, let s be a string in $S{:}q$, and let v be a string in $V{:}p$ with $v \approx_k l.\mathit{get}\ s$. We analyze two cases:

Case $k \not\sqsupseteq q$**:** Then $\approx_k$ is the total relation on $S{:}q$ and $l.\mathit{put}\ v\ s \approx_k s$ trivially.

Case $k \sqsupseteq q$**:** By the definition of $\approx_k^{(V:p)}$ we have $v \approx_k^V l.\mathit{get}\ s$. By GetPut for l we have $l.\mathit{put}\ v\ s \approx_k^S s$. Finally, by the definition of $\approx_k^{(S:q)}$ we have $l.\mathit{put}\ v\ s \approx_k^{(S:q)} s$, as required.

▶ **GetNoLeak**: Let j be a label in $\mathcal{P}$ and let s and s' be strings in $S{:}\,q$ with $s \sim_k s'$. We analyze two cases:

Case $j \not\sqsupseteq p$**:** Then $\sim_j$ is the total relation on $V{:}p$ and $l.\mathit{get}\ s \approx_j l.\mathit{get}\ s'$ trivially.

Case $j \sqsupseteq p$**:** By the definition of $\sim_j^{(S:q)}$ we have $s \sim_j^S s'$. By GetNoLeak for l we have $l.\mathit{get}\ s \sim_k^V l.\mathit{get}\ s'$. Finally, by the definition of $\sim_j^{(V:q)}$ we have $l.\mathit{get}\ s \sim_j^{(V:q)} l.\mathit{get}\ s'$ as required. □

$$\boxed{\dfrac{E \text{ well-formed} \qquad \forall\, (j,k) \in \mathcal{C}.\ \sim_j \subseteq \approx_k}{\mathit{copy}\ E \in E \overset{🔒}{\Longleftrightarrow} E}}$$

5.5.1 Lemma: Let E be a well-formed security-annotated regular expression such that for every clearance (j,k) in $\mathcal{C}$ we have $\sim_j \subseteq \approx_k$. Then *copy* E is a dynamic secure lens in $E \overset{🔒}{\Longleftrightarrow} E$.

Proof: We prove each dynamic secure lens law separately.
▶ **GetPut:** Let (j,k) be a clearance in $\mathcal{C}$, let s be a string in E, and let v be a string in E with $(\mathit{copy}\ E).\mathit{safe}\ (j,k)\ v\ s$. We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{copy}\ E).\mathit{put}\ v\ s & \\
= v & \text{by definition } (\mathit{copy}\ E).\mathit{put} \\
\approx_k s & \text{by definition } (\mathit{copy}\ E).\mathit{safe}
\end{array}$$

and obtain the required equivalence.
▶ **PutNoLeak:** Let (j,k) be a clearance in $\mathcal{C}$, let s, s', v and v' be strings in E such that:

$$\begin{array}{ll}
s \sim_j s' & (\mathit{copy}\ E).\mathit{safe}\ (j,k)\ v\ s \\
v \sim_j v' & (\mathit{copy}\ E).\mathit{safe}\ (j,k)\ v'\ s'
\end{array}$$

We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{copy}\ E).\mathit{put}\ v\ s & \\
= v & \text{by definition of } (\mathit{copy}\ E).\mathit{put} \\
\sim_j v' & \text{by assumption} \\
= (\mathit{copy}\ E).\mathit{put}\ v'\ s' & \text{by definition of } (\mathit{copy}\ E).\mathit{put}
\end{array}$$

and obtain the required equivalence.
▶ **SafeNoLeak:** Let (j,k) be a clearance in $\mathcal{C}$, and let s, s', v and v' be strings in E such that $v \sim_j v'$ and $s \sim_j s'$. Then as $\sim_j \subseteq \approx_k$ we have $s \approx_k s'$ and $v \approx_k v'$. Using these equivalences, we calculate as follows

$$\begin{array}{ll}
\quad (\mathit{copy}\ E).\mathit{safe}\ (j,k)\ v\ s & \\
= v \approx_k s & \text{by definition of } (\mathit{copy}\ E).\mathit{safe} \\
= v' \approx_k s' & \text{by symmetry and transitivity of } \approx_k \\
= (\mathit{copy}\ E).\mathit{safe}\ (j,k)\ v'\ s' & \text{by definition of } (\mathit{copy}\ E).\mathit{safe}
\end{array}$$

and obtain the required equality. □

$$\boxed{\dfrac{E, F \text{ well-formed} \qquad F = \{u\}}{\mathit{const}\ E\ F\ d \in E \overset{🔒}{\Longleftrightarrow} F}}$$

5.5.2 Lemma: Let E and F be well-formed security-annotated regular expressions such that $F = \{u\}$ for some string u. Then *const* E F is a dynamic secure lens in $E \overset{🔒}{\Longleftrightarrow} F$.

Proof: We prove each dynamic secure lens law separately.
▶ **GetPut:** Let (j,k) be a clearance in $\mathcal{C}$, let s be a string in E, and let v be a string in F with $(\mathit{const}\ E\ F).\mathit{safe}\ (j,k)\ v\ s$. We calculate as follows

$$\begin{array}{ll}
\quad (\mathit{const}\ E\ F).\mathit{put}\ u\ s & \\
= s & \text{by definition } (\mathit{const}\ E\ F).\mathit{put} \\
\approx_k s & \text{by reflexivity of } \approx_k
\end{array}$$

and obtain the required equivalence.

▶ **PutNoLeak:** Let (j,k) be a clearance in $\mathcal{C}$, let s and s' be strings in E, and let v and v' be strings in F such that:

$$\begin{array}{ll} s \sim_j s' & (\mathit{const}\ E\ F).\mathit{safe}\ (j,k)\ v\ s \\ v \sim_j v' & (\mathit{const}\ E\ F).\mathit{safe}\ (j,k)\ v'\ s' \end{array}$$

We calculate as follows

$$\begin{array}{rll} & (\mathit{const}\ E\ F).\mathit{put}\ v\ s & \\ = & s & \text{by definition of } (\mathit{const}\ E\ F).\mathit{put} \\ \sim_j & s' & \text{by assumption} \\ = & (\mathit{const}\ E\ F).\mathit{put}\ v'\ s' & \text{by definition of } (\mathit{const}\ E\ F).\mathit{put} \end{array}$$

and obtain the required equivalence.

▶ **SafeNoLeak:** Let s and s' be strings in E and let v and v' be strings in F such that $s \sim_j s'$ and $v \sim_j v'$. By the definition of $(\mathit{const}\ E\ F).\mathit{safe}$ we immediately have

$$(\mathit{const}\ E\ F).\mathit{safe}\ (j,k)\ v\ s = \top = (\mathit{const}\ E\ F).\mathit{safe}\ (j,k)\ v'\ s',$$

as required. □

$$\boxed{\dfrac{\begin{array}{cc} l_1 \in S_1 \overset{🔒}{\Longleftrightarrow} V_1 & S_1{\cdot^{!}}S_2 \\ l_2 \in S_2 \overset{🔒}{\Longleftrightarrow} V_2 & V_1{\cdot^{!}}S_2 \\ \multicolumn{2}{c}{p = \bigvee\{k \mid k \text{ min obs. } S_1{\cdot^{!}}S_2\}} \end{array}}{l_1{\cdot}l_2 \in (S_1{\cdot}S_2) \overset{🔒}{\Longleftrightarrow} (V_1{\cdot}V_2){:}p}}$$

5.5.3 Lemma: Let $l_1 \in S_1 \overset{🔒}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{🔒}{\Longleftrightarrow} V_2$ be dynamic secure lenses such that $S_1{\cdot^{!}}S_2$ and $V_1{\cdot^{!}}V_2$. Then $l_1{\cdot}l_2$ is a dynamic secure lens in $(S_1{\cdot}S_2) \overset{🔒}{\Longleftrightarrow} (V_1{\cdot}V_2) : p$ where the label p is $\bigvee\{k \mid k \text{ min obs. } S_1{\cdot^{!}}S_2\}$.

Proof: We prove each dynamic secure lens law separately.

▶ **GetPut:** Let (j,k) be a clearance in $\mathcal{C}$, let $v_1{\cdot}v_2$ be a string in $(V_1{\cdot}V_2){:}p$, and let $s_1{\cdot}s_2$ be a string in $S_1{\cdot}S_2$ such that $(l_1{\cdot}l_2).\mathit{safe}\ (j,k)\ (v_1{\cdot}v_2)\ (s_1{\cdot}s_2)$. By the definition of $(l_1{\cdot}l_2).\mathit{safe}$ we have:

$$l_1.\mathit{safe}\ (j,k)\ v_1\ s_1 \qquad l_2.\mathit{safe}\ (j,k)\ v_2\ s_2$$

Using these facts, we calculate as follows

$$\begin{array}{rll} & (l_1{\cdot}l_2).\mathit{put}\ (v_1{\cdot}v_2)\ (s_1{\cdot}s_2) & \\ = & (l_1.\mathit{put}\ v_1\ s_1){\cdot}(l_2.\mathit{put}\ v_2\ s_2) & \text{by definition of } (l_1{\cdot}l_2).\mathit{put} \\ \approx_k & s_1{\cdot}s_2 & \text{by } \textsc{GetPut} \text{ for } l_1 \text{ and } l_2 \end{array}$$

and obtain the required equivalence.

▶ **PutNoLeak:** Let (j,k) be a clearance in $\mathcal{C}$, let $s_1{\cdot}s_2$ and $s'_1{\cdot}s'_2$ be strings in $S_1{\cdot}S_2$, and let $v_1{\cdot}v_2$ and $v'_1{\cdot}v'_2$ be strings in $(V_1{\cdot}V_2){:}p$ such that:

$$\begin{array}{ll} (s_1{\cdot}s_2) \sim_j (s'_1{\cdot}s'_2) & (l_1{\cdot}l_2).\mathit{safe}\ (j,k)\ (v_1{\cdot}v_2)\ (s_1{\cdot}s_2) \\ (v_1{\cdot}v_2) \sim_j (v'_1{\cdot}v'_2) & (l_1{\cdot}l_2).\mathit{safe}\ (j,k)\ (v'_1{\cdot}v'_2)\ (s'_1{\cdot}s'_2) \end{array}$$

By the definition of $(l_1 \cdot l_2).\mathit{safe}$, we have j observes $S_1 \cdot^! S_2$ and $V_1 \cdot^! V_2$ and:

$$\begin{array}{llll} s_1 \sim_j^{S_1} s_1' & s_2 \sim_j^{S_2} s_2' & l_1.\mathit{safe}\ (j,k)\ v_1\ s_1 & l_2.\mathit{safe}\ (j,k)\ v_2\ s_2 \\ v_1 \sim_j^{V_1} v_1' & v_2 \sim_j^{V_2} v_2' & l_1.\mathit{safe}\ (j,k)\ v_1\ s_1' & l_2.\mathit{safe}\ (j,k)\ v_2\ s_2' \end{array}$$

Using all these facts, we calculate as follows

$$\begin{array}{rll} & (l_1 \cdot l_2).\mathit{put}\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) & \\ = & (l_1.\mathit{put}\ v_1\ s_1) \cdot (l_2.\mathit{put}\ v_2\ s_2') & \text{by definition of } (l_1 \cdot l_2).\mathit{put} \\ \sim_j & (l_1.\mathit{put}\ v_1'\ s_1') \cdot (l_2.\mathit{put}\ v_2'\ s_2') & \text{by PutNoLeak for } l_1 \text{ and } l_2 \\ = & (l_1 \cdot l_2).\mathit{put}\ (v_1' \cdot v_2')\ (s_1' \cdot s_2') & \text{by definition of } (l_1 \cdot l_2).\mathit{put} \end{array}$$

and obtain the required equivalence.

▶ **SafeNoLeak**: Let (j,k) be a clearance in $\mathcal{C}$, let $s_1 \cdot s_2$ and $s_1' \cdot s_2'$ be strings in $S_1 \cdot S_2$, and let $v_1 \cdot v_2$ and $v_1' \cdot v_2'$ be strings in $(V_1 \cdot V_2){:}p$ such that:

$$(s_1 \cdot s_2) \sim_j (s_1' \cdot s_2') \qquad (v_1 \cdot v_2) \sim_j (v_1' \cdot v_2')$$

We analyze two cases.

Case j **observes** $S_1 \cdot^! S_2$ **and** $V_1 \cdot^! V_2$**:** From the assumptions of the case and the definition of $\sim_j$ we have:

$$\begin{array}{ll} s_1 \sim_j^{S_1} s_1' & s_2 \sim_j^{S_2} s_2' \\ v_1 \sim_j^{V_1} v_1' & v_2 \sim_j^{V_2} v_2' \end{array}$$

Using these facts, we calculate as follows

$$\begin{array}{rll} & (l_1 \cdot l_2).\mathit{safe}\ (j,k)\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) & \\ = & (l_1.\mathit{safe}\ (j,k)\ v_1\ s_1) \land & \\ & (l_2.\mathit{safe}\ (j,k)\ v_2\ s_2) & \text{by definition of } (l_1 \cdot l_2).\mathit{safe} \\ = & (l_1.\mathit{safe}\ (j,k)\ v_1'\ s_1') \land & \\ & (l_2.\mathit{safe}\ (j,k)\ v_2'\ s_2') & \text{by SafeNoLeak for } l_1 \text{ and } l_2 \\ = & (l_1 \cdot l_2).\mathit{safe}\ (j,k)\ (v_1' \cdot v_2')\ (s_1' \cdot s_2') & \end{array}$$

and obtain the required equality.

Case j **does not observe** $S_1 \cdot^! S_2$ **and** $V_1 \cdot^! V_2$**:** By the definition of $(l_1 \cdot l_2).\mathit{safe}$ we immediately have

$$(l_1 \cdot l_2).\mathit{safe}\ (j,k)\ (v_1 \cdot v_2)\ (s_1 \cdot s_2) = \bot = (l_1 \cdot l_2).\mathit{safe}\ (j,k)\ (v_1' \cdot v_2')\ (s_1' \cdot s_2')$$

as required. □

$$\boxed{\dfrac{\begin{array}{c} S_1 \cap S_2 = \emptyset \\ l_1 \in S_1 \overset{🔒}{\Longleftrightarrow} V_1 \\ l_2 \in S_2 \overset{🔒}{\Longleftrightarrow} V_2 \\ p = \bigvee\{k \mid k \text{ min obs. } S_1 \cap S_2 = \emptyset\} \end{array}}{l_1 \mid l_2 \in (S_1 \mid S_2) \overset{🔒}{\Longleftrightarrow} (V_1 \mid V_2){:}p}}$$

5.5.4 Lemma: Let $l_1 \in S_1 \overset{\text{🔒}}{\Longleftrightarrow} V_1$ and $l_2 \in S_2 \overset{\text{🔒}}{\Longleftrightarrow} V_2$ be dynamic secure lenses such that $(\mathcal{L}(S_1) \cap \mathcal{L}(S_2)) = \emptyset$. Then $l_1 \,|\, l_2$ is a dynamic secure lens in $(S_1 \,|\, S_2) \overset{\text{🔒}}{\Longleftrightarrow} (V_1 \,|\, V_2) : p$ where the label p is $\bigvee\{k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset\}$.

Proof: We prove each dynamic secure lens law separately.

▶ **GetPut:** Let (j,k) be a clearance in $\mathcal{C}$, let s be a string in $S_1 \,|\, S_2$, and let v be a string in $(V_1 \,|\, V_2) \colon p$ such that $(l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v\ s$. We analyze two cases.

Case $s \in S_1$**:** By the definition of $(l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v\ s$ we have $v \in V_1$ and $l_1.\mathit{safe}\ (j,k)\ v\ s$. Using these facts, we calculate as follows

$$\begin{array}{ll} \quad (l_1 \,|\, l_2).\mathit{put}\ v\ s & \\ = l_1.\mathit{put}\ v\ s & \text{by definition of } (l_1 \,|\, l_2).\mathit{put} \\ \quad\ \text{with } v \in V_1 & \\ \approx_k s & \text{by } \textsc{GetPut} \text{ for } l_1 \end{array}$$

and obtain the required equivalence.

Case $s \in S_2$**:** Symmetric to the previous case.

▶ **PutNoLeak:** Let (j,k) be a clearance in $\mathcal{C}$, let s and s' be strings in $S_1 \,|\, S_2$, and let v and v' be strings in $(V_1 \,|\, V_2)\colon p$ such that:

$$\begin{array}{ll} s \sim_j s' & (l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v\ s \\ v \sim_j v' & (l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v'\ s' \end{array}$$

We analyze two cases.

Case $s \in S_1$**:** By the definition of $(l_1 \,|\, l_2).\mathit{safe}$ we have that j observes $S_1 \cap S_2 = \emptyset$ and $V_1\ \&\ V_2$ agree. Thus, from $s \sim_j s'$ we have $s' \in S_1$ and $s \sim_j^{S_1} s'$. Next, from $(l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v\ s$ and $(l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v'\ s'$ we have:

$$\begin{array}{ll} v \in V_1 & l_1.\mathit{safe}\ (j,k)\ v\ s \\ v' \in V_1 & l_1.\mathit{safe}\ (j,k)\ v'\ s' \end{array}$$

Finally, from j observes $V_1\ \&\ V_2$ agree we also have $v \sim_j^{V_1} v'$. Putting all these facts together, we calculate as follows

$$\begin{array}{ll} \quad (l_1 \,|\, l_2).\mathit{put}\ v\ s & \\ = l_1.\mathit{put}\ v\ s & \text{by definition of } (l_1 \,|\, l_2).\mathit{put} \\ \quad\ \text{with } v \in V_1 & \\ \sim_j l_1.\mathit{put}\ v\ s' & \text{by } \textsc{PutNoLeak} \text{ for } l_1 \\ = (l_1 \,|\, l_2).\mathit{put}\ v\ s' & \text{by definition of } (l_1 \,|\, l_2).\mathit{put} \\ \quad\ \text{with } v' \in V_1 & \end{array}$$

and obtain the required equivalence.

Case $s \in S_2$**:** Symmetric to the previous case.

▶ **SafeNoLeak:** Let (j,k) be a clearance in $\mathcal{C}$, let v and v' be strings in $(V_1 \,|\, V_2)\colon p$, and let s and s' be strings in $S_1 \,|\, S_2$ such that $v \sim_j v'$ and $s \sim_j s'$ and $(l_1 \,|\, l_2).\mathit{safe}\ v\ s$ and $(l_1 \,|\, l_2).\mathit{safe}\ v'\ s'$. We analyze several cases.

Case j observes $S_1 \cap S_2 = \emptyset$ and V_1 & V_2 agree and $s \in S_1$:
By the facts of the case we have $s' \in S_1$ and $s \sim_j^{S_1} s'$ and also that $v \in V_1$ if and only if $v' \in V_1$ and $v \sim_j^{V_1} v'$. Using these facts we calculate as follows

$$\begin{array}{ll} \quad (l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v\ s & \\ = v \in V_1 \wedge l_1.\mathit{safe}\ (j,k)\ v\ s & \text{by definition } (l_1 \,|\, l_2).\mathit{safe} \\ = v' \in V_1 \wedge l_1.\mathit{safe}\ (j,k)\ v'\ s' & \text{by } \textsc{SafeNoLeak} \text{ for } l_1 \\ = (l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v'\ s' & \text{by definition } (l_1 \,|\, l_2).\mathit{safe} \end{array}$$

Case j observes $S_1 \cap S_2 = \emptyset$ and V_1 & V_2 agree and $s \in S_2$:
Similar to the previous case.

Case j does not observe $S_1 \cap S_2 = \emptyset$ and V_1 & V_2 agree:
By the definition of $(l_1 \,|\, l_2).\mathit{safe}$ we immediately have

$$(l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v\ s = \bot = (l_1 \,|\, l_2).\mathit{safe}\ (j,k)\ v'\ s',$$

as required. □

$$\boxed{\dfrac{\begin{array}{c} l \in S \overset{🔒}{\Longleftrightarrow} V \\ p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\} \end{array}}{l^* \in S^* \overset{🔒}{\Longleftrightarrow} (V^*){:}\,p}}$$

5.5.5 Lemma: Let $l \in S \overset{🔒}{\Longleftrightarrow} V$ be a dynamic secure lens such that $S^{!*}$ and $V^{!*}$. Then l^* is a dynamic secure lens in $(S^*) \overset{🔒}{\Longleftrightarrow} (V^*){:}\,p$ where $p = \bigvee \{k \mid k \text{ min obs. } S^{!*}\}$.

Proof: We prove each dynamic secure lens law separately.
► **GetPut**: Let (j,k) be a clearance in $\mathcal{C}$, let $v_1 \cdots v_n$ be a string in $(V^*){:}\,p$, and let $s_1 \cdots s_m$ be a string in S^* such that $l^*.\mathit{safe}\ (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)$. By the definition of $l^*.\mathit{safe}$ we have $n = m$ and $l.\mathit{safe}\ (j,k)\ v_i\ s_i$ for i from 1 to n. Using these facts, we calculate as follows

$$\begin{array}{ll} \quad l^*.\mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_m) & \\ = (l.\mathit{put}\ v_1\ s_1) \cdots (l.\mathit{put}\ v_n\ s_n) & \text{by definition } l^*.\mathit{put} \\ \approx_k s_1 \cdots s_m & \text{by } \textsc{GetPut} \text{ for } l \end{array}$$

and obtain the required equivalence.
► **PutNoLeak**: Let (j,k) be a clearance in $\mathcal{C}$, let $v_1 \cdots v_m$ and $v'_1 \cdots v'_n$ be strings in $(V^*){:}\,p$, and let $s_1 \cdots s_o$ and $s'_1 \cdots s'_p$ be strings in S^* such that

$$\begin{array}{ll} (v_1 \cdots v_m) \sim_j (v'_1 \cdots v'_n) & l^*.\mathit{safe}\ (j,k)\ (v_1 \cdots v_m)\ (s_1 \cdots s_o) \\ (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p) & l^*.\mathit{safe}\ (j,k)\ (v'_1 \cdots v'_n)\ (s_1 \cdots s'_p) \end{array}$$

By the definition of $l^*.\mathit{safe}$ we have that j observes $S^{!*}$ and $V^{!*}$ and so $m = n = o = p$ and

$$\begin{array}{ll} v_i \sim_j^V v'_i & l.\mathit{safe}\ (j,k)\ v_i\ s_i \\ s_i \sim_j^S s'_i & l.\mathit{safe}\ (j,k)\ v'_i\ s'_i \end{array}$$

for i from i to n. Using these facts, we calculate as follows

$$\begin{array}{rll} & l^*.\mathit{put}\ (v_1 \cdots v_n)\ (s_1 \cdots s_o) & \\ = & (l.\mathit{put}\ v_1\ s_1) \cdots (l.\mathit{put}\ v_n\ s_n) & \text{by definition of } l^*.\mathit{put} \\ \sim_j & (l.\mathit{put}\ v'_1\ s'_1) \cdots (l.\mathit{put}\ v'_m\ s'_m) & \text{by PUTNOLEAK for } l \\ = & l^*.\mathit{put}\ (v'_1 \cdots v'_m)\ (s'_1 \cdots s'_p) & \text{by definition of } l^*.\mathit{put} \end{array}$$

and obtain the required equivalence.

▶ **SafeNoLeak**: Let (j, k) be a clearance in $\mathcal{C}$, let $v_1 \cdots v_n$ and $v'_1 \cdots v'_m$ be strings in $(V^*)\!: p$, and let $s_1 \cdots s_o$ and $s'_1 \cdots s'_p$ be strings in S^* such that:

$$(v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_m) \qquad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)$$

We analyze several cases.

Case j observes $S^{!*}$ and $V^{!*}$ and $n = o$: From the assumptions and the definition of $l^*.\mathit{safe}$ we have

$$m = n = o = p \qquad v_i \sim_k^V v'_i \qquad s_i \sim_j^S s'_i$$

for i from 1 to n. Using these facts we calculate as follows

$$\begin{array}{rll} & l^*.\mathit{safe}\ (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_n) & \\ = & l.\mathit{safe}\ (j,k)\ v_1\ s_1 \wedge \cdots \wedge l.\mathit{safe}\ (j,k)\ v_n\ s_n & \text{by definition } l^*.\mathit{safe} \\ = & l.\mathit{safe}\ (j,k)\ v'_1\ s'_1 \wedge \cdots \wedge l.\mathit{safe}\ (j,k)\ v'_n\ s'_n & \text{by SAFENOLEAK for } l \\ = & l^*.\mathit{safe}\ (j,k)\ (v'_1 \cdots v'_n)\ (s'_1 \cdots s'_n) & \text{by definition } l^*.\mathit{safe} \end{array}$$

and obtain the required equality.

Case j observes $S^{!*}$ and $V^{!*}$ and $n \neq o$: From the assumptions of the case and

$$(v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_m) \qquad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)$$

we have $n = m$ and $o = p$ and hence $m \neq p$. By the definition of $l^*.\mathit{safe}$ we immediately have

$$l^*.\mathit{safe}\ (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_o) = \bot = l^*.\mathit{safe}\ (v'_1 \cdots v'_m)\ (s'_1 \cdots s'_p),$$

as required.

Case j does not observe $S^{!*}$ and $V^{!*}$: By the definition of $l^*.\mathit{safe}$ we immediately have

$$l^*.\mathit{safe}\ (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_o) = \bot = l^*.\mathit{safe}\ (v'_1 \cdots v'_m)\ (s'_1 \cdots s'_p),$$

as required. □

$$\boxed{\dfrac{\begin{array}{c} l_1 \in S \overset{🔒}{\Longleftrightarrow} U \\ l_2 \in U \overset{🔒}{\Longleftrightarrow} V \end{array}}{l_1; l_2 \in S \overset{🔒}{\Longleftrightarrow} V}}$$

5.5.6 Lemma: Let $l_1 \in S \overset{🔒}{\Longleftrightarrow} U$ and $l_2 \in U \overset{🔒}{\Longleftrightarrow} V$ be dynamic secure lenses. Then $(l_1;l_2)$ is a dynamic secure lens in $S \overset{🔒}{\Longleftrightarrow} V$.

Proof: We prove each dynamic secure lens law separately.

▶ **GetPut**: Let (j,k) be a clearance in $\mathcal{C}$, let s be a string in S, and let v be a string in V such that $(l_1;l_2).\mathit{safe}\ (j,k)\ v\ s$. By the definition of $(l_1;l_2).\mathit{safe}$ we have

$$l_1.\mathit{safe}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s$$

Using this fact, we calculate as follows

$$\begin{array}{ll}
\quad (l_1;l_2).\mathit{put}\ v\ s & \\
= l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s & \text{by definition of } (l_1;l_2).\mathit{put} \\
\approx_k s & \text{by } \textsc{GetPut} \text{ for } l_1
\end{array}$$

and obtain the required equivalence.

▶ **PutNoLeak**: Let (j,k) be a clearance in $\mathcal{C}$, let v and v' be strings in V, and let s and s' be strings in S such that:

$$\begin{array}{ll}
s \sim_j s' & (l_1;l_2).\mathit{safe}\ (j,k)\ v\ s \\
v \sim_j v' & (l_1;l_2).\mathit{safe}\ (j,k)\ v'\ s'
\end{array}$$

By the definition of $(l_1;l_2).\mathit{safe}$ we have

$$l_1.\mathit{safe}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s \qquad l_1.\mathit{safe}\ (l_2.\mathit{put}\ v'\ (l_1.\mathit{get}\ s'))\ s'$$

Using these facts, we calculate as follows

$$\begin{array}{ll}
\quad (l_1;l_2).\mathit{put}\ v\ s & \\
= l_1.\mathit{put}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s & \text{by definition of } (l_1;l_2).\mathit{put} \\
\sim_j l_1.\mathit{put}\ (l_2.\mathit{put}\ v'\ (l_1.\mathit{get}\ s'))\ s' & \text{by } \textsc{PutNoLeak} \text{ for } l_1 \\
= (l_1;l_2).\mathit{put}\ v'\ s' & \text{by definition of } (l_1;l_2).\mathit{put}
\end{array}$$

and obtain the required equivalence.

▶ **SafeNoLeak**: Let (j,k) be a clearance in $\mathcal{C}$, let s and s' be strings in S, and let v and v' be strings in V such that $s \sim_j s'$ and $v \sim_j v'$. By GetNoLeak for l_1 we have that:

$$(l_1.\mathit{get}\ s) \sim_j (l_1.\mathit{get}\ s')$$

By PutNoLeak for l_2 we have

$$l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s) \sim_j l_2.\mathit{put}\ v'\ (l_1.\mathit{get}\ s')$$

By SafeNoLeak for l_1 have

$$l_1.\mathit{safe}\ (l_2.\mathit{put}\ v\ (l_1.\mathit{get}\ s))\ s = l_1.\mathit{safe}\ (l_2.\mathit{put}\ v'\ (l_1.\mathit{get}\ s'))\ s'$$

as required. □

$$\boxed{\dfrac{\begin{array}{c} E, F \text{ well-formed} \qquad E \cap F = \emptyset \qquad (E \,|\, F)^{!*} \\ p \sqsupseteq \bigvee\{k \mid k \text{ observes } E \cdot^! F \text{ and } F \cdot^! E\} \\ \forall\ (j,k) \in \mathcal{C}.\ \sim_j^E \subseteq \approx_k^E \end{array}}{\mathit{filter}\ E\ F \in (E \,|\, F{:}p)^* \overset{🔒}{\Longleftrightarrow} E^*}}$$

5.5.7 Lemma: Let E and F be well-formed security-annotated regular expressions such that $E \cap F = \emptyset$ and $(E\,|\,F)^{!*}$ and for every clearance (j,k) in $\mathcal{C}$ we have $\sim_j^E \subseteq \approx_k^E$. Then for every confidentiality label p such that $p \sqsupseteq \bigvee\{k \mid k \text{ observes } E{\cdot^!}F \text{ and } F{\cdot^!}E\}$, the dynamic secure lens *filter* E F is in $(E\,|\,F{:}p)^* \stackrel{🔒}{\Longleftrightarrow} E^*$.

Proof: We prove each dynamic secure lens law separately. To shorten the proof, we will abbreviate $\mathit{filter}\ E\ F$ as l.

▶ **GetPut:** Let (j,k) be a clearance in $\mathcal{C}$, let $v_1 \cdots v_n$ be a string in E^*, and let $s_1 \cdots s_m$ be a string in $(E\,|\,F{:}p)^*$ such that:

$$l.\mathit{safe}\ (j,k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)$$

Let $[e_1, \ldots, e_i]$ be the sequence of substrings of $s_1 \cdots s_m$ that belong to E. By the definition of $l.\mathit{safe}$ we have

$$k \text{ observes } E^{!*} \qquad (v_1 \cdots v_n) \approx_k \mathsf{str\_filter}\ E\ (s_1 \cdots s_m)$$

and hence

$$n = o \qquad v_i \approx_k^E e_i$$

for i from 1 to n. Using these facts, and the definition of str_unfilter it follows that

$$\mathit{hide}_k(\mathsf{str\_unfilter}\ F\ (v_1 \cdots v_n)\ (s_1 \cdots s_m)) = \mathit{hide}_k(s_1 \cdots s_m)$$

The required equivalence is immediate by the definition of $l.\mathit{put}$.

▶ **PutNoLeak:** Let (j,k) be a clearance in $\mathcal{C}$, let $v_1 \cdots v_m$ and $v'_1 \cdots v'_n$ be strings in E^* and $s_1 \cdots s_o$ and $s'_1 \cdots s'_p$ be strings in $E\,|\,F{:}p)$: * such that:

$$\begin{array}{ll} (v_1 \cdots v_m) \sim_j (v'_1 \cdots v'_n) & \quad l.\mathit{safe}\ (j,k)\ (v_1 \cdots v_m)\ (s_1 \cdots s_o) \\ (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p) & \quad l.\mathit{safe}\ (j,k)\ (v'_1 \cdots v'_n)\ (s'_1 \cdots s'_p) \end{array}$$

Let

$$[\vec{f_1}, \ldots, \vec{f_r}] \qquad \text{and} \qquad [\vec{f_1}', \ldots, \vec{f_t}']$$

be the sequences of contiguous elements of F in $s_1 \cdots s_o$ and $s'_1 \cdots s'_p$—i.e. elements not separated by an E. As j observes $E{\cdot^!}F$ and $F{\cdot^!}E$ we have $r = t$ and $\mathit{hide}_j(\vec{f_i}) = \mathit{hide}_j(\vec{f_i}')$ for i from 1 to r. Also, as j observes $E^{!*}$ we have $m = n$ and $v_i \approx_j v'_i$ for i from 1 to n. Using these facts and the definition of str_unfilter it follows that:

$$\begin{array}{rl} & \mathit{hide}_j(\mathsf{str\_unfilter}\ F\ (v_1 \cdots v_m)\ (s_1 \cdots s_o)) \\ = & \mathit{hide}_j(\mathsf{str\_unfilter}\ F\ (v'_1 \cdots v'_n)\ (s'_1 \cdots s'_p) \end{array}$$

The required equivalence is immediate by the definition of $l.\mathit{put}$.

▶ **SafeNoLeak:** Let (j,k) be a clearance in $\mathcal{C}$, let $v_1 \cdots v_n$ and $v'_1 \cdots v'_m$ be strings in E^*, and let $s_1 \cdots s_o$ and $s'_1 \cdots s'_p$ be strings in $(E\,|\,F{:}p)^*$ such that:

$$(v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_m) \qquad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)$$

We analyze two cases.

Case j observes $E{\cdot^!}F$ and $F{\cdot^!}E$ and j and k observe $E^{!*}$:

Let $[e_1, \ldots, e_r]$ and $[e'_1, \ldots, e'_t]$ be the sequences of substrings of $s_1 \cdots s_o$ and $s'_1 \cdots s'_p$ that belong to E. By the assumptions of the case we have that $r = t$ and $\mathit{hide}_j(e_i) = \mathit{hide}_j(e'_i)$ for i from 1 to n. As $\sim_j^E \subseteq \approx_k^E$ we also have $hide_k(t_i) = hide_k(t'_i)$ for i from 1 to n. Using these facts, we calculate as follows

$$\begin{array}{ll} \quad l.\mathit{safe}\ (j,k)\ (v_1 \ldots v_m)\ (s_1 \cdots s_o) & \\ = (v_1 \ldots v_n) \approx_k \textsf{str\_filter}\ E\ (s_1 \cdots s_o) & \text{by definition } l.\mathit{safe} \\ = (v_1 \ldots v_n) \approx_k (e_1 \cdots e_r) & \text{by definition } \textsf{str\_filter} \\ = hide_k(v_1 \cdots v_m) = hide_k(e_1 \cdots e_r) & \text{by definition } \approx_k \\ = hide_k(v'_1 \cdots v'_n) = hide_k(e'_1 \cdots e'_t) & \text{by facts above} \\ = (v'_1 \ldots v'_n) \approx_k (e'_1 \cdots e'_t) & \text{by definition } \approx_k \\ = (v'_1 \ldots v'_n) \approx_k \textsf{str\_filter}\ E\ (s'_1 \cdots s'_p) & \text{by definition } \textsf{str\_filter} \\ = l.\mathit{safe}\ (j,k)\ (v'_1 \ldots v'_n)\ (s'_1 \cdots s'_p) & \text{by definition } l.\mathit{safe} \end{array}$$

and obtain the required equivalence.

Case j does not observe $E \cdot^! F$ and $F \cdot^! E$ or j or k do not observe $E^{!*}$:
By the definition of *l.safe* we immediately have

$$\begin{array}{l} \quad l.\mathit{safe}\ (j,k)\ (v_1 \ldots v_m)\ (s_1 \cdots s_o) \\ = \bot \\ = l.\mathit{safe}\ (j,k)\ (v'_1 \ldots v'_n)\ (s'_1 \cdots s'_p) \end{array}$$

as required. □