Bidirectional Programming Languages

John Nathan Foster
University of Pennsylvania, jnfoster@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/edissertations
Part of the Databases and Information Systems Commons, and the Programming Languages and Compilers Commons

Recommended Citation
Foster, John Nathan, "Bidirectional Programming Languages" (2009). Publicly Accessible Penn Dissertations. 56.
http://repository.upenn.edu/edissertations/56

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/edissertations/56
For more information, please contact libraryrepository@pobox.upenn.edu.
Bidirectional Programming Languages

Abstract
The need to edit source 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 that computes the view from the source and another that handles updates. This rudimentary design is tedious for programmers, difficult to reason about, and a nightmare to maintain.

This dissertation presents bidirectional programming languages, which provide an elegant and effective mechanism for describing updatable views. Unlike programs written in an ordinary language, which only work in one direction, programs 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 define a general semantic space of well-behaved bidirectional transformations called lenses. Then, building on this foundation, we describe a particular language for defining lenses on strings with syntax based on the familiar regular operators (union, concatenation, and Kleene star). We present extensions to the basic framework that address the subtle complications that arise when lenses are used to manipulate, data containing unimportant details, ordered data, and confidential data.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Benjamin C. Pierce

Keywords
bidirectional languages, lenses, view update, regular expression types

Subject Categories
Databases and Information Systems | Programming Languages and Compilers

This dissertation is available at ScholarlyCommons: http://repository.upenn.edu/edissertations/56
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

Benjamin C. Pierce, Professor of Computer and Information Science
Supervisor of Dissertation

Jianbo Shi, Associate Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee
Zachary G. Ives, Associate Professor of Computer and Information Science
Val Tannen, Professor of Computer and Information Science
Philip Wadler, Professor of Theoretical Computer Science, University of Edinburgh
Steve Zdancewic, Associate Professor of Computer and Information Science
ABSTRACT
Bidirectional Programming Languages
John Nathan Foster
Supervisor: Benjamin C. Pierce

The need to edit source 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 that computes the view from the source and another that handles updates. This rudimentary design is tedious for programmers, difficult to reason about, and a nightmare to maintain.

This dissertation presents bidirectional programming languages, which provide an elegant and effective mechanism for describing updatable views. Unlike programs written in an ordinary language, which only work in one direction, programs 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 define a general semantic space of well-behaved bidirectional transformations called lenses. Then, building on this foundation, we describe a particular language for defining lenses on strings with syntax based on the familiar regular operators (union, concatenation, and Kleene star). We present extensions to the basic framework that address the subtle complications that arise when lenses are used to manipulate, data containing unimportant details, ordered data, and confidential data.
Acknowledgments

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

Throughout my time at Penn, my adviser Benjamin Pierce, has always been available to brainstorm about research, crank through a proof, hack on some code, or polish a piece of writing. Thank you Benjamin for setting the standard high, and for encouraging me, nurturing me, 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 wish to thank them for the many hours they spent reading my proposal and this dissertation. Their many insightful comments and valuable suggestions have improved it immensely.

Kim Bruce gave me my first research experience as an undergraduate and told me I would go to graduate school before I even knew I would want to. I am grateful for the opening into the research world he gave me and for close to ten years of guidance.

Alan Schmitt helped me navigate the transition to graduate school during the last few months of his postdoc and has continued to be a collaborator, informal mentor, and close friend in the years since. He also hosted me at Rhône-Alpes for a productive and extremely enjoyable visit in the spring of 2006. I also wish to thank my mentors Ravi Konuru, Jérôme
Siméon, and Lionel Villard for an enjoyable internship at IBM Research during the summer of 2007.

To the members of Penn’s PLClub and Database Groups, thank you for creating a stimulating and friendly 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 lunch runs, coffee breaks, and pleasant distractions from work. Thanks 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. The department wouldn’t function without you.

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

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

Philadelphia, PA
December 7, 2009

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

1 Introduction 1
   1.1 The View Update Problem 2
   1.2 Bidirectional Programming Languages 3
   1.3 Goal and Contributions 7
   1.4 Acknowledgments 10

2 Notational Preliminaries 11
   2.1 Basic Concepts 11
   2.2 Formal Language Theory 13
   2.3 Regular Languages 15

3 Basic Lenses 18
   3.1 Semantics 18
   3.2 Basic Lens Properties 24
   3.3 Combinators 29
   3.4 Summary 44

4 Quotient Lenses 45
   4.1 Semantics 48
   4.2 Quotient Lens Combinators 50
   4.3 Regular Operators 59
   4.4 Primitives 66
   4.5 Typechecking 69
   4.6 Example 73
List of Figures

3.1 Basic Lens Architecture ........................................ 19
3.2 Pseudocode for string_filter and string_unfilter............ 42
4.1 Lens architecture with “canonizers at the edges” .............. 47
4.2 Adding a canonizer to a quotient lens (on the left) ............ 51
5.1 Resourceful Lens Architecture .................................. 82
5.2 Alignment species .................................................. 102
6.1 Non-interference in Secure Lenses ............................... 113
7.1 Core Boomerang Syntax .......................................... 135
Chapter 1

Introduction

Most programs work in only one direction, from input to output. As Baker observed when he wrote “the S combinator cheerfully copies ... the K combinator knowingly kills” (1992), the very fundamentals of computation seem intrinsically unidirectional. However, the world is full of situations where, after computing an initial output, we need to be able to update it and then “compute backwards” to find a correspondingly modified input. There are numerous examples, across many areas of computing where these bidirectional transformations are needed:

**Data Synchronization**: bidirectional transformations bridge the gap between heterogeneous replicas (Brabrand et al., 2008; Kawanaka and Hosoya, 2006; Foster et al., 2007a).

**Data Management**: bidirectional transformations provide mechanisms for propagating updates to views (Foster et al., 2007b; Bohannon et al., 2006; Bancilhon and Spyratos, 1981; Dayal and Bernstein, 1982), for data exchange (Miller et al., 2001), and for converting data into new formats in schema evolution (Berdaguer et al., 2007).

**Software Engineering**: bidirectional transformations maintain the consistency of formal software models (Schürr, 1995; Stevens, 2007; Xiong et al., 2007).

**Security**: bidirectional transformations provide a way to do fine-grained sharing of confidential documents (Foster et al., 2009).

**Serialization**: bidirectional transformations map between the on-disk and in-memory representations of ad hoc and binary data (Fisher and Gruber, 2005; Eger, 2005).

**Systems Administration**: bidirectional transformations map between the flat low-level representations and higher-level structured representations of system configuration files (Lutterkort, 2008).
**Programming Languages:** bidirectional transformations handle boxing and unboxing of runtime values in embedded interpreters (Benton, 2005; Ramsey, 2003), and they convert between source code written in different high-level languages (Ennals and Gay, 2007).

**User Interfaces:** bidirectional transformations provide convenient editing interfaces for complex documents (Hu et al., 2008) and can also be used to maintain the consistency of graphical interface elements (Meertens, 1998; Greenberg and Krishnamurthi, 2007).

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 that transforms inputs to outputs and another that maps outputs to inputs—a rudimentary design that is tedious to construct, difficult to reason about, and a nightmare to maintain.

This dissertation proposes a different approach: languages in which every program can be read in two ways—from left to right as a function mapping inputs to outputs and from right to left as a function that propagates updates to outputs back to 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 databases, an area where they have been studied extensively because of their close connection to the **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 of sources and views throughout this dissertation.) The view update problem is the following: given an update \( u \) that transforms \( v \) to \( v' \), calculate a source update \( t \) (the “translation” of \( u \)) that transforms \( s \) to \( s' \) and makes the following diagram commute:

\[
\begin{array}{c}
S \longrightarrow q \longrightarrow v \\
\uparrow \quad \quad \quad \downarrow u \\
\quad t \quad \quad \quad \quad \quad \\
\downarrow \quad \quad \quad \quad \quad \\
S' \longrightarrow q \longrightarrow v'
\end{array}
\]
Despite extensive study over several decades, this problem remains largely unsolved. The main 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, certain updates to the view have many corresponding source updates. One can impose additional constraints to guide the selection of an update—e.g., requiring that the \( t \) have minimal “side effects” on the source—but when the query and schema languages are sufficiently complicated, calculating updates satisfying these additional constraints is intractable (Buneman et al., 2002). Even worse, when \( q \) is not surjective, some updates to the view produce structures that lie outside of the codomain of the query! It is not hard to see that in these situations it will be impossible to propagate the view update back to a corresponding source update—i.e., there is no \( t \) that makes the above diagram commute. Systems often choose to reject these updates, but doing so breaks the abstraction boundary between the source and view. It adds hidden constraints on how the view may be updated that are only revealed when the user of the view tries to propagate an update back to the source.

Because of these difficulties, views in relational database systems are generally read-only (except for those defined by very simple queries). In situations where an updatable view is needed, programmers have to rely on a variant of the rudimentary mechanism discussed above: they define a separate procedure called a trigger that the system executes whenever the view is modified. Triggers are arbitrary programs, so they can be used to implement any view update policy the programmer wants. However, they are not a very attractive solution. For one thing, checking that a trigger correctly propagates updates requires intricate, manual reasoning about the way it works in tandem with the query. Moreover, the 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 if 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 one that propagates updates. 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. Sup-
pose that the source is an XML document representing the names, dates, and nationalities of a
collection of classical music composers...

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

...and the views are comma-separated lines of ASCII text representing just their names and
dates—i.e., compared to the source, the nationalities and all XML formatting have been dropped:

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

Having computed this view, we might want to edit the ASCII output—e.g., correcting the error
in Sibelius’s death date and the misspelling in Briten’s name...

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

...and push the changes back into the original XML format:

```xml
<composers>
  <composer>
    <name>Jean Sibelius</name>
    <lived>1865-1957</lived>
    <nationality>Finnish</nationality>
  </composer>
</composers>
```
Here is a bidirectional program, written in the language of basic lenses described in Chapter 3, that denotes both transformations, from XML to ASCII and from ASCII to XML:

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

(* helper function *)
let xml_elt (tag:string) (body:lens) : lens =  
  del WHITESPACE*  
  . del ("<" . tag . ">")  
  . body  
  . del WHITESPACE*  
  . del ("</" . tag . ">")

(* helper lens *)
let composer : lens =  
  xml_elt "composer"  
  ( xml_elt "name" (copy (ALPHA . " " . ALPHA))  
    . ins ("",")  
    . xml_elt "lived" (copy (YEAR . "-" . YEAR))  
    . xml_elt "nationality" ( default (del ALPHA) (fun x:string -> "Unknown") ) )

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

In the forward direction, it can be read as an ordinary function on strings. The first few lines define regular expressions (in standard POSIX notation) for whitespace, alphabetic strings, and years. The helper function `xml_elt` defined next takes a string `tag` and a lens `body` as arguments and returns a lens that processes an XML element named `tag`. It first removes all of the XML formatting for the element and then processes the children of the element using `body`. The concatenation operator `.` combines lenses in the obvious way. The `composer` lens defined next, instantiates `xml_elt` several times to construct a lens that handles XML elements representing 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 final definition, the `composers`
lens, processes XML elements containing a sequence of composers. It uses union (|) and Kleene star (*) to iterate composer over the list of composers, adding a newline between each. Because this is a bidirectional program, we can also run it backwards. The details of how this works are not important for now (see Chapter 3 for precise definitions); the key point is that we can use the same composers lens to map a modified ASCII view back to an updated XML source.

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

```ocaml
def composers_forward s =
  let sx = parse_string s in
  List.fold_left
    (fun acc sxi ->
      let [nx;dx;_] = children xi in
      let sep = if acc <> "" then "\n" else "" in
      (acc ^ sep ^ pcdata nx ^ " , " ^ pcdata dx))
    ""
    (children sx)
```

(The functions parse_string, children, and pcdata are functions from a simple XML library). It parses the source string s into an XML structure, and then folds down the list of composer elements, extracting the name and dates of each composer and adding them to the view. The corresponding composers_backward function is written as follows:

```ocaml
def composers_backward v s =
  let sx = parse_string s in
  let sl = children sx in
  let vl = split "\n" v in
  let rec aux acc vl sl = match vl with
    | [] -> to_string (element "composers" (List.rev acc))
    | vh::vt ->
      let [n;d] = split ", " vi in
      let nx = pcdata_element "name" n in
      let dx = pcdata_element "lived" d in
      let c,sl' = match sl with
        | sh::st -> (List.nth (children sxi) 2,st)
        | [] -> ("Unknown",[]) in
      let cx = pcdata_element "nationality" c in
      let acc' = (element "composer" [nx;dx;cx])::acc in
      aux acc' vt st in
  aux [] vl sl
```

(Again, the functions `pCDATA_element`, `element`, and `to_string` all come from the XML library.) It takes two arguments, an updated view \( v \) and the original source \( s \), and weaves them together, propagating the names and dates from the view, and restoring the nationalities from the source.

One advantage of the lens program over the OCaml version is that we only have to write one program instead of two. Another is that key correctness properties can be proven automatically for the lens program. In particular, the type system for lenses described in Chapter 3 guarantees 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 them in either order yields the identity function. By contrast, to verify that the two OCaml programs correctly implement the updatable view, we would need to establish 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, however, is that it is much easier to maintain. Suppose that, for whatever reason, we decide that the character used to separate the fields in each line of ASCII text should be “;” instead of “,”. To update the lens program, we would simply replace the single occurrence of “,” with a “;”. Both functions denoted by the lens would change and the typechecker would reverify the well-behavedness properties automatically. The OCaml programs, however, have multiple occurrences of “,”. We would need to make coordinated changes to both functions—something that would be easy to get wrong, especially if the example were slightly more complicated. We would also need to modify and reverify the well-behavedness proof by hand. Thus, even for this almost trivial example, the bidirectional language is a much more attractive option.

### 1.3 Goal and Contributions

The goal of this dissertation is to show that bidirectional languages are an effective way of defining updatable views. Its contributions are divided between three broad areas: foundations, language design, and implementation.
Foundations

One of the key benefits of using a bidirectional language is automatic reasoning about correctness. However, before we can even talk about correctness, we need to characterize the transformations that have the properties we want. Many current systems for building updatable views are constructed in an ad hoc fashion and have a rather casual attitude about correctness. The first contribution of this dissertation is a general semantic framework of well-behaved bidirectional transformations called lenses that organizes this whole area and provides a clean foundation for building reliable updatable views. Chapter 3 describes the framework of basic lenses, which provide the formal underpinnings for the whole project. Lenses are based on semantic laws that are closely related to the classic conditions on view update translators that have been proposed in databases. Chapter 4 describes quotient lenses, which relax the basic lens laws by allowing certain specified portions of the source and view to be treated as “unimportant”. This generalization is motivated by experience writing lenses for real-world data, which often contain ignorable details such as whitespace. Chapter 5 describes resourceful lenses which address the issue of alignment, which comes up when the source and view are ordered structures. In general, to do a good job of propagating an updates to ordered structures, a lens needs to be able to compute a correspondence between pieces of the view and the pieces of the source. Resourceful lenses extend basic lenses with new mechanisms for calculating and using these correspondences, which we call alignments, and new properties that ensure that they use alignment information correctly. Finally, Chapter 6 describes secure lenses, which extend basic lenses with additional guarantees about security properties of data. These properties ensure that lenses do not reveal certain specified regions of the source in the view, and that propagating updates to the view does not corrupt other specified portions of the source.

Language Design

Interpreting programs bidirectionally requires changing the way that we describe computations—some constructs that make sense in ordinary programming languages do not make sense as lenses. The second contribution of this dissertation is a concrete design for a lens language with natural, compositional syntax and a type system that guarantees the lens laws. For simplicity, we focus our efforts on languages for transforming for strings rather than richer structures
such as trees or complex values. However, even strings are simple structures, they still expose many fundamental issues. Additionally, because there is also a lot of string data in the world—textual databases, structured documents, scientific data, simple XML, and all kinds of ad hoc data—having a language for defining lenses on strings is quite useful in practice.

Chapter 3 introduces the core language studied throughout this dissertation. It features a set of combinators for transforming strings that includes generic operators (identity, constant, sequential composition), the regular operators (union, concatenation, Kleene star), and some additional operators we have found useful in applications (filter, swap, merge, etc.). Types play a central role in this language. Some of our operators only make sense as lenses when specific side conditions are met, and we use types to express and verify these constraints. To ensure that typechecking can be mechanized, we use regular expressions as types. Regular expressions balance the tradeoffs between precision and decidability: they describe data schemas at a reasonably high level of detail and all of the properties of regular expressions 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 4 extends it with new constructs (canonizers, quotient operators) for dealing with ignorable data. Chapter 5 adds features (chunks, keys, thresholds) for aligning data, and Chapter 6 extends the language with a more refined type system based on an information-flow analysis that tracks security properties of data including confidentiality and integrity.

Implementation

The final contribution of this dissertation is an implementation of our ideas in a full-blown programming language called Boomerang. This language, described in Chapter 7, demonstrates our thesis that bidirectional languages are an effective way to specify updatable views. To make it easier for programmers to develop lens programs of realistic size, Boomerang includes all the features usually found in a modern functional language: first-order functions, an expressive type system, modules, unit tests, etc. Additionally, Boomerang has several features specifically designed for expressing string transformations: built-in regular expressions, operator overloading, subtyping, and syntax for describing lenses based on grammars. We have implemented a
full working Boomerang prototype and used it to develop lenses for a number of real-world data formats. Although this prototype is not industrial strength (in particular, we have not yet attempted to optimize performance), it is efficient enough to handle many examples of realistic size—e.g., our largest lens is a 4KLoc program that computes views over XML documents representing scientific databases.

Lenses have been applied in industry. Augeas (Lutterkort, 2008), a tool developed by RedHat, Inc. for managing operating system configurations, is a near-clone of Boomerang (it supports some additional constructs for building trees). Augeas programmers have developed lenses for handling nearly all standard Linux configuration file formats. This independent application of our ideas provides additional evidence that the concept of lenses is robust, and that our design for Boomerang is practical.

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 3 is based on an article by Foster, Greenwald, Moore, Pierce, and Schmitt (2007b). The string lenses presented in that chapter are based on material from a paper by Bohannon, Foster, Pierce, Pilkiewicz, and Schmitt (2008). Chapter 4 is a revised version of a paper by Foster, Pilkiewicz, and Pierce (2008b). Chapter 5 describes new work with Davi Barbosa and Julien Cretin that generalizes the dictionary lenses described in the same paper by Bohannon, Foster, Pierce, Pilkiewicz, and Schmitt (2008). Chapter 6 is a revised version of a CSF paper by Foster, Pierce, and Zdancewic (2009). The design and implementation of Boomerang described in Chapter 7 was done in cooperation with Michael Greenberg, Alexandre Pilkiewicz, Benjamin Pierce, and Alan Schmitt. Davi Barbosa and Julien Cretin also helped polish and extend many aspects of the Boomerang implementation. Boomerang’s precise type system was designed and implemented in collaboration with Michael Greenberg. Its grammar notation was designed with help from Adam Magee and Danny Puller. Finally, Chapter 8 is an expanded version of survey of related work on lenses by Foster, Greenwald, Moore, Pierce, and Schmitt (2007b).
Chapter 2

Notational Preliminaries

This chapter fixes notation for a standard mathematical concepts (sets, relations, functions, etc.) and notions from formal language theory (strings, languages, regular languages, etc.). It is intended as a reference and can be skipped on a first reading and referred back to as needed.

2.1 Basic Concepts

We start by fixing notation for a few standard mathematical concepts.

2.1.1 Notation [Sets]: We write \(\{x_1, \ldots, x_n\}\) to denote the set containing elements \(x_1\) to \(x_n\). We often describe sets by comprehension, writing \(\{x \in S \mid P(x)\}\) to denote the subset of \(S\) consisting of elements satisfying a predicate \(P\). We write \(x \in S\) to indicate that \(x\) is an element of \(S\) and \(\{\}\) for the empty set. We write \((S_1 \cup S_2)\) for the union, \((S_1 \cap S_2)\) for the intersection, and \((S_1 - S_2)\) for the difference of sets \(S_1\) and \(S_2\).

2.1.2 Definition [Relation]: A relation \(R\) on sets \(S\) and \(T\) is a subset of \((S \times T)\).

2.1.3 Definition [Relation Composition]: Let \(Q \subseteq (S \times T)\) and \(R \subseteq (T \times U)\) be relations. The composition of \(Q\) and \(R\), written \(R \circ Q\) is the relation on \(R\) and \(U\) defined as follows:\(^1\)

\[
(R \circ Q) \triangleq \{(s, u) \in (S \times U) \mid \exists t \in T. (s, t) \in Q \land (t, u) \in R\}
\]

\(^1\)Note the order of the relations being composed. We write the relations in this order so that function composition is a special case of relation composition.
2.1.4 Definition [Inverse]: The inverse of a relation $R \subseteq (S \times T)$ on $S$ and $T$ is the relation $R^{-1}$ on $T$ and $S$ defined as:

$$R^{-1} \triangleq \{(t, s) \in (T \times S) \mid (s, t) \in R\}$$

2.1.5 Definition [Equivalence Relation]: A binary relation $R \subseteq (S \times S)$ on $S$ is an equivalence if it is

- reflexive: i.e., $\forall s \in S. (s, s) \in R$,
- symmetric: i.e., $\forall s, s' \in S. (s, s') \in R \implies (s', s) \in R$, and
- transitive: i.e., $\forall s, s', s'' \in S. (s, s') \in R \land (s', s'') \in R \implies (s, s'') \in R$.

2.1.6 Definition [Identity Relation]: The identity relation $\text{Id}(S)$, the finest equivalence relation on $S$, is defined as $\text{Id}(S) \triangleq \{(s, s) \in (S \times S) \mid s \in S\}$.

2.1.7 Definition [Total Relation]: The total relation $\text{Tot}(S)$, the coarsest equivalence relation on $S$, is defined as $\text{Tot}(S) \triangleq \{(s, s') \in (S \times S) \mid s \in S \land s' \in S\}$.

2.1.8 Definition [Transitive Closure]: The transitive closure of a binary relation $R \subseteq (S \times S)$ on $S$, written $\text{TransClosure}(R)$ is the smallest transitive relation $S$ that contains $R$.

2.1.9 Definition [Function]: A relation $f \subseteq (S \times T)$ is a function if and only if for every element of $S$ there is at most one element of $T$ related to it by $f$:

$$\forall s \in S, s' \in S, t \in T. (s, t) \in f \land (s, t') \in f \implies t = t'$$

We write $f(s) = t$ instead of $(s, t) \in f$ when $f$ is a function.

2.1.10 Notation [Defined]: Let $f \in S \to T$ be a function and let $s \in S$. We say that $f$ is defined on $s$, written $f(s) \downarrow$, if $f(s) = t$ for some $t \in T$ and $f(s) = \bot$ otherwise.

2.1.11 Convention: Throughout this dissertation, we take function application to be strict. That is, if $g \in S \to T$ and $f \in T \to U$ are functions then $f(g(x)) \downarrow$ implies $g(x) \downarrow$.

2.1.12 Definition [Domain, Codomain]: Let $f \in S \to T$ be a function. The domain of $f$, written $\text{dom}(f)$, is the subset of $S$ on which $f$ is defined as $\text{dom}(f) \triangleq \{s \in S \mid f(s) \downarrow\}$. Similarly, the codomain of $f$ written $\text{cod}(f)$ is the subset of $T$ that is mapped to by $f$, defined as $\text{cod}(f) \triangleq \{t \in T \mid \exists s \in S. f(s) = t\}$. 

12
2.1.13 Definition [Total Function]: A function \( f \in S \rightarrow T \) is total (on \( S \)) if \( \text{dom}(f) = S \).

2.1.14 Definition [Injective Function]: A function \( f \in S \rightarrow T \) is injective if for every \( t \in T \) there is at most one \( s \in S \) such that \( f(s) = t \):

\[
\forall s \in S, s' \in S, t \in T. \ f(s) = t \land f(s') = t \implies s = s'
\]

2.1.15 Definition [Surjective Function]: A function \( f \in S \rightarrow T \) is surjective if for every element of \( T \) there exists at least one element of \( S \) such that \( f(s) = t \).

2.1.16 Definition [Bijective Function]: A function \( f \in S \rightarrow T \) is bijective if it is injective and surjective.

2.1.17 Definition [Curry and Uncurry]: Let \( f \in A \times B \rightarrow C \) be a function. The function \( \text{curry} \ f \in A \rightarrow B \rightarrow C \) is defined as:

\[
\text{curry} \ f \triangleq \lambda a : A. \lambda b : B. \ f(a, b)
\]

Likewise, let \( \text{uncurry} \ f \in A \rightarrow B \rightarrow C \) be a function. The function \( \text{uncurry} \ f \in A \times B \rightarrow C \) is defined as:

\[
\text{uncurry} \ f \triangleq \lambda (a, b) : (A \times B). \ f \ a \ b
\]

2.2 Formal Language Theory

Now we turn to formal language theory, fixing notation for alphabets, strings, languages, etc.

2.2.1 Definition [Alphabet]: Let \( \Sigma = \{c_1, \ldots, c_n\} \) be a finite set of symbols (e.g., ASCII). We call the set \( \Sigma \) an alphabet and every symbol \( c \in \Sigma \) a character.

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

2.2.3 Notation [Empty String]: We write \( \epsilon \) for the empty string.

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

2.2.5 Notation [String Concatenation]: We write \( u \cdot v \) for the concatenation of strings \( u \) and \( v \).
2.2.6 Definition [Language]: A language is a subset of \( \Sigma^* \).

2.2.7 Notation [Representative Function]: The function \( \text{representative}(\cdot) \in 2^{(\Sigma^*)} \rightarrow \Sigma^* \) takes as a language \( L \subseteq \Sigma^* \) as an argument. It yields an arbitrary string belonging to \( L \) if \( L \) is non-empty and is otherwise undefined.

2.2.8 Definition [Language Concatenation]: The concatenation of languages \( L_1 \subseteq \Sigma^* \) and \( L_2 \subseteq \Sigma^* \), written \( L_1 \cdot L_2 \), is the language containing every concatenation of a string in \( L_1 \) with a string in \( L_2 \):
\[
L_1 \cdot L_2 = \{ u \cdot v \mid u \in L_1 \text{ and } v \in L_2 \}
\]

2.2.9 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:
\[
L^0 = \{ \epsilon \}
\]
\[
L^{i+1} = L^i \cdot L
\]

Note that \( L^1 = L \) as \( (L^0 \cdot L) = (\{\epsilon\} \cdot L) \).

2.2.10 Definition [Kleene Closure]: The Kleene closure of \( L \), written \( L^* \), is the union of every iteration of \( L \):
\[
L^* = \bigcup_{i=0}^{\infty} L^i
\]

Many of our definitions 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.

2.2.11 Definition [Unambiguous Concatenation]: Two languages \( L_1 \subseteq \Sigma^* \) and \( L_2 \subseteq \Sigma^* \) are unambiguously concatenable, written \( L_1 \vdash L_2 \), 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, many of the definitions that follow 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.
2.2.12 Definition [Unambiguous Iteration]: A language $L \subseteq \Sigma^*$ is unambiguously iterable, written $L^\dagger$, 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 $n$.

2.3 Regular Languages

This section reviews the regular languages, which enjoy many nice properties including closure under the boolean operators as well as algorithms for deciding equivalence, inclusion, disjointness, and the ambiguity conditions just defined. There are many equivalent ways to define regular languages. We give a syntactic characterization in terms of regular expressions here.

2.3.1 Definition [Regular Expressions]: The set of regular expressions over $\Sigma$ is the smallest set generated by the following grammar

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

where $u$ ranges over strings (including $\epsilon$) in $\Sigma^*$.

2.3.2 Definition [Semantics of Regular Expressions]: The set of strings $[R]$ denoted by a regular expression $R \in \mathcal{R}$ is defined by structural induction on $R$ as follows:

$$[u] \triangleq \{u\}$$

$$[\emptyset] \triangleq \{\}$$

$$[R_1 \cdot R_2] \triangleq [R_1][R_2]$$

$$[R_1 | R_2] \triangleq [R_1] \cup [R_2]$$

$$[R_1]^* \triangleq [R_1]^*$$

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

2.3.4 Fact [Closure Properties]: Let $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ be regular languages. The following languages are also regular (Hopcroft and Ullman, 1979):

- the intersection $(L_1 \cap L_2)$ of $L_1$ and $L_2$,
- the difference $(L_1 - L_2)$ of $L_1$ and $L_2$, 

15
• the left quotient \((L_1 \backslash L_2) = \{ u \in \Sigma^* \mid \exists v \in L_1. u \cdot v \in L_2 \}\) of \(L_1\) by \(L_2\), and
• the right quotient \((L_1 / L_2) = \{ u \in \Sigma^* \mid \exists v \in L_2. u \cdot v \in L_1 \}\) of \(L_1\) by \(L_2\).

2.3.5 Fact [Emptiness]: It is decidable whether a regular language \(L \subseteq \Sigma^*\) is empty (Hopcroft and Ullman, 1979). As regular languages are effectively closed under negation and intersection, inclusion and equivalence for regular languages are also decidable.

We finish this preliminary section by showing that ambiguous concatenation and iteration is decidable for regular languages.

2.3.6 Lemma: It is decidable whether the concatenation of regular languages \(L_1 \subseteq \Sigma^*\) and \(L_2 \subseteq \Sigma^*\) is ambiguous.

Proof: Let \(L_1 \subseteq \Sigma^*\) and \(L_1 \subseteq \Sigma^*\) be regular languages. Define languages \(S_1\) and \(P_2\) as follows:

\[
S_1 \triangleq (L_1 \backslash L_1) = \{ v \mid \exists u \in L_1. u \cdot v \in L_1 \}
\]

\[
P_2 \triangleq (L_2 / L_2) = \{ u \mid \exists v \in L_2. u \cdot v \in L_2 \}
\]

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 \(|u_1| > |v_1|\) and \(|u_2| < |v_2|\). 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 $u_2 \neq w \cdot u_2$, which contradicts the assumption that $L_1$ and $L_2$ are unambiguously concatenable. 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. 

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

Proof: Let $L \subseteq \Sigma^*$. We will prove that $\epsilon \not\in L$ and $L \cdot ! L^*$ if and only if $L$ is unambiguously iterable.

$(\Rightarrow)$ Suppose, for a contradiction, that $\epsilon \not\in L$ and $(L \cdot ! L^*)$ but $L$ 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 $\epsilon \not\in L$ 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 $\epsilon \not\in L$ 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 $n$.

Hence, we have a contradiction. 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 $\epsilon \not\in L$ and $(L \cdot ! L^*)$. 

□
Chapter 3

Basic Lenses

“Never look back unless you are planning to go that way.”
—Henry David Thoreau

This chapter presents the syntax and semantics of a simple bidirectional language for transforming strings. Although the computational power of this language is limited—it is based on finite state transducers (Berstel, 1979)—it is still powerful enough to express a large class of examples. It also cleanly illustrates many of the fundamental choices in the design of lenses. We begin the chapter in Section 3.1 by defining the semantics of lenses and developing some of their main properties. Section 3.2 develops some of the main properties of basic lenses. Section 3.3 presents syntax for a core set of string lenses combinators. We conclude in Section 3.4.

3.1 Semantics

Before we can delve into concrete language design, we need a framework in which we can precisely evaluate whether a bidirectional transformation correctly implements a view, or not. In this section, we present a natural mathematical space of well-behaved bidirectional transformations called basic lenses, and we develop some of fundamental properties of lenses.

Although we will primarily focus on lenses for strings in this dissertation, the semantics of lenses can be formulated over arbitrary structures. Thus, in this section, we will work in a completely abstract setting, parameterizing all of our definitions on a universe \( \mathcal{U} \) of objects. Later, we will instantiate this framework by picking \( \mathcal{U} \) to be the set of ASCII strings.
3.1.1 Definition [Basic Lens]: Fix a universe $\mathcal{U}$ of objects and let $S \subseteq \mathcal{U}$ (“sources”) and $V \subseteq \mathcal{U}$ (“views”) be sets of objects. A basic lens $l$ from $S$ to $V$ comprises three total functions

$$l.\text{get} \in S \rightarrow V$$
$$l.\text{put} \in V \rightarrow S \rightarrow S$$
$$l.\text{create} \in V \rightarrow S$$

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

$$l.\text{get} (l.\text{put} v s) = v \quad \text{(PutGet)}$$

$$l.\text{get} (l.\text{create} v) = v \quad \text{(CreateGet)}$$

$$l.\text{put} (l.\text{get} s) s = s \quad \text{(GetPut)}$$

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

The intuition behind the names $\text{get}$, $\text{put}$ and $\text{create}$ is that the $\text{get}$ function “lifts” a view out of a source structure, while the $\text{put}$ function “pushes down” an updated view into the original source, yielding a new source that reflects the modifications made in the view. (We will often say “put $v$ into $s$ (using $l$)” instead of “apply $l$’s put function to $v$ and $s$.”) The $\text{create}$ 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 uses the view to manufacture a new source “from scratch,” filling in any missing data with defaults. The diagram in Figure 3.1 depicts these components graphically.

Broadly speaking, the semantics of lenses is designed to provide three important guarantees:
1. Lenses are robust abstractions. Users can make arbitrary updates to views without having to consider whether their changes can be propagated back to the source.

2. Lenses propagate view updates “exactly” to the underlying source.

3. When possible, lenses preserve the source data that is hidden by the view.

Formally, these guarantees are ensured by the “round-tripping” laws given in Definition 3.1.1, which govern the handling of data as it is mapped back and forth between the source and view, and by the requirement that \( \text{put} \) be a total function. In the next few paragraphs, we describe how these formal conditions guarantee the informal properties enumerated above, and we discuss the benefits and tradeoffs of our design choices.

**Robustness** A fundamental choice in the design of a bidirectional language is whether it handles every update to a view or only certain updates. Hegner has coined the terms “closed views” and “open views” to describe these alternatives (1990). Many frameworks take the second alternative—i.e., they allow updates to fail. This gives these systems maximum flexibility, since the decision about whether to propagate a particular update can be made dynamically, on a case-by-case basis. However, although they can handle many transformations that would not be valid as lenses, these frameworks also have a significant drawback: their views are “leaky” abstractions of the source. In particular, users cannot treat views like ordinary data structures because when they update the view, they need to consider the possibility that their update will be incompatible with the hidden information in the source and will be rejected. By contrast, the \( \text{put} \) 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 changed dramatically. In the kinds of applications where we plan to use lenses—e.g., in data synchronizers (Foster et al., 2007a), which are often run unsupervised—it critical that views be robust abstractions.

An interesting effect of this choice is that, in practice, it forces lens languages to have extremely precise type systems. The way that a lens can free itself from the obligation to handle a particular view is to exclude it from the set of views mentioned in its type. Lenses that manipulate data in complicated ways typically need to be able to describe the type of that data in correspondingly precise ways.
**Exact Translation**  Another fundamental consideration in the design of a bidirectional language is the handling of updates. Many systems require that they be translated “exactly” to the source—i.e., to a new source that reflects all of the changes made to the view. In the framework of lenses, the *PutGet* law is the bedrock principle that guarantees this property. Formally, it stipulates that, given an updated view and an old source, the *put* function must produce a new source that the *get* function maps back to the very same view. As an example of a transformation that violates *PutGet*, fix an alphabet $\Sigma$, let $S = \Sigma^*$ be the set of strings over $\Sigma$, let $V = (\Sigma^* \times \mathbb{N})$ be the set of pairs of strings and natural numbers, and define functions $\text{get}$, $\text{put}$ and $\text{create}$ as follows:

$$
\begin{align*}
\text{l.} \, \text{get} \, u & = (u, 0) \\
\text{l.} \, \text{put} \, (u', n) \, u & = u' \\
\text{l.} \, \text{create} \, (u', n) & = u'
\end{align*}
$$

If we use $l$ to put the view (“abc”, 1) into the source “xyz”, we get the new source “abc”. However, the *get* function maps this string to (“abc”, 0) not (“abc”, 1)—i.e., *PutGet* fails. Intuitively, the reason for this failure is some of the information in the view—the number—is not propagated to the source and so updates to the number are lost. It turns out that requiring the *PutGet* law implies that the *put* function must be injective (the precise condition, semi-injectivity, is slightly stronger; see Lemma 3.2.1 (2) below).

**Source Integrity**  The third key consideration in the design of lenses is the integrity of the underlying source data. In some cases, for the *put* function to translate updates to the view exactly, it needs to modify the underlying source, including the parts of the source that are not exposed in the view. However, to the extent possible, we would like lenses to preserve this hidden source data—i.e., *put* should not have extra “side effects”. It turns out that there are range of conditions one can impose to ensure the integrity of source data. For lenses, we choose a simple condition, embodied in the *GetPut* law, which stipulates that the *put* function must restore the original source exactly when its arguments are a view $v$ and a source $s$ that generates the very same view—i.e., such that $\text{get} \, s = v$. As example of a transformation that does not obey *GetPut* (it does obey *PutGet*), let $S = (\Sigma^* \times \mathbb{N})$ be the set of pairs of strings over $\Sigma$ and
natural numbers, let $V = \Sigma^*$ be the set of strings, and define $\text{get}$, $\text{put}$, and $\text{create}$ as follows:

\[
\begin{align*}
&l.\text{get} \; (u, n) = u \\
&l.\text{put} \; u' \; (u, n) = (u', 0) \\
&l.\text{create} \; u' = (u', 0)
\end{align*}
\]

If we use $l$ to compute a view from (“abc”, 1) and immediately put it back, we get (“abc”, 0), which is a different source than the one we started with. One way of seeing the problem with this transformation is that the $\text{put}$ function has extra side effects on the source—it sets the number to 0 even when the view is unchanged. The GETPUT law restricts the effects that $\text{put}$ can have on the source by forcing it to have no effect at all, whenever it is possible for it to do so without disobeying the other laws.

We might be tempted to go a step further and stipulate that the $\text{put}$ function must always have “minimal” effects on the source, and not only when the view is unchanged. Unfortunately, even stating this condition requires 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 agnostic to the way that updates are expressed—the $\text{put}$ function takes the whole state of the updated view (in database terminology, a materialized view) as input rather than an explicit operation in an update language. Being state based makes it easy to use lenses in many different application scenarios (since applications do not need to be retooled to manipulate the view via operations in a special update language) and to use lenses with data in non-standard and ad hoc formats (which do not usually come equipped with canonical update languages). But it makes it hard to express properties formulated in terms of updates. So, because lenses are agnostic to the way that updates are expressed, we only impose the weaker GETPUT law, which can be stated abstractly and without assuming any particular notion of update. However, even though the GETPUT law only provides a relatively loose constraint on the behavior of lenses, it is still a useful tool for designing lens primitives. We have used it many times to generate, test, and reject candidate transformations.

A different idea for ensuring the integrity of source data is to require that the $\text{put}$ 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 (Bancilhon and Spyратos, 1981). 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 \(\text{get}\) 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 \(\text{put}\) function pairs up the new view \(v'\) with the old complement \(c\) and applies the witness to the isomorphism in the other direction to \((v', c)\) to obtain the updated source. Importantly, as the \(\text{put}\) function is implemented by an injective function from \((V \times C)\) to \(S\), it necessarily propagates all of the information in the complement back to the source—i.e., the complement is held “constant”.

We can formulate a law for lenses that is equivalent to the constant complement condition by stipulating that the source obtained after doing two \(\text{put}\)s in a row must be the same as doing just the second:

\[
\text{lp} \cdot \text{put} \; v' \; (\text{lp} \cdot \text{put} \; v \; s) = \text{lp} \cdot \text{put} \; v' \; s \quad \text{(PutPut)}
\]

This law does not explicitly mention complements, but it forces the \(\text{put}\) function to restore all of the information in the old source that is not reflected in the view—i.e., a complement—because if the lens did not restore some of this information to the intermediate source produced as the result of the first \(\text{put}\), then that information would not be available for the second \(\text{put}\).

Unfortunately, insisting that every lens obey \text{PutPut} is a draconian restriction that rules out many transformations that are indispensable in practice. As an example, of a lens that does not obey \text{PutPut}, let \(S = (\Sigma^* \times \mathbb{N}) \; \text{list}\) be the set of lists of pairs of strings and natural numbers, let \(V = (\Sigma^* \; \text{list})\) be the set of lists of strings, and define \(\text{get}, \text{put}\) and \(\text{create}\) as follows:

\[
\begin{align*}
\text{lp} \cdot \text{get} \; [(u_1, n_1), \ldots, (u_k, n_k)] &= [u_1, \ldots, u_k] \\
\text{lp} \cdot \text{put} \; [u'_1, \ldots, u'_l] \; [(u_1, n_1), \ldots, (u_k, n_k)] &= [(u'_1, n'_1), \ldots, (u'_l, n'_l)]
\end{align*}
\]

where \(n'_i = \begin{cases} n_i & \text{for } i \in \{1, \ldots, \min(l, k)\} \\ 0 & \text{for } i \in \{k + 1, \ldots, l\} \end{cases}\)

\[
\text{lp} \cdot \text{create} \; [u'_1, \ldots, u'_l] = [(u'_1, 0), \ldots, (u'_l, 0)]
\]

The \(\text{get}\) component of this lens takes a list of pairs of strings and numbers and projects away the numbers. The \(\text{put}\) component takes a view and a source and weaves them together, propagating the strings from the view and restoring as many numbers from the source as there are strings in the view (because \(\text{put}\) must be total, it needs to handle cases where the source and view lists have
different lengths). To see why \texttt{PutPut} fails, observe that when the list of strings in the view has fewer elements than the source list, the \texttt{put} function discards some of the numbers in the source, as it must to satisfy \texttt{PutGet}. For example, putting \(["a", "b"]\) into \([(["a", 1], (["b", 2], (["c", 3]))\) discards the 3 and yields \([(["a", 1], (["b", 2]))\). This accurately reflects the change made to the view, but if we \texttt{put} \(["a", "b", "c"]\) into this intermediate source we get \([(["a", 1], (["b", 2], (["c", 0]))\), which is different than the result \([(["a", 1], (["b", 2], (["c", 3]))\) we would have obtained if we had put this view into the original source without doing the inner \texttt{put}. Because many applications depend on transformations that do not obey \texttt{PutPut}, we do not require it. However, we pay special attention to lenses that do obey it, calling them \textit{very well behaved}.

3.1.2 Definition [Very Well Behaved Lens]: A lens \(l \in S \implies V\) is \textit{very well behaved} if and only if it obeys the \texttt{PutPut} law for all views \(v' \in V\) and \(v \in V\) and every source \(s \in S\).

Interestingly, the weaker integrity guarantee embodied in the \texttt{GetPut} law can be formulated as a special case of \texttt{PutPut}. The \texttt{PutTwice} law stipulates that the effect of doing two \texttt{puts} using \textit{the same view} must be the same as doing just a single \texttt{put}:

\[
l.\text{put} \ v \ (l.\text{put} \ v \ s) = l.\text{put} \ v \ s \tag{\text{PutTwice}}
\]

We show in Lemma 3.2.2, below, that every lens obeys \texttt{PutTwice}.

Another important class of lenses are those whose \texttt{put} functions do not use their source argument. We call these lenses \textit{oblivious}.

3.1.3 Definition [Oblivious Lens]: A lens \(l \in S \implies V\) is \textit{oblivious} if \((l.\text{put} \ v \ s) = (l.\text{put} \ vs')\) for all views \(v\) in \(V\) and sources \(s\) and \(s'\) in \(S\).

Both functions denoted by an oblivious lenses are bijective. They are trivially very well behaved, since the result computed by \texttt{put} never depends on its source argument.

3.2 Basic Lens Properties

Now we use the semantics of lenses to derive some of their basic properties. We begin by proving some lemmas establishing some elementary facts about lenses. The most interesting result, presented at the very end of this section, is an alternate characterization of lenses entirely in terms of \texttt{put} functions.
3.2.1 Lemma: For every \( l \in S \rightleftharpoons V \), the following facts hold:

1. \( l.get \) and \((\text{uncurry } l.put)\) 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. \( l.create \) is an injective function.

Proof: Let \( l \in S \rightleftharpoons V \) be a lens. We prove each fact separately.

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

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

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

\[
\begin{align*}
v &= l.get (l.put v s) & \text{by } \text{PutGet} \text{ for } l \\
  &= l.get (l.put v' s') & \text{by assumption} \\
  &= v' & \text{by } \text{PutGet} \text{ for } l \\
\end{align*}
\]

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:

\[
\begin{align*}
v &= l.get (l.create v) & \text{by } \text{CreateGet} \text{ for } l \\
  &= l.get (l.create v') & \text{by assumption} \\
  &= v' & \text{by } \text{CreateGet} \text{ for } l \\
\end{align*}
\]

That is, \( v = v' \). We conclude that \( l.create \) is injective. \(\square\)

\(^1\)Note that the surjectivity of \((\text{uncurry } l.put)\) 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 \( f \in S \rightarrow S \) there exists a \( v \in V \) such that the partially-applied function \((l.put v)\) and \( f \) behave the same.
Next we prove that every lens obeys the PUTTWNCE law.

3.2.2 Lemma: Every lens \( l \in S \iff V \) obeys PUTTWNCE for every \( v \in V \) and \( s \in S \).

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

\[
\begin{align*}
    l.\text{put} \ v \ (l.\text{put} \ v \ s) \\
    = l.\text{put} \ (l.\text{get} \ (l.\text{put} \ v \ s)) \ (l.\text{put} \ v \ s) \quad \text{by PUTGET for } l \\
    = l.\text{put} \ v \ s \quad \text{by GETPUT for } l
\end{align*}
\]

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.

3.2.3 Lemma: For every total and surjective function \( g \in S \to V \) there exists a lens \( l \in S \iff V \) such that \( (l.\text{get} \ s) = (g \ s) \) for every \( s \) in \( S \).

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

\[
\begin{align*}
    l.\text{get} \ s & = g s \\
    l.\text{put} \ v \ s & = \begin{cases} s & \text{if } v = g s \\ \hat{v} & \text{otherwise} \end{cases} \\
    l.\text{create} \ v & = \hat{v}
\end{align*}
\]

By the definition of \( l.\text{get} \), we immediately have \( (l.\text{get} \ s) = (g \ s) \) for every \( s \) in \( S \). We show that \( l \) is a well-behaved lens by proving each of the lens laws directly:

▶ GetPut: Let \( s \in S \). We calculate as follows

\[
\begin{align*}
    l.\text{put} \ (l.\text{get} \ s) \ s \\
    = l.\text{put} \ (g \ s) \ s \quad \text{by definition } l.\text{get} \\
    = s \quad \text{by definition } l.\text{put}
\end{align*}
\]

and obtain the required equality.
**PutGet:** Let \( v \in V \) and \( s \in S \). We calculate as follows

\[
\begin{aligned}
l.\text{get} (l.\text{put} v s) &= \left\{ \\
&= l.\text{get} s \text{ if } v = g s \text{ by definition } l.\text{put} \\
&= l.\text{get} \hat{v} \text{ otherwise} \\
&= g s \text{ if } v = g s \text{ by definition } l.\text{get} \\
&= g \hat{v} \text{ otherwise} \\
&= v \text{ as either } v = g s \text{ (by assumption) or } g \hat{v} = v \text{ (by definition } \hat{v})
\end{aligned}
\]

and obtain the required equality.

**CreateGet:** Similar to the case for PutGet.

We can prove a similar completeness result in the other direction as well.

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

**Proof:** Let \( p \in V \to S \to S \) be a total and semi-injective function such that \((\text{uncurry } p)\) is surjective and \((p v (p v s)) = (p v s)\) for every \( v \in V \) and \( s \in S \). Also let \( s \in S \) be a source. We will prove that for every source \( s \in S \) there is a unique view \( v \) such that \((p v s) = s\). We first demonstrate that there is at least one such \( v \). By the assumption that \((\text{uncurry } p)\) is surjective, there exists a view \( v \in V \) and source \( s' \in S \) such that \((p v s') = s\). We calculate as follows

\[
\begin{aligned}
p v s &= p v (p v s') \text{ by } (p v s') = s \\
&= p v s' \text{ by } p v (p v s') = (p v s') \\
&= s \text{ by } (p v s') = s
\end{aligned}
\]

and obtain the desired equality. Next let \( v' \in V \) be a view \( v' \in V \) satisfying \( p v' s = s \). As \( p \) is semi-injective, we have \( v' = v \). Hence, \( v \) is unique.

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

\[
\begin{aligned}
\text{get } s &= \hat{s} \\
\text{put } v s &= p v s \\
\text{create } v s &= p v (\text{representative}(S))
\end{aligned}
\]

27
By construction, we immediately have $l.\text{put } v s = p \text{ v s}$ for every $v \in V$ and $s \in S$. We prove that $l$ is a well-behaved lens by showing each of the lens laws directly:

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

$$l.\text{put} (l.\text{get } s) s$$

$$= p \hat{s} s \quad \text{by definition } l.\text{get} \text{ and } l.\text{put}$$

$$= s \quad \text{by definition } \hat{s}$$

and obtain the required equality.

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

$$l.\text{get} (\text{put } v s)$$

$$= p v s \quad \text{by definition } l.\text{get} \text{ and } l.\text{put}$$

$$= s'$$

where $s' = p v s$

$$= v' \quad \text{by definition } \cdot$$

where $p v' s' = s'$

$$= v \quad \text{by semi-injectivity of } p \text{ and } (p v s) = s'$$

and obtain the required equality.

**CreateGet:** Similar to the case for PutGet. \[\square\]

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.

**3.2.5 Lemma:** For every pair of lenses $l_1 \in S \iff V$ and $l_2 \in S \iff V$ if $(l_1.\text{put } v s) = (l_2.\text{put } v s)$ for every $v \in V$ and $s \in S$ then $(l_1.\text{get } s) = (l_2.\text{get } s)$ for every $s \in S$.

**Proof:** Let $l_1 \in S \iff V$ and $l_2 \in S \iff V$ be lenses such that $(l_1.\text{put } v s) = (l_2.\text{put } v s)$ for every view $v \in V$ and source $s \in S$. Also let $s \in S$ be a source. We calculate as follows

$$l_1.\text{get } s$$

$$= l_2.\text{get} (l_2.\text{put} (l_1.\text{get } s) s) \quad \text{by PutGet for } l_2$$

$$= l_2.\text{get} (l_1.\text{put} (l_1.\text{get } s) s) \quad \text{as } l_1.\text{put} = l_2.\text{put}$$

$$= l_2.\text{get } s \quad \text{by GetPut for } l_1$$

and obtain the required equality. \[\square\]
Next, we show that the create component of every lens is consistent with its put component.

3.2.6 Lemma: For every lens \( l \in S \leftrightarrow V \) and view \( v \in V \) there exists a source \( s \in S \) such that \( (l.create\ v) = (l.put\ v\ s) \).

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

\[
\begin{align*}
  l.put\ v\ (l.create\ v) \\
  = l.put\ (l.get\ (l.create\ v))\ (l.create\ v) & \text{ by CREATEGET for } l \\
  = l.create\ v & \text{ by GETPUT for } l
\end{align*}
\]

and obtain the required equality, with \( l.create\ v \) as the source \( s \in S \).

This last lemma suggests that we can merge the behaviors of the create and put functions into a single function \( put' \in V \rightarrow S \ \text{option} \rightarrow S \) where the optional \( S \) indicates whether we are doing a put or a create. Using Lemmas 3.2.4 and 3.2.5 (generalized appropriately with \( put' \) functions) we obtain an alternate characterization of the semantics of lenses just in terms of \( put' \) functions. Thus, although lens programmers feel like they are writing the forward transformation (because the names of our primitives connote the forward transformation) and getting the backward transformation “for free”, it is actually the opposite: they are writing the backward transformation and getting the forward transformation for free.

However, while \( put' \) functions suffice for a semantic characterization of lenses, the construction of get functions in Lemma 3.2.4 is not effective—in general, it requires finding a \( v \in V \) satisfying \( (put\ v\ s) = s \). So, in the next section, when we present the primitives of a simple language of string lenses, we will revert to the original definition of lenses and define the get, put, and create functions for each primitive explicitly.

3.3 Combinators

With the semantic foundations of basic lenses in place, we now turn our attention to syntax. From this point on, we fix an alphabet \( \Sigma \) and take the universe \( \mathcal{U} \) of objects to be the set \( \Sigma^* \) of strings over \( \Sigma \). Why strings and not richer structures such as trees or complex values? There are several reasons. First, although strings are simple, they crisply expose many foundational issues, including all of the complications having to do with ordered data—see Chapter 5. Second,
there is a lot of string data in the world and it is convenient to have tools for computing views over these strings directly, without having to first parse them into other formats. Third, programmers are already familiar with standard string transformation languages (based on regular expressions).

**Atomic Lenses**

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

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

$$
\begin{align*}
E & \in \mathcal{R} \\
\text{copy } E & \in \llbracket E \rrbracket \leftrightarrow \llbracket E \rrbracket \\
\text{get } e & = e \\
\text{put } e' e & = e' \\
\text{create } e & = e
\end{align*}
$$

In the *get* direction, (copy $E$) copies every string in (the language denoted by) $E$ from source to view. Likewise, in the *put* direction, it copies the view and ignores its source argument. This behavior makes intuitive sense: since the view was obtained by copying the source 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 $\llbracket E \rrbracket$ to $\llbracket E \rrbracket$. As this is our first lens, we prove it explicitly. We include analogous well-behavedness proofs for each of the lens primitives described in the rest of this dissertation. However, since these proofs are largely calculational we defer them to the Appendix.

**3.3.1 Lemma:** Let $E \in \mathcal{R}$ be a regular expression. Then (copy $E$) is a basic lens in $\llbracket E \rrbracket \leftrightarrow \llbracket E \rrbracket$.

**Proof:**
\textbf{GetPut:} Let \( e \) be a string in \([E]\). We calculate as follows
\[
\begin{align*}
(copy\ E).put\ ((copy\ E).get\ e)\ e
=\ (copy\ E).put\ e\ e & \quad \text{by definition}\ (copy\ E).get \\
=\ e & \quad \text{by definition}\ (copy\ E).put 
\end{align*}
\]
and obtain the required result.

\textbf{PutGet:} Let \( e \) and \( e' \) be strings in \([E]\). We calculate as follows
\[
\begin{align*}
(copy\ E).get\ ((copy\ E).put\ e'\ e)
=\ (copy\ E).get\ e' & \quad \text{by definition}\ (copy\ E).put \\
=\ e' & \quad \text{by definition}\ (copy\ E).get 
\end{align*}
\]
and obtain the required result.

\textbf{CreateGet:} Let \( e \) be a string in \([E]\). We calculate as follows
\[
\begin{align*}
(copy\ E).get\ ((copy\ E).create\ e)
=\ (copy\ E).get\ e & \quad \text{by definition}\ (copy\ E).create \\
=\ e & \quad \text{by definition}\ (copy\ E).get 
\end{align*}
\]
and obtain the required result, which completes the proof. \(\square\)

By inspection, \(copy\) is oblivious (and therefore very well behaved).

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

\begin{center}
\begin{table}[h]
\begin{tabular}{c}
\hline
\(E \in \mathcal{R}\) & \([E] \neq \{\}\) & \(u \in \Sigma^*\) \\
\hline
\end{tabular}
\end{table}
\end{center}

\begin{align*}
\text{get } e &= u \\
\text{put } u\ e &= e \\
\text{create } u &= \text{representative}(E)
\end{align*}
3.3.2 Lemma: Let $E \in R$ be a regular expression and $u \in \Sigma^*$ a string. Then $(\text{const } E u)$ is a basic lens in $[E] \leftrightarrow \{u\}$.

The constant lens is not oblivious, but it is very well behaved. This is easy to see since it propagates its entire source argument in the put direction. We will often write $E \leftrightarrow u$ instead of $(\text{const } E u)$, especially in examples.

Several lenses can be expressed as derived forms using constant:

$$
\begin{align*}
\text{del } E & \in [E] \leftrightarrow \{\epsilon\} \\
\text{del } E & \triangleq E \leftrightarrow \epsilon \\
\text{ins } u & \in \{\epsilon\} \leftrightarrow \{u\} \\
\text{ins } u & \triangleq \epsilon \leftrightarrow u
\end{align*}
$$

These lenses are used in the composers lens from Chapter 1 to remove parts of the source (e.g., the (del ALPHA) lens used to remove the nationality of each composer) and to add fixed structure to the view (e.g., the (ins "", ") lens used to separate the name of each composer from their dates in the view). The get component of (del $E$) matches 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.

Default The create component of the const lens we just described produces an arbitrary element of the source type. This is more convenient than always having to specify the default string to use for create. In many applications, however, the choices of these defaults is important. The default lens gives programmers a way to control these choices. It takes as arguments a lens $l \in S \leftrightarrow V$ and a total function $f \in V \rightarrow S$. It overrides the create component of $l$ with a call to put, using $f$ to manufacture the source argument from the view.

$$
\begin{align*}
l \in S & \leftrightarrow V \\
f \in V & \rightarrow S
\end{align*}
\quad
\frac{\text{default } l \ f \in S \leftrightarrow V}{\text{get } s = l.\text{get } s} \\
\quad
\frac{\text{put } v \ s = l.\text{put } v \ s}{\text{create } v = l.\text{put } v (f \ v)}
$$

3.3.3 Lemma: Let $l \in S \leftrightarrow V$ be a basic lens and $f \in V \rightarrow S$ a function from $S$ to $V$. Then default $l \ f$ is a basic lens in $S \leftrightarrow V$. 

32
The default lens can be used, for example, to change the behavior of const. Suppose that we are using the const lens to erase a date string from the source:

```ocaml
let DATE : regexp = DIGIT{4} . "-" . DIGIT{2} . "-" . DIGIT{2}
let l : lens = DATE <-> ""
test (l.get "2009-12-06") = ""
```

The line that starts with test is a unit test—mechanically checked in our implementation—which veriæes that the expressions on the left and right sides of the equal sign evaluate to the same value. We will use these unit tests throughout this dissertation to illustrate the behavior of lenses on simple examples. The standard create function for l produces a representative string generated from the DATE regular expression:

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

In some applications, this string may not be what we want to use as a default. Using the default operator, we can override it with a better choice—e.g., the start of the Unix epoch

```ocaml
let l' : lens = default l (fun (x:string) -> "1970-01-01")
test l'.create "" = "1970-01-01"
```

or the current date (fetched by calling out to an external command):

```ocaml
let l'' : lens = default l (fun (x:string) -> Sys.exec "date +%Y-%m-%d | tr -d '\n'")
test l''.create "" = "2009-12-06"
```

By Lemma 3.2.6, we also have a kind of completeness result for default. Because the create component must be consistent with its put function, we can use default to equip a lens with every computable create function.

**Regular Operators**

Now we turn from simple operators to lens combinators that build bigger lenses out of smaller ones. The next few combinators are based on the regular operators—union, concatenation, and Kleene star. Although the details of these combinators are somewhat tailored to strings, the same essential issues arise in the conditional, product, and iteration operators of lenses for richer structures such as trees—c.f., the original paper on lenses (Foster et al., 2007b).

**Concatenation** Let us start with the concatenation operator, which is simplest:
3.3.4 Lemma: Let $l_1 \in S_1 \iff V_1$ and $l_2 \in S_2 \iff V_2$ 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 $(S_1 \cdot S_2) \iff (V_1 \cdot V_2)$.

In the get direction, the concatenation lens splits the source string into two smaller strings $s_1$ and $s_2$ belonging to $S_1$ and $S_2$, applies the get components of $l_1$ and $l_2$ to these smaller strings, and concatenates the resulting views. The put and create functions are similar. To lighten the notation, we write $(s_1 \cdot s_2)$ in the box above to indicate that $s_1$ and $s_2$ are strings belonging to $S_1$ and $S_2$ that concatenate to $(s_1 \cdot s_2)$ rather than making the split function explicit. We adopt this convention silently in the rest of this dissertation.

The typing rule for concatenation requires that the source and view types each be unambiguously concatenable. These conditions ensure two important properties:

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

2. that the lens obeys the round-tripping laws.

To see what would go wrong if we omitted them, consider the (ill-typed) transformation $l_{\text{ambig}}$:

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

(we assume that “$\leftrightarrow$” binds tighter than “$|$”, which is the union combinator defined below). Unfortunately, get is not a function since, according to the above specification, $l_{\text{ambig}}.\text{get} \; \text{aaa} = ab$ if we split $\text{aaa}$ into $a$ and $aa$, but also $l_{\text{ambig}}.\text{get} \; \text{aaa} = aab$ if we split it into $aa$ and $a$ instead. We could avoid this issue by giving the programmer a way to specify a policy for choosing among the multiple parses of ambiguous source strings—e.g., using a shortest match heuristic we would split $(\text{aaa})$ into $a$ and $aa$ so $(l_{\text{ambig}}.\text{get} \; \text{aaa})$ would only ever produce $ab$ as a result. But it is not straightforward to ensure the lens laws when heuristics are used to split...
the source and view. Intuitively, just because we split the source string using one policy does not mean we can use the same policy for the view. As an example, consider the (ill-behaved) lens \(l_{\text{bogus}}\) defined as \((k \cdot k)\) where:

\[
k \triangleq (a \leftrightarrow bb \mid aa \leftrightarrow a \mid b \leftrightarrow b \mid ba \leftrightarrow ba)
\]

Using the shortest match policy for the source we have:

\[
l_{\text{bogus}}.\text{get} \ aaa = (k.\text{get} \ a) \cdot (k.\text{get} \ aa)
\]

\[
= bb \cdot a
\]

\[
= bba
\]

If we also used the shortest match policy to split the view, we would obtain

\[
l_{\text{bogus}}.\text{put} \ bba \ aaa = (k.\text{put} \ b \ a) \cdot (k.\text{put} \ ba \ aa)
\]

\[
= b \cdot ba
\]

\[
= bba
\]

\[\neq aaa\]

as a result, which violates the GETPUT law because the new source, bba, and the original source, aaa, are different. There are similar counterexamples to the PUTGET law.

It turns out that the condition that the source types be unambiguously concatenable is essential for soundness, but the condition on the view types is not. The key observation is that we can allow the concatenation of the view types to be ambiguous if we replace \(\text{put}\) 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 way of splitting the view if so.\(^2\) This \(\text{put}\) 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. As an example, on the same strings as above, the \(\text{put}\) of \(l_{\text{bogus}}\) refined in this way would behave as follows:

\[
l_{\text{bogus}}.\text{put} \ bba \ aaa = (k.\text{put} \ bb \ a) \cdot (k.\text{put} \ a \ aa)
\]

\[
= a \cdot aa
\]

\[
= aaa
\]

\(^2\)In more detail, here is how it would work, as originally proposed by Julien Cretin. Call the position of a split in the set of all possible splits (ordered lexicographically using the length of the strings in each split) its \textit{index}. As examples, the index of the split obtained using the shortest match is 0 and the index of the split obtained using the longest match is equal to the size of the set of possible splits. Revise the \(\text{put}\) function so that the view is split by the index of \((l_1.\text{get} \ s_1, l_2.\text{get} \ s_2)\) if the index corresponds to a valid split, and by any other heuristic if not.
The same technique can not be used on the source side. The \( \text{get} \) function only takes a single argument, so we have no way to force it to split the source in the same way as arbitrary strings generated by \( l_1 \) and \( l_2 \)'s \( \text{put} \) 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:

\[
l_{\text{bogus}} \cdot \text{get} \cdot (l_{\text{bogus}} \cdot \text{put} \cdot \text{babb} \cdot \text{baa}) = l_{\text{bogus}} \cdot \text{get} \cdot ((k \cdot \text{put} \cdot \text{ba} \cdot \text{b}) \cdot (k \cdot \text{put} \cdot \text{bb} \cdot \text{aa}))
\]

\[
= l_{\text{bogus}} \cdot \text{get} \cdot (\text{ba} \cdot \text{a})
\]

\[
= l_{\text{bogus}} \cdot \text{get} \cdot (\text{ba})
\]

\[
= (k \cdot \text{get} \cdot \text{b}) \cdot (k \cdot \text{get} \cdot \text{aa})
\]

\[
= (\text{b} \cdot \text{a})
\]

\[
= \text{ba}
\]

\[
\neq \text{babb}
\]

The \( \text{get} \) function has no information to use to determine that \( \text{bba} \) needs be split into \((\text{ba} \cdot \text{a})\) to satisfy \( \text{PutGet} \) so it blindly uses shortest match, yielding 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 \( l_1 \)'s \( \text{put} \) to be passed to \( l_2 \)'s \( \text{get} \), or vice versa.

In fact, although it is interesting to see that we can allow the concatenation of the view types to be ambiguous, we do not allow it. One reason is that, in our experience, programmers are not very good at tracking subtle forms of ambiguity. Most of the ambiguous concatenations we have encountered in real-world examples turned out to be bugs. Another reason is that, as discussed in Section 3.1, the \( \text{GetPut} \) law is a rather weak condition, intended more as a loose guide than as a complete specification of correctness. Using the source string to decide how to split the view when we must to satisfy \( \text{GetPut} \), but splitting it in a completely different way in all other cases violates the spirit of the law—that \( \text{put} \) function should respect the integrity of hidden source data. The strict version of the lens (i.e., with a typing rule that requires that the concatenation of the view types be ambiguous) produces a very well behaved lens when applied to very well behaved arguments, and is “no less” very well behaved than its arguments when it is applied to lenses that are merely well behaved. The relaxed version, however, does not even preserve very well behavedness because the \( \text{put} \) function does not always preserve the
information from the source used to split the view. This means that making a small edit to the view can cause it to be split in radically a different way. We believe that programmers would find this behavior unintuitive. A final reason for preferring the strict version is that it simplifies several extensions of basic lenses—e.g., quotient, resourceful, and secure variants—that we consider later.

**Union** The next combinator is a conditional operator on lenses:

\[
\begin{align*}
S_1 \cap S_2 &= \emptyset \\
l_1 \in S_1 &\iff V_1 \\
l_2 \in S_2 &\iff V_2 \\
(l_1 | l_2) &\in (S_1 \cup S_2) \iff (V_1 \cup V_2)
\end{align*}
\]

\[
\begin{align*}
\text{get } s &= \begin{cases} 
  l_1.\text{get } s & \text{if } s \in S_1 \\
  l_2.\text{get } s & \text{if } s \in S_2
\end{cases} \\
\text{put } v \ s &= \begin{cases} 
  l_1.\text{put } v \ s & \text{if } v \in V_1 \text{ and } s \in S_1 \\
  l_2.\text{put } v \ s & \text{if } v \in V_2 \text{ and } s \in S_2 \\
  l_1.\text{create } v & \text{if } v \in (V_1 - V_2) \text{ and } s \in S_2 \\
  l_2.\text{create } v & \text{if } v \in (V_2 - V_1) \text{ and } s \in S_1
\end{cases} \\
\text{create } v &= \begin{cases} 
  l_1.\text{create } v & \text{if } v \in V_1 \\
  l_2.\text{create } v & \text{if } v \in (V_2 - V_1)
\end{cases}
\]

**3.3.5 Lemma:** Let \(l_1 \in S_1 \iff V_1\) and \(l_2 \in S_2 \iff V_2\) be basic lenses such that \((S_1 \cap S_2) = \emptyset\). Then \((l_1 \mid l_2)\) is a basic lens in \((S_1 \cup S_2) \iff (V_1 \cup V_2)\).

Like conditionals in ordinary programming languages, the union lens selects a branch 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 \(V_1\) and \(V_2\) to overlap. It tries to select one of \(l_1\) or \(l_2\) using the view types, and falls back to the source argument to disambiguate in cases where the view belongs to both \(V_1\) and \(V_2\). The create function is similar, but just uses \(l_1\) in cases where the view belongs to both \(V_1\) and \(V_2\) (as it has no source argument to use). This is an arbitrary choice, but not a limitation: to use \(l_2\) instead, the programmer can just write \((l_2 \mid l_1)\). It does mean, however, that union is not commutative.
Because *put* is a total function, it needs to handle situations where the view comes from one side of the union (e.g., \((V_1 - V_2)\)) and the source comes from the other side (e.g., \(S_2\)). 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 a function from \(l_1\). In this basic lens version of union, we simply discard the source and use *create*. However, this is not the only option—although the source \(s\) belongs to \(S_2\), it might contain information that could be represented in \(S_1\). In some situations, we would like *put* to reintegrate this information—in some way—with the new source. For example, consider putting \(\lambda\) into \(3\) using the lens \(l\) defined as follows:

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

We might like the lens to propagate the \(\lambda\) from the view and restore the \(3\) from the source, but \(l\)'s *put* invokes the *create* function from \(l_1\), which yields \(A0\):

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

In the original paper on basic lenses (Foster et al., 2007b), we described a union combinator with “fixup” functions—mappings from \(S_2\) to \(S_1\) and vice versa. The idea was that these functions would be provided by programmers and used by the *put* function to extract information from sources on one side of the union for use with views on the other side. Semantically fixup functions are exactly what is needed—one can show that the union lens described in that paper is most general. But syntactically they are a nightmare—the programmer has to write down two additional total functions on the source types! We refrain from introducing fixup functions here because resourceful lenses offer a mechanism for passing information from one side of a union to another that seems to balance the semantic and syntactic tradeoffs between these two extremes nicely—see Chapter 5.

By analogy with concatenation, one might wonder why we allow the view types to overlap in the union lens. One reason for making this choice is that, in our experience, programmers are much more comfortable reasoning about disjointness than they are reasoning about about ambiguous concatenation. Another reason is that we have many examples that demand both forms of union—overlapping and disjoint. Moreover, it turns out that the main semantic argument made in the discussion of concatenation does not apply to union: disjoint unions do not preserve very well behavedness. In fact, editing the view from one side of the union to the other causes the union to discard the entire source, violating PUTPUT:
let l1 : lens = copy [a-m] . del [0-4]
let l2 : lens = copy [n-z] . del [5-9]
let l : lens = l1 | l2

In general, union only preserves very well behavedness when the view types of the sublenses are identical, making it impossible to edit the view from one side of the union to the other, or when both branches are oblivious so there is no hidden source data to preserve.

**Kleene Star** The last regular operator, Kleene star, iterates a lens. It combines the behavior of concatenation and union:

\[
\begin{align*}
\frac{l \in S}{l^* \in S^*} & \iff V^* \\
(l^* \in S^*) & \iff V^*
\end{align*}
\]

**Lemma:** Let \(l \in S \iff V\) be a basic lens such that \(S^*\) and \(V^*\). Then \(l^*\) is a basic lens in \(S^* \iff V^*\).

In the \(\text{get}\) direction, it takes the source string, splits it (unambiguously) into a list of substrings belonging to the source type of the iterated lens \(l\), applies the \(\text{get}\) component to each string in this list, and concatenates the results. The \(\text{put}\) and \(\text{create}\) functions are similar. Note, however, that the \(\text{put}\) function has to handle situations where the number of substrings of the view is not equal to the number of substrings of the source. To satisfy PutGet, it must produce a source with the same number of substrings as its view argument has substrings. When there are more source substrings than view substrings, it simply discards the extras. When there are more view substrings than source substrings, it uses \(l.\text{create}\) to process the extras.

Because it sometimes discards substrings of the source, lenses written using Kleene star are not always very well behaved, as illustrated in the following example:

```
let 1 = copy [a-z] . del [0-9]
test 1*.put "xyz" into (1*.put "xy" into "a1b2c3") = "x1y2z0"
test 1*.put "xyz" into "a1b2c3" = "x1y2z3"
```
This is unfortunate, but unavoidable: we either have to allow the put component of Kleene star to discard some of the information in the source, sacrificing very well behavedness, or we have to restrict put to only accept views having the same number of 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 are powerful enough to express a large collection of useful transformations. Some applications, however, require just a little more power. It is not difficult to extend our set of combinators with additional primitives—we only require that their types be given by regular languages, to ensure that typechecking remains decidable. In this section, we present a few of the primitives that we have found especially useful in building applications using basic lenses.

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

\[
\begin{align*}
    l_1 \in S & \iff U \\
    l_2 \in U & \iff V \\
    (l_1;l_2) \in S & \iff V \\
    \text{get } s & = l_2.\text{get } (l_1.\text{get } s) \\
    \text{put } v s & = l_1.\text{put } (l_2.\text{put } v (l_1.\text{get } s)) \\
    \text{create } v & = l_1.\text{create } (l_2.\text{create } v)
\end{align*}
\]

3.3.7 Lemma: Let \( l_1 \in S \iff U \) and \( l_2 \in U \iff V \) be basic lenses. Then \((l_2;l_2)\) is a basic lens in \( S \iff V \).

The get component of the composition lens processes the source string in two phases, using 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 strings flowing through the composition operator have the correct type for each application.
As an example illustrating how composition can be useful, recall the composers lens from Chapter 1. It requires that the name of each composer be a string belonging to the set of strings described by the regular expression `ALPHA`. Suppose that we wanted to extend it so that the name could be an arbitrary string. It is not difficult to do this, but we need to be careful to respect the escaping conventions of the XML and ASCII formats, escaping ‘&amp;’, ‘<’, ‘>’, etc. on the XML side and ‘,’ ‘\n’, and ‘\’ on the ASCII side. If we already have lenses `xml_unesc` and `csv_esc` that handle unescaping for XML and escaping for ASCII (e.g., from a library), is simpler to just compose these lenses than it is to write an end-to-end escaping lens from scratch:

```coq
let composer : lens =
  xml_elt "composer"
  ( xml_elt "name" ( xml_unesc ; csv_esc )
    . ins (COMMA . SPACE)
    . xml_elt "lived" (copy (YEAR . DASH . YEAR) )
    . xml_elt "nationality" ( default (del ALPHA) (fun (x:string) -> "Unknown")) )
```

This new version of the composers lens maps XML sources...

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

...to ASCII views

*Duke Ellington & His Orchestra, 1899-1974*

transforming the name of the composer according to the escaping conventions for each format.

**Filter**  The next lens, *filter* takes as arguments two regular expressions $E$ and $F$ and yields a lens that filters the $F$s away from a list of $E$s and $F$s. Its *get* and *put* functions are defined in terms of helper functions *string_filter* and *string_unfilter*, which are shown in Figure 3.2.

$$
\text{let } \text{get } s = \text{string}_{\text{filter}} E s \\
\text{let } \text{put } v s = \text{string}_{\text{unfilter}} F v s \\
\text{let } \text{create } v = v
$$

**3.3.8 Lemma:** Let $E$ and $F$ be regular expressions satisfying $(\emptyset \cap [F]) = \emptyset$ and $(\emptyset \cup [F])^*$. Then $\text{filter } E F$ is a basic lens in $(E \cup F)^* \equiv E^*$.
let rec string_filter S xs = match xs with
  | ϵ → ϵ
  | x·xs' →
    let xs'' = string_filter S xs' in
    if x ∈ S then x·xs'' else xs''

let rec string_unfilter T ys xs = match ys, xs with
  | ϵ, _ → string_filter T xs
  | y·ys', x·xs' →
    if x ∈ T then x·(string_unfilter ys xs')
    else y·(string_unfilter ys' xs')

Figure 3.2: Pseudocode for string_filter and string_unfilter.

It is tempting to define filter as (copy E | 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 extra Fs in the source while string_unfilter’s put always restores all of the Fs in its second argument.

As an example illustrating the use of filter, suppose that we wanted to define a lens that computes a view containing composers born in the 20th century. To achieve this, we can revise the lens using filter as follows:

let YEAR_20c = "19" . DIGIT{2} in
let composer (birth:regexp) : lens =
  xml_elt "composer"
    ( xml_elt "name" (copy (ALPHA . SPACE . ALPHA))
      . ins (COMMA . SPACE)
      . xml_elt "lived" (copy (birth . DASH . YEAR))
      . xml_elt "nationality" (default (del ALPHA) (fun (x:string) → "Unknown")))
let composer_other : lens = composer (YEAR - YEAR_20c)
let composer_20c : lens = composer YEAR_20c
let composers : lens =
  xml_elt "composers"
    ( filter (stype composer_20c) (stype composer_other);
      ( copy EPSILON | composer_20c . (ins NEWLINE . composer_20c)* ) )

In this definition, we have parameterized the composer lens by a regular expression for the birth year, and instantiated it twice: the composer_20c lens handles composers born in the 20th century, and composer_other handles composers born in any other century. The main composers lens, first filters away non-20th century composers (the function stype extracts the regular expression representing the source type of a lens) and then processes the remaining composers.
using composer_20c. On the original XML source, this lens computes the view:

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

where Sibelius, born in 1865, has been filtered away. Note that the filter lens always restores the filtered composers from the source. For example, here is the source we get by putting the empty list of composers into the original source:

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

It is not difficult to show that \((\text{filter } E \ F)\) is a very well behaved lens.

Swap All of the lenses we have described so far have get functions that are expressible as one-way finite state transducers (Berstel, 1979). This class of transformations is powerful enough to express a large collection of examples, but it has a fundamental limitation: the restriction to finite state means that it is impossible to “remember” arbitrary amounts of data. This means that we cannot write a variant of the composers lens where the order of the name and dates are swapped in the view. Fortunately, lifting this restriction poses no semantic problems. The swap lens is like concatenation but inverts the order of strings in the view:

\[
\begin{array}{c|c|c}
S_1 \cdot S_2 & V_2 \cdot V_1 \\
\hline
l_1 \in S_1 & l_2 \in S_2 & l_1 \sim l_2 \in (S_1 \cdot S_2) \iff (V_1 \cdot V_2)
\end{array}
\]

\[
\begin{array}{c|c}
\text{get} (s_1 \cdot s_2) & = (l_2 \cdot \text{get} s_2) \cdot (l_1 \cdot \text{get} s_1) \\
\text{put} (v_2 \cdot v_1) & = (l_1 \cdot \text{put} v_1 s_1) \cdot (l_2 \cdot \text{put} v_2 s_2) \\
\text{create} (v_2 \cdot v_1) & = (l_1 \cdot \text{create} v_1) \cdot (l_2 \cdot \text{create} v_2)
\end{array}
\]

3.3.9 Lemma: Let \(l_1 \in S_1 \iff V_1\) and \(l_2 \in S_2 \iff V_2\) 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 \((S_1 \cdot S_2) \iff (V_1 \cdot V_2)\).

Like concatenation, the get component of swap splits the source into strings belonging to \(S_1\) and \(S_2\), applies the get component of \(l_1\) and \(l_2\) to each string. However, before it concatenates the results, it swaps them, putting the results in the opposite order. The put function is similar: it splits the view into strings belonging to \(V_2\) and \(V_1\) and the source into strings belonging to \(S_1\)
and $S_2$, swaps the order of the view strings, applies the put component of $l_1$ and $l_2$ to each, and concatenates the results.

Let us implement a variant of the composers lens described above where the name and dates of each composer are swapped in the view:

```ml
let composer : lens =
  xml_elt "composer"
  ( ( xml_elt "name" (copy (ALPHA . SPACE . ALPHA) )
      - ( xml_elt "lived" (copy (YEAR . DASH . YEAR) )
          . ins (COMMA . SPACE) )
      . xml_elt "nationality" ( default (del ALPHA) (fun (x:string) -> "Unknown") ) )
  )
```

Compared to the previous version of the lens, we had to make two changes. First, we moved the lens $(\text{ins (COMMA . SPACE)})$ down so that it follows the lens for the $\text{lived}$ element. We also replaced the concatenation (.) between the $\text{name}$ and $\text{lived}$ elements with swap ($\sim$). On the original XML source, this lens computes the view

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

It turns out that the concatenation and swap lenses can both be obtained as instance of a more general combinator $\text{permute}$ that takes as arguments a permutation $\sigma$ on $\{1,\ldots,n\}$ and a list of lenses $[l_1,\ldots,l_n]$. In the get direction, it splits the source string, processes each string using the corresponding lens from the list of lenses, permutes the list of view strings, and concatenates the results. The put and create functions are similar. To obtain the concatenation $(l_1,l_2)$, we let $\sigma$ be the identity permutation on $\{1,2\}$ and the list of lenses be $[l_1,l_2]$. To obtain $(l_1 \sim l_2)$, we let $\sigma$ be the transposition on $\{1,2\}$ and again let the list of lenses be $[l_1,l_2]$.

### 3.4 Summary

Basic lenses are a natural class of well-behaved bidirectional transformations that provide the semantic foundations of bidirectional programming languages. Their design emphasizes both robustness and ease of use, guaranteeing totality and strong well-behavedness conditions, formulated as intuitive round-tripping laws. Many familiar transformations can be interpreted as basic lenses including the identity and constant functions, composition, iteration, conditionals, product, and many others. In the domain of strings, these constructs can be used to elegantly describe updatable views over many formats of practical interest.
Chapter 4

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! In the real world, most bidirectional transformations do not obey the basic lenses laws. Or rather, they obey them in spirit, but not to the letter—i.e., only “modulo unimportant details.” The nature of these details varies from one application to another: examples include whitespace, artifacts of representing richer structures (relations, trees, and graphs) as text, escaping of atomic data (XML PCDATA, vCard, and BibTeX values), ordering of fields in record-structured data (BibTeX fields, XML attributes), wrapping of long lines in ASCII formats (RIS bibliographies, UniProtKB genomic data bases), and duplicated information (aggregate values, tables of contents).

To illustrate, consider the composers lens again. The information about each composer could be larger than fits comfortably on a single line in the ASCII view. 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: as shown in Lemma 2, the PUTGET law is only satisfied when put is semi-injective. This
means that the lens must map these views, which we intuitively regard as equivalent, to different XML sources—i.e., the presence or absence of linebreaks in the view must be reflected in the source. We could build a lens that does this—e.g., storing the line break inside the PCDATA string...

```xml
<composer>
    <name>Jean Sibelius</name>
    <lived>1865-1957</lived>
    <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 requires that the PCDATA not contain 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 gracefully is the goal of this chapter.

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

1. We can be informal, stating the basic lens laws in their strict form and explaining that they “essentially hold” for our program, perhaps providing evidence in support of this claim by describing how unimportant details are processed algorithmically. In many applications, being informal is a perfectly acceptable strategy, and several bidirectional languages adopt it. For example, the biXid language, which describes XML to XML conversions using pairs of intertwined tree grammars, provides no explicit guarantees about round-trip behavior, but its designers clearly intend it to be “morally bijective” (Kawanaka and Hosoya, 2006). The PADS system is similar (Fisher and Gruber, 2005).

2. We can weaken the laws. The designers of the X language have argued that `PutGet` should be replaced with a weaker “round-trip and a half” version (Hu et al., 2004):

\[
\frac{s' = \text{put } v \ s}{\text{put } (\text{get } s') \ s' = s'}
\]  

(PutGetPut)

Their reason for advocating this law is that they want to support a duplication operator. Having duplication makes it possible to express many useful transformations—e.g.,
augmenting a document with a table of contents—but because the duplicated data is not preserved exactly on round trips (consider making a change to just one copy of the duplicated data), the \texttt{PutGet} law is not satisfied.

The weaker \texttt{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 \texttt{put} functions, lenses whose \texttt{get} component is the identity and and whose \texttt{put} component is \((\texttt{put} \, v \, s) = s\), etc.\footnote{Later work by the same authors (the journal version of Liu et al. 2007) excludes such transformations by decorating data with “edit tags,” ordering this data according to a “more edited than” relation, and adding a new law stipulating that doing a \texttt{put} followed by a \texttt{get} must yield a “more edited” view.}

3. We can divide bidirectional programs into a “core component” that is a lens in the strict sense and “canonization” phases that operate at the perimeters of the transformation, standardizing the representation of inessential data. See Figure 4.1.

For example, in our previous work on lenses for trees, the end-to-end transformations on actual strings (i.e., concrete representations of trees in the filesystem) only obey the lens laws up to the equivalence induced by a \textit{viewer}—a parser and pretty printer mapping between raw strings and more structured representations of trees (Foster et al., 2007b). 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 (Brabrand et al., 2008).\footnote{In XSugar, XML canonization is treated as a distinct “pre-processing” phase, but canonization of other ignor-}
This approach is quite workable when the data formats and canonizers are generic. However, for ad-hoc data including 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 component of the canonizer! In other words, we end up back in the situation that lenses were designed to avoid. In our experience, these difficulties quickly become unmanageable for most formats of practical interest.

4. We can develop a more refined account of the whole lens framework that allows us to say, precisely and truthfully, that the lens laws hold modulo a particular equivalence relation. This is the approach we pursue in this chapter. The main advantage over the approach using viewers, as we will see, is that it allows us to define and use canonizers anywhere in a lens program, not only at the perimeters.

This chapter is organized as follows. Section 4.1 presents the relaxed semantic space of quotient lenses. Section 4.2 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 4.3 defines quotient lens versions of the regular operators—concatenation, union, and Kleene star. Section 4.4 introduces new primitives that are possible in the relaxed space of quotient lenses. Section 4.5 discusses the issue of typechecking quotient lenses. Section 4.6 illustrates some uses of quotient lenses on a large example—a lens for converting between XML and ASCII versions of a large genomic database. We conclude in Section 4.7.

4.1 Semantics

At the semantic level, the definition of quotient lenses is a straightforward refinement of basic lenses. We enrich the types of lenses with equivalence relations—instead of the basic lens type $S \iff V$, we write $S/\sim_S \iff V/\sim_V$, where $\sim_S$ is an equivalence on $S$ and $\sim_V$ is an equivalence on $V$—and we relax the lens laws accordingly.
4.1.1 Definition [Quotient Lens]: Let $S \subseteq U$ and $V \subseteq U$ be sets of sources and views and let $\sim_S$ and $\sim_V$ be equivalence relations on $S$ and $V$. A quotient lens $l$ has components with the same types as a basic lens

\[
l.\text{get} \in S \rightarrow V \\
l.\text{put} \in V \rightarrow S \rightarrow S \\
l.\text{create} \in V \rightarrow S
\]

but is only required to obey the lens laws up to $\sim_S$ and $\sim_V$:

\[
l.\text{put} (l.\text{get} s) \sim_S s \\
l.\text{get} (l.\text{put} v s) \sim_V v \\
l.\text{get} (l.\text{create} v) \sim_V v
\]

(GetPut)   (PutGet)   (CreateGet)

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

\[
\frac{s \sim_S s'}{l.\text{get} s \sim_V l.\text{get} s'} \quad \text{(GetEquiv)}
\]

\[
\frac{v \sim_V v' \quad s \sim_S s'}{l.\text{put} v s \sim_S l.\text{put} v' s'} \quad \text{(PutEquiv)}
\]

\[
\frac{v \sim_V v'}{l.\text{create} v \sim_S l.\text{create} v'} \quad \text{(CreateEquiv)}
\]

We write $S/\sim_S \iff V/\sim_V$ for the set of 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 $V/\sim_V$, and when we pick $\sim_S$ and $\sim_V$ to be equality—the finest equivalence relation—they are equivalent to the basic laws precisely. However, although we reason about the behavior of quotient lenses as if they operated on equivalence classes, note that the component functions actually transform members of the underlying sets of sources and views—i.e, the type of $\text{get}$ is $S \rightarrow V$, not $S/\sim_S \rightarrow V/\sim_V$. The second group of laws ensures 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 composition, defined below, is sound.
4.2 Quotient Lens Combinators

So much for semantics; the story in this chapter is much more interesting on the syntactic side. We begin our exploration of syntax by describing some generic operators for constructing quotient lenses.

Lift  Intuitively, it is clear that quotient lenses generalize basic lenses. A concrete witness to this fact is the lift operator, which lifts a basic lens to a quotient lens.

\[
\begin{align*}
\text{lift } l & \in S \leftrightarrow V \\
\text{lift } l & \in S/\Rightarrow V/\Rightarrow
\end{align*}
\]

\[
\begin{align*}
\text{get } s & = l.\text{get } s \\
\text{put } v & = l.\text{put } v, s \\
\text{create } v & = l.\text{create } v
\end{align*}
\]

**4.2.1 Lemma:** Let \( l \in S \leftrightarrow V \) be a basic lens. Then \( \text{lift } l \) is a quotient lens in \( S/\Rightarrow \leftrightarrow V/\Rightarrow \).

The get, put, and create components of \( \text{lift } l \) are identical to those of \( l \). The equivalence relations on the source and view are the finest equivalence relation—equality. The proof that \( (\text{lift } l) \) is a well-behaved quotient lens is immediate using the basic lens laws for \( l \).

Left and Right Quotient  The next two combinators provide mechanisms for loosening up a lens, widening the set of structures they can handle, and coarsening their equivalence relations in a corresponding way.

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., \( l \) might 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 construct a new quotient lens whose domain is a larger set \( S \)—e.g., the same set of strings, except that with arbitrary amounts of whitespace may appear in various places—with an equivalence \( \sim_S \) that relates pairs of strings differing only in whitespace.

To get back and forth between \( S \) and \( U \), we need two functions. The first function, called canonize, maps every element of \( S \) to its “canonical representative” 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 obtained by adding whitespace according to a layout convention. The \textit{canonize} and \textit{choose} functions together are called a \textit{canonizer}—see Figure 4.2.

Clearly, a canonizer is a bit like a lens (minus the \textit{put} component). The key difference is that canonizers only have to obey one round-tripping law.

4.2.2 Definition [Canonizer]: Let $S \subseteq U$ and $U \subseteq U$ be sets of objects and let $\sim_U$ be an equivalence relation on $U$.

A canonizer $q$ from $S$ to $U/\sim_U$ comprises two functions

$$q.\text{canonize} \in S \rightarrow U$$
$$q.\text{choose} \in U \rightarrow S$$

obeying

$$q.\text{canonize} \ (q.\text{choose} \ u) \sim_U u \quad \text{(ReCANONIZE)}$$

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

Having described how quotienting works at a high level, we now give precise definitions. Lenses can be quotiented in two ways—on the left, treating part of the source as ignorable, or on the right, treating part of view as ignorable.

**Left Quotient** The $lquot$ operator takes as arguments a canonizer $q$ and a quotient lens $l$ and produces a new quotient lens where $l$’s type is coarsened on the source side using $q$.

\footnote{We state the equivalence on $U$ explicitly because, when we put the canonizer together with a quotient lens using $lquot$ and $rquot$, we will need the equivalences on $U$ to match up. We do not mention the equivalence on $S$, however, because it will be calculated later (by the typing rules for $lquot$ and $rquot$).}
4.2.3 Lemma: Let \( q \in S \mapsto U/\sim_U \) be a canonizer and \( l \in U/\sim_U \leftrightarrow V/\sim_V \) be a quotient lens. Then \( (lquot q l) \) is a quotient lens in \( S/\sim_S \leftrightarrow V/\sim_V \) where \( s \sim_S s' \) if and only if \( q.cannonize s \sim_U q.cannonize s' \).

The \( get \) component of \( lquot \) canonizes its source argument to an element of \( U \) using \( q.cannonize \) and then maps it to a \( V \) using \( l.get \). Similarly, the \( put \) component first maps its source argument to a \( U \) using \( q.cannonize \), and then puts the view into this \( U \) using \( l.put \). It uses \( q.choose \) to produce the final source. The equivalence relation \( \sim_S \) on the source is the relation induced by \( q.cannonize \) and \( \sim_U \)—i.e., two sources are equivalent if \( q.cannonize \) maps them to equivalent elements of \( U \) (according to \( \sim_U \)).

As an example of left quotienting, recall the \( del \) lens. It deletes a string in the \( get \) direction and restores it in the \( put \) direction. In most situations, this is the behavior we want. However, if the data being deleted is “unimportant”—e.g., whitespace—we might like the \( put \) function to always produce a particular string \( e \in \mathbb{[E]} \) rather than restoring the original. This transformation is not a basic lens—it violates GETPUT—but it is easy to define as a quotient lens using \( lquot \):

```
let qdel (E:regexp) (e:string in E) : lens =
  lquot
  (cannonizer_of_lens (default (del E) (fun (x:string) -> e)))
  (copy "")
  test (qdel [A-Z] "Z").get "A" = ""
  test (qdel [A-Z] "Z").put "" into "A" = "Z"
```

(The \( cannonizer_of_lens \) operator, which builds a canonizer out of a lens, is defined below.) The \( get \) function first canonizes the source string to the empty string, and then copies the empty string into the view. The \( put \) function first copies the empty string, and then invokes the canonizer’s
choose function, which yields $e$. The type of $qdel \ E \ e$ is $[E]/\text{Tot}(E) \iff \{e\} = $, which records the fact that every source string is treated equivalently.

A common use of the $qdel$ is standardizing whitespace. For example, we can rewrite the $xml\_elt$ helper function used in the composer lens so that it always produces pretty-printed XML. Here is a version that allows the indentation before each opening and closing tag to be controlled separately:

```
let WS : regexp = WHITESPACE*
let full_xml_elt (tag:string) (ws1:string in WS) (ws2:string in WS) (body:lens) : lens =
  qdel WS ws1
  . del ("<" . tag . ">")
  . body
  . qdel WS ws2
  . del ("</" . tag . ">")
```

We can instantiate $full\_xml\_elt$ to obtain several additional useful helper functions as follows:

```
let xml elt (tag:string) (indent:string in WS) (body:lens) : lens =
  full_xml_elt tag indent indent body
let xml PCDATA elt (tag:string) (indent:string in WS) (body:lens) : lens =
  full_xml_elt tag EPSILON body
let xml outer elt (tag:string) (indent:string in WS) (body:lens) : lens =
  full_xml_elt tag EPSILON indent body
```

The $xml\_elt$ lens takes just one argument, which is used as the indentation before opening and closing tags. The $xml\_PCDATA\_elt$ function adds the specified indentation before the opening tag but none before the closing tag (e.g., because the value in the element is PCDATA). The $xml\_outer\_elt$ helper handles the outer-most element in a document. It adds no indentation before the opening tag and the specified indentation before the closing tag. Using these helper functions, we can rewrite the composer lens so that it produces pretty-printed XML as follows:

```
let INDENT0 = NEWLINE
let INDENT1 = INDENT0 . SPACE . SPACE
let INDENT2 = INDENT1 . SPACE . SPACE
let composer : lens =
  xml elt "composer" INDENT1
  ( xml PCDATA elt "name" INDENT2 (copy (ALPHA . SPACE . ALPHA) )
    . ins (COMMA . SPACE)
    . xml PCDATA elt "lived" INDENT2 (copy (YEAR . DASH . YEAR) )
    . xml PCDATA elt "nationality" INDENT2
    ( default (del ALPHA) (fun (x:string) -> "Unknown") ) )
  let composers : lens =
  xml outer elt "composers" INDENT0
  ( copy EPSILON | composer . (ins NEWLINE . composer)* )
```

In the get direction it discards the whitespace between XML elements. 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 an XML source where each element has no indentation

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

produces a pretty-printed result

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

that also reflects the update made to the view.

**Right Quotient**  The \( rquot \) operator is symmetric. It quotients a lens \( l \in S/\sim_S \iff U/\sim_U \) on the right, using a canonizer \( q \) from \( V \) to \( U/\sim_U \). An interesting difference compared to \( lquot \) is that its canonizer argument 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 “head to head” composition—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 lenses.
4.2.4 Lemma: Let \( l \in S/\sim_S \ bottled U/\sim_U \) be a quotient lens and \( q \in V \leftrightarrow U/\sim_U \) a canonizer. Then \( rquot l q \) is a quotient lens in \( S/\sim_S \leftrightarrow V/\sim_V \) where \( v \sim_V v' \) if and only if \( q\cdot canonize v \sim_U q\cdot canonize v' \).

The \( get \) function first transforms the source to an intermediate view using \( l\cdot get \), and then picks a representative using \( q\cdot choose \). Similarly, the \( put \) function canonizes the view using \( q\cdot canonize \), and then uses \( l\cdot put \) to compute the new source.

Recall the basic lens \( (\text{ins } e) \), which inserts the fixed string \( e \) into the view. The quotient lens version, \( (q\text{ins } E e) \), behaves like \( (\text{ins } e) \) in the \( get \) direction, but accepts any string in the set \( [E] \) in the \( put \) direction (where \( [E] \) must contain \( e \)). We often use \( q\text{ins} \) to insert formatting elements—e.g., whitespace—into the view. For example, if we flipped the composers example around so that the source was ASCII and the view was XML, it would be critical that we be able to do this. It is straightforward to define \( q\text{ins} \) using \( rquot \):

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

Note that the \( put \) component of \( q\text{ins} \) is not semi-injective:

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

Of course, it is semi-injective modulo the equivalence relation on the set of views.

Using \( rquot \) and \( quotient \), we can also define a quotient lens version of the constant lens:

```ocaml
let qconst (u:string) (E:string) (D:string) (v:string) : lens = 
  qdel E u . qins D v
```
This lens accepts any \( E \) in the \textit{get} direction and maps it to \( v \). In the \textit{put} direction, it accepts any \( D \) and maps it to \( u \). Its type, \([ E ]/\text{Tot}( [ E ] ) \) \( \iff \) \([ D ]/\text{Tot}( [ D ] ) \), records the fact that all strings in the source and view are treated equivalently.

\textbf{Subsumption} The \textit{lquot} and \textit{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:

\[
\begin{array}{c}
q \in V \xrightarrow{\sim} U/\sim_U \\
\sim \text{ is a refinement of } \sim_{U'}
\end{array}
\]

\[
q \in V \xleftarrow{\sim} U/\sim_{U'}
\]

\textbf{4.2.5 Lemma:} Let \( q \in V \xleftarrow{\sim} U/\sim_U \) 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 \( V \xleftarrow{\sim} U/\sim_{U'} \).

This rule allows the equivalence relation component of a canonizer’s type to be coarsened. For example, if we want to quotient a lens \( l \in U/\sim_U \xleftarrow{\sim} V/\sim_V \) on the left using a canonizer \( q \in S \xleftarrow{\sim} U/\sim_U \), we can use it to promote \( q \) to the type \( S \xleftarrow{\sim} U/\sim_U \).

\textbf{Sequential Composition} The next combinator composes quotient lenses themselves.

\[
\begin{array}{c}
l_1 \in S/\sim_S \xleftrightarrow{\sim} U/\sim_U \\
l_2 \in U/\sim_U \xleftrightarrow{\sim} V/\sim_V
\end{array}
\]

\[
l_1;l_2 \in S/\sim_S \xleftrightarrow{\sim} V/\sim_V
\]

\[
\begin{array}{c}
\text{get } s = l_2 \cdot \text{get } (l_1 \cdot \text{get } s) \\
\text{put } v \ s = l_1 \cdot \text{put } (l_2 \cdot \text{put } v (l_1 \cdot \text{get } s)) \ s \\
\text{create } v = l_1 \cdot \text{create } (l_2 \cdot \text{create } v)
\end{array}
\]

\textbf{4.2.6 Lemma:} Let \( l_1 \in S/\sim_S \xleftrightarrow{\sim} U/\sim_U \) and \( l_2 \in U/\sim_U \xleftrightarrow{\sim} V/\sim_V \) be quotient lenses. Then \( (l_2;l_2) \) is a quotient lens in \( S/\sim_S \xleftrightarrow{\sim} V/\sim_V \).

The components of \( (l_2;l_2) \) 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 4.5. To see what goes wrong if this condition is dropped, consider

\[ l_1 = \text{copy} \{a\} \in \{a\} = \iff \{a\} = \]
\[ l_2 = \text{copy} (\{a\} | \{b\}) \in \{a, b\} = \iff \{a, b\} = \]

and \( q \in \{a, b\} \iff \{a\} = \) defined by

\[ q.\text{canonize} = a \]
\[ q.\text{choose} a = a. \]

If we now take \( l = (rquot l_1 q): l_2 \) (where the equivalence on the left is the total relation on \( \{a, b\} \), which is strictly coarser than equality, the relation on the right), then the PutGet law fails:

\[ l.\text{get} (l.\text{put} b b) \]
\[ = l.\text{get} a \]
\[ = l_2.\text{get} (q.\text{choose} (l_1.\text{get} a)) \]
\[ = a \]
\[ \neq b \]

Conversely, if we take

\[ l = l_2; (lquot q l_1) \]

(where the left equivalence is equality and the right equivalence is the total relation on \( \{a, b\} \)), then the GetPut law fails, since \( a = l.\text{get} b \) but

\[ l.\text{put} a b \]
\[ = l_2.\text{put} ((lquot q l_1).\text{put} a (l_2.\text{get} b)) b \]
\[ = l_2.\text{put} (q.\text{choose} (l_1.\text{put} a (q.\text{canonize} (l_2.\text{get} b)))) b \]
\[ = l_2.\text{put} a b \]
\[ = a \]
\[ \neq b. \]

This failures make intuitive 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 their types using canonizers, and how to compose them. We have not, however, discussed where canonizers come from! Of course, we can always define canonizers 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 we can do better: we can build a canonizer out of the get and create components of an arbitrary lens—indeed, an arbitrary quotient lens!

\[
\begin{array}{c}
l \in S/\sim_S \iff U/\sim_U \\
\text{canonizer_of_lens } l \in S \leftrightarrow U/\sim_U \\
canonize \ s = l.\text{get} \ s \\
\text{choose } u = l.\text{create} \ u
\end{array}
\]

**4.2.7 Lemma:** Let \( l \in S/\sim_S \iff U/\sim_U \) be a quotient lens. Then \( \text{canonizer_of_lens } l \) is a canonizer in \( S \leftrightarrow U/\sim_U \).

Building canonizers from lenses gives us a pleasingly parsimonious design, allowing us to define canonizers using whatever generic or domain-specific primitives are already available on lenses (e.g., in our implementation, primitives for copying, deleting, etc., as well as all of the regular operators).

A composition operator on canonizers can be derived from the quotienting operators. We state a simple version here, whose type can be derived straightforwardly from the types of \( \text{copy} \), \( \text{lquot} \), and \( \text{canonizer_of_lens} \).

\[
q_1 \in S \leftrightarrow U/ = q_2 \in U \leftrightarrow V/ = \\
(q_1; q_2) \in S \leftrightarrow V/ = \\
(q_1; q_2) \triangleq \text{canonizer_of_lens } (\text{lquot } q_1 (\text{lquot } q_2 (\text{copy } V)))
\]

(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 useful to design primitive canonizers *de novo*. The canonizer law imposes fewer restrictions than the lens laws so we have considerable latitude for writing specific canonizing transformations that would not be valid as lenses—see Section 4.4.

### 4.3 Regular Operators

Having presented the semantic space of quotient lenses and several generic combinators, we now focus our attention on quotient lenses for strings. The quotient lenses combinators in this section are direct generalizations of corresponding basic string combinators described in the last chapter. In particular, their functional components are identical to the basic lens versions. However, the typing rules are different, since they define equivalence relations on the source and view types.

**Concatenation** Before we can define the quotient version of concatenation, we need to lift concatenation to relations:

#### 4.3.1 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 $L_2$. The relation $(R_1 \cdot R_2)$ is defined as $w (R_1 \cdot R_2) w'$ 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 slightly careful here: the concatenation of two equivalence relations $\sim_1$ and $\sim_2$ on languages $L_1$ and $L_2$ is not an equivalence in general because it may not be transitive. However, it is an equivalence in two important cases

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

When we concatenate the equivalence relations of the source and view in the concatenation lenses, the typing rule will ensure that the concatenations of the underlying languages are both unambiguous.

With this definition in hand, the concatenation operator is simply:
4.3.2 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 \((S_1 \cdot S_2) / \sim S \) and \((V_1 \cdot V_2) / \sim V \). Then \((l_1 \cdot l_2) / \sim \) is a quotient lens in \((S_1 \cdot S_2) / \sim S \iff (V_1 \cdot V_2) / \sim V \) where \( \sim S \triangleq (\sim S_1 \cdot \sim S_2) \) and \( \sim V \triangleq (\sim V_1 \cdot \sim V_2) \).

Concatenation raises an interesting point: suppose that we have two canonizers, \( q_1 \) and \( q_2 \), and two 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 \( l_1 \) and \( l_2 \) and the canonizers \( q_1 \) and \( q_2 \) first and then quotient the results. Both are possible in our system and both constructions yield equivalent quotient lenses (when they are well-typed)\(^4\). We define concatenation on canonizers precisely and prove this fact below—see Lemma 4.3.8.

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

4.3.3 Definition [Relation Iteration]: Let \( L \) be a regular language, and let \( R \) be a binary relation on \( L \). The relation \( R^* \) is defined as \( w R^* w' \) if and only if there exist strings \( w_1 \) to \( w_n \) belonging to \( L \) and strings \( w'_1 \) to \( w'_n \) in \( L \) with \( w = (w_1 \cdots w_n) \) and \( w' = (w'_1 \cdots w'_n) \) and \( w_i R w'_i \) for all \( i \) in \( \{1 \ldots n\} \).

Again, the iteration of a 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.

\(^4\)Quotienting the lenses first is a little more flexible, since the concatenation of the original quotient lenses need not be unambiguous.
Using this definition, the generalization of Kleene star to quotient lenses is straightforward:

\[
\begin{align*}
    & l \in S \bowtie S \leftrightarrow V \bowtie V \quad S^* \quad V^* \\
    & l^* \in S^*/(\sim_S^*) \leftrightarrow V^*/(\sim_V^*)
\end{align*}
\]

4.3.4 Lemma: Let \( l \in S \bowtie S \leftrightarrow V \bowtie V \) be a quotient lens such that \( S^* \) and \( V^* \). Then \( l^* \) is a quotient lens in \( S^*/\sim_S^* \leftrightarrow V^*/\sim_V^* \).

Union

The next combinator forms the union of two quotient lenses:

\[
\begin{align*}
    & l_1 \in S_1 \bowtie S_1 \leftrightarrow V_1 \bowtie V_1 \\
    & l_2 \in S_2 \bowtie S_2 \leftrightarrow V_2 \bowtie V_2 \\
    & S_1 \cap S_2 = \emptyset \\
    & \forall v, v' \in (V_1 \cap V_2). v \bowtie V_1 v' \leftrightarrow v \bowtie V_2 v' \\
    & \sim_S = \sim_{S_1} \cup \sim_{S_2} \\
    & \sim_V = \sim_{V_1} \cup \sim_{V_2} \\
    & l_1 | l_2 \in S_1 \cup S_2 \bowtie V_1 \leftrightarrow V_1 \cup V_2 \bowtie V_2
\end{align*}
\]

4.3.5 Lemma: Let \( l_1 \in S_1 \bowtie S_1 \leftrightarrow V_1 \bowtie V_1 \) and \( l_2 \in S_2 \bowtie S_2 \leftrightarrow V_2 \bowtie V_2 \) be quotient lenses such that \( (S_1 \cap S_2) = \emptyset \) and for all \( v \) and \( v' \) in \( (V_1 \cap V_2) \) we have \( v \bowtie V_1 v' \) if and only if \( v \bowtie V_2 v' \). Then \( (l_1 | l_2) \) is a quotient lens in \( S_1 \cup S_2 \bowtie V \leftrightarrow V_1 \cup V_2 \bowtie V \) where \( \sim_S = (\sim_{S_1} \cup \sim_{S_2}) \) and \( \sim_V = (\sim_{V_1} \cup \sim_{V_2}) \).
The relations $\sim_S$ and $\sim_V$ are formed by taking the the union of the corresponding relations 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 the quotient lens laws. It stipulates that $\sim_{V_1}$ and $\sim_{V_2}$ must only relate strings in the intersection $(V_1 \cap V_2)$ to other elements of $(V_1 \cap V_2)$ and that $\sim_{V_1}$ and $\sim_{V_2}$ must agree in the intersection. To see why this is needed, suppose we have $v$ in $(V_1 \cap V_2)$ and $v' \in (V_2 - V_1)$ with $v \sim_{V_2} v'$, and let $s \in S_1$ with $(l_1 | l_2).\text{get} \ s = v$. Then $(l_1 | l_2).\text{put} \ v' \ s = l_2.\text{create} \ v' \ c$. Since $(\text{cod}(l_1.\text{put}) \cap \text{cod}(l_2.\text{create})) = \emptyset$, the result cannot be related to $s$ by $\sim_S$—i.e., GETPUT fails.

**Swap** We can also define a quotient lens version of swap. Strictly speaking, swap is not a regular operator—we include it in this section because it is a natural generalization of concatenation. The definition of swap is as follows:

\[
\begin{array}{c|c|c}
S_1 \cdot 1 \cdot S_2 & V_2 \cdot 1 \cdot V_1 \\
\hline
l_1 \in S_1 / \sim_{S_1} \iff V_1 / \sim_{V_1} & l_2 \in S_2 / \sim_{S_2} \iff V_2 / \sim_{V_2} \\
\sim_S = \sim_{S_1} \cdot \sim_{S_2} & \sim_V = \sim_{V_2} \cdot \sim_{V_1} \\
(l_1 \sim l_2) \in (S_1 \cdot S_2) / \sim_S \iff (V_1 \cdot V_2) / \sim_V
\end{array}
\]

\[
\begin{align*}
\text{get} (s_1 \cdot s_2) & = (l_2.\text{get} \ s_2) \cdot (l_1.\text{get} \ s_1) \\
\text{put} (v_2 \cdot v_1) (s_1 \cdot s_2) & = (l_1.\text{put} \ v_1 \ s_1) \cdot (l_2.\text{put} \ v_2 \ s_2) \\
\text{create} (v_2 \cdot v_1) r & = (l_1.\text{create} \ v_1) \cdot (l_2.\text{create} \ v_2)
\end{align*}
\]

**4.3.6 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 $(S_1 \cdot 1 \cdot S_2)$ and $(V_2 \cdot 1 \cdot V_1)$. Then $(l_1 \sim l_2)$ is a quotient lens in $(S_1 \cdot S_2) / \sim_S \iff (V_1 \cdot V_2) / \sim_V$ where $\sim_S \triangleq (\sim_{S_1} \cdot \sim_{S_2})$ and $\sim_V \triangleq (\sim_{V_2} \cdot \sim_{V_1})$.

**Canonizers**

Now we define each of the regular operators on canonizers. Since canonizers only have to satisfy the RECANONIZE law, we have some additional flexibility compared to lenses.

**Concatenation** The concatenation operator on quotient lenses requires that the concatenations of the languages on the left and on the right each be unambiguous; with canonizers, we only need the concatenation on the left be unambiguous:
\( q_1 \in S_1 \leftrightarrow U_1 / \sim U_2 \quad q_2 \in S_2 \leftrightarrow U_2 / \sim U_2 \)

\[ S_1 \cdot S_2 \sim U = \text{TransClosure}(\sim U_1 \cdot \sim U_2) \]

\[ \text{split} \in \Pi u : (U_1 \cdot U_2). \{(u_1, u_2) \in (U_1 \times U_2) | (u_1 \cdot u_2) = u\} \]

\[ q_1 \cdot q_2 \in S_1 \cdot S_2 \leftrightarrow U_1 \cdot U_2 / \sim U \]

\[
\text{canonize} \ (c_1 \cdot c_2) = (q_1.\text{canonize} \ s_1) \cdot (q_2.\text{canonize} \ s_2) \\
\text{choose} \ u = (q_1.\text{choose} \ u_1) \cdot (q_2.\text{choose} \ u_2) \\
\text{where} \ \text{split} \ u = (u_1, u_2)
\]

4.3.7 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 \( S_1 \cdot S_2 \). Also let \( \text{split} \) be a function in \( \Pi u : (U_1 \cdot U_2). \{(u_1, u_2) \in (U_1 \times U_2) | (u_1 \cdot u_2) = u\} \). Then \( q_1 \cdot q_2 \) is a canonizer in \( S_1 \cdot S_2 \leftrightarrow U_1 \cdot U_2 / \sim U \) where \( \sim U = \text{TransClosure}(\sim U_1 \cdot \sim U_2) \).

The \( \text{split} \) function determines how strings in the concatenation of \( U_1 \) and \( U_2 \), which may be ambiguous, should be split. The dependent type for \( \text{split} \) ensures that it splits strings in \( (U_1 \cdot U_2) \) into substrings belonging to \( U_1 \) and \( U_2 \) (e.g., we can instantiate \( \text{split} \) with a function that uses a longest-match policy, or with one that uses a shortest-match policy.) Note that we need to 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 a result we described earlier: canonizing and quotienting (on the left) in either order yields equivalent quotient lenses

4.3.8 Lemma: Let

\[
l_1 \in U_1 / \sim V_1 \leftrightarrow V_1 / \sim V_1 \quad q_1 \in S_1 \leftrightarrow U_1 / \sim U_1 \\
l_2 \in U_2 / \sim U_2 \leftrightarrow V_2 / \sim V_2 \quad q_2 \in S_2 \leftrightarrow U_2 / \sim U_2
\]

be quotient lenses and canonizers and suppose that

\[
l \triangleq (l \text{quot} \ q_1 \ l_1) \cdot (l \text{quot} \ q_2 \ l_2)
\]

\[
l' \triangleq l \text{quot} \ (q_1 \cdot q_2) \ (l_1 \cdot l_2)
\]

are well typed (according to the inference rules given in this section). Then \( l \) and \( l' \) are equivalent.

Proof: Let \( l_1 \) and \( l_2 \) be quotient lenses and \( q_1 \) and \( q_2 \) be canonizers with the types above and define quotient lenses \( l \) and \( l' \). Also, suppose that \( l \) and \( l' \) are well typed according to the inference rules given in this section.
By the typing derivation for \( l \), we have that \( V_1 \cdot V_2 \). Similarly, by the typing derivation for \( l' \), we have that \( U_1 \cdot U_2 \). Using these facts, we prove that the component functions of \( l \) and \( l' \) are equivalent:

**get:** Let \( s = (s_1 \cdot s_2) \in (S_1 \cdot S_2) \) be a string. As \( S_1 \cdot S_2 \) there exist unique strings \( s_1 \in S_1 \) and \( s_2 \in S_2 \) such that \( s = (s_1 \cdot s_2) \). We calculate as follows

\[
l.\text{get } s = l.\text{get } (s_1 \cdot s_2)
= ((l\text{quot } q_1 \cdot l_1) \cdot (l\text{quot } q_2 \cdot l_2)).\text{get } (s_1 \cdot s_2)
= ((l\text{quot } q_1 \cdot l_1).\text{get } s_1) \cdot ((l\text{quot } q_2 \cdot l_2).\text{get } s_2)
= (l_1.\text{get } ((q_1.\text{canonize } s_1)) \cdot (l_2.\text{get } ((q_2.\text{canonize } s_2)))
= (l_1.l_2).\text{get } ((q_1.q_2).\text{canonize } (s_1 \cdot s_2))
= (l_1.l_2).\text{get } ((q_1.q_2).\text{canonize } (s_1 \cdot s_2))
= l'.\text{get } (s_1 \cdot s_2)
= l'.\text{get } s
\]

and obtain the required equality.

**put:** Let \( v = (v_1 \cdot v_2) \in (V_1 \cdot V_2) \) and \( s \in (S_1 \cdot S_2) \) be strings. 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 \( s = (s_1 \cdot s_2) \). We calculate as follows

\[
l.\text{put } v \cdot s = l.\text{put } (v_1 \cdot v_2) \cdot (s_1 \cdot s_2)
= ((l\text{quot } q_1 \cdot l_1) \cdot (l\text{quot } q_2 \cdot l_2)).\text{put } (v_1 \cdot v_2) \cdot (s_1 \cdot s_2)
= ((l\text{quot } q_1 \cdot l_1).\text{put } v_1 \cdot s_1) \cdot ((l\text{quot } q_1 \cdot l_1).\text{put } v_2 \cdot s_2)
= (q_1.\text{choose } ((l_1.\text{put } v_1 \cdot (q_1.\text{canonize } s_1))).
\quad (q_2.\text{choose } ((l_2.\text{put } v_2 \cdot (q_2.\text{canonize } s_2)))
= (q_1.q_2).\text{choose } ((l_1.l_2).\text{put }
\quad (v_1 \cdot v_2) \cdot ((q_1.q_2).\text{canonize } (s_1 \cdot s_2))).
= (l\text{quot } q_1.q_2).\text{put } (v_1 \cdot v_2) \cdot (s_1 \cdot s_2)
= l'.\text{put } (v_1 \cdot v_2) \cdot (s_1 \cdot s_2)
= l'.\text{put } v \cdot s
\]

64
and obtain the required equality.

\textbf{create:} Similar to the previous case.

\textbf{Kleene Star} The iteration operator on canonizers is similar:

\begin{align*}
q \in S \quad & \mapsto U/\sim_U \\
S^t \quad & \sim_U = \text{TransClosure}(\sim_U) \\
split \in \Pi u : *. \{[u_1, \ldots, u_n] \in U \mid (u_1 \cdots u_n) = u\} \\
q^* \in S^* \quad & \mapsto U^*/\sim_U \\
\text{canonize } s_1 \cdots s_n = (q.\text{canonize } s_1) \cdots (q.\text{canonize } s_n) \\
\text{choose } u = (q.\text{choose } u_1) \cdots (q.\text{choose } u_n) \\
\text{where } \text{split } u = [u_1, \ldots, u_n]
\end{align*}

\textbf{4.3.9 Lemma:} Let \( q \in S \mapsto U/\sim_U \) be a canonizer such that \( S^t \). Also let \( \text{split} \) be a function in \( \Pi u : *. \{[u_1, \ldots, u_n] \in U \mid (u_1 \cdots u_n) = u\} \). Then \( q^* \) is a canonizer in \( S^* \mapsto U^*/\sim_U \) where \( \sim_U = \text{TransClosure}(\sim_U^*) \).

Here the \( \text{split} \) function takes a string in \( U^* \) and splits it into a list of substrings belonging to \( U \).

\textbf{Union} The final combinator in this section forms the union of two canonizers.

\begin{align*}
q_1 \in S_1 \quad & \mapsto U_1/\sim_U \quad (S_1 \cap S_2) = \emptyset \\
q_2 \in S_2 \quad & \mapsto U_2/\sim_U \\
\sim_U = \text{TransClosure}(\sim_{U_1} \cup \sim_{U_2}) \\
q_1 \mid q_2 \in (S_1 \cup S_2) \quad & \mapsto (U_1 \cup U_2)/\text{Trans Closure}(\sim_{U_1} \cup \sim_{U_2}) \\
\text{canonize } s = \begin{cases} q_1.\text{canonize } s \quad & \text{if } s \in S_1 \\ q_2.\text{canonize } s \quad & \text{otherwise} \end{cases} \\
\text{choose } u = \begin{cases} q_1.\text{choose } u \quad & \text{if } u \in U_1 \\ q_2.\text{choose } u \quad & \text{otherwise} \end{cases}
\end{align*}

\textbf{4.3.10 Lemma:} Let \( q_1 \in S_1 \mapsto U_1/\sim_U \) and \( q_2 \in S_2 \mapsto U_2/\sim_U \) be canonizers such that \( (S_1 \cap S_2) = \emptyset \). Then \( (q_1 \mid q_2) \) is a canonizer in \( S_1 \cup S_2 \mapsto (U_1 \cup U_2)/\sim_U \) where \( \sim_U = \text{TransClosure}(\sim_{U_1} \cup \sim_{U_2}) \).

The typing rule explicitly closes \( (\sim_{U_1} \cup \sim_{U_2}) \) transitively to ensure that it is an equivalence relation.
4.4 Primitives

So far, we have mostly focused on quotient lenses defined using $lquot$, $rquot$, $canonizer$ of $lens$, and basic lenses. In this section, we give some examples of quotient lens and canonizers that cannot be built using the quotienting and coercion operators from basic lenses.

**Duplication**  In many applications, it is useful to have a way to duplicate part of the the view—e.g., consider augmenting a view of a document with a generated table of contents. Unfortunately, if $S$ is the source type, it is impossible to have a duplication lens with type $S \equiv (S \cdot S)$—e.g., consider editing just one copy of the duplicated data; the $PutGet$ law will not be satisfied because the $get$ function will make the two copies equal. We can have duplication if we change its type to the following:

$$S \equiv \{(s \cdot s') \in (S \cdot S) \mid s = s'\}$$

but the types used in our string lenses—regular languages—are not expressive enough to track equalities between arbitrary strings. However, in the relaxed space of quotient lenses, we can give the duplication operator a natural type:

$$l \in S/\sim S \equiv V_1/\sim V_1 \quad f \in S \rightarrow V_2 \quad V_1 \cdot V_2 \sim V \equiv \sim V_1 \cdot \text{Tot}(V_2)$$

$$\text{dup}_1 \quad l \in S/\sim S \equiv (V_1 \cdot V_2)/\sim V$$

$$\begin{align*}
g & \text{get } s & = (l \text{.get } s) \cdot (f \text{ s}) \\
p & \text{put } (v_1 \cdot v_2) \text{ s} & = (l \text{.put } v_1 \text{ s}) \\
c & \text{create } (v_1 \cdot v_2) & = (l \text{.create } v_1)
\end{align*}$$

**4.4.1 Lemma:** Let $l \in S/\sim S \equiv V_1/\sim V_1$ be a basic lens and $f \in S \rightarrow V_2$ a function such that $V_1 \cdot V_2$. Then $(\text{dup}_1 \ l \ f)$ is a quotient lens in $S/\sim S \equiv (V_1 \cdot V_2)/\sim V$ where $\sim V = \sim V_1 \cdot \sim V_2$.

The $\text{dup}_1$ lens takes as arguments a quotient lens $l$ and a function $f$ with the same domain as $l$ ($f$ is often the $get$ component of a lens). The $get$ function copies the source string, passes one copy to $l$'s $get$ component and the other copy to $f$, and concatenates the results. The $put$ function discards the portion of the view generated by $f$ and invokes $l$'s $put$ function on the rest of the view.
As an example, suppose that \( e \) and \( e' \) are strings belonging to \([E]\). The get component of
\[
dup_1 \text{ (copy } E \text{)} \ ((\text{copy } E).\text{get})
\]
maps \( e \) to \((e\cdot e')\), and the create component maps \((e\cdot e')\) to \( e \). The typing rule for \( \text{dup}_1 \) records the fact that it ignores the part of the view generated by \( f \). A symmetric operator \((\text{dup}_2 f l)\) discards the first copy instead of the second in the reverse direction.

In both of these quotient lenses, the handling of duplicated data is admittedly simple. (In particular, unlike the duplication operators proposed and extensively studied by Hu et al. (2004), put and create do not make any attempt to merge changes to the duplicated data in the abstract string.) Nevertheless, they suffice for many examples that arise in practice. For example, when \( f \) is an aggregation operator such as \((\text{count } E)\), which takes a string \( u \) belonging to \([E]^*\) and returns the number of substrings belonging to \( E \) that \( u \) can be split into, discarding the aggregate value while propagating the changes made to the other copy often makes sense.

**Normalize**  The next operator is a combinator that builds a canonizer from a function that maps a set of structures onto a “normalized” subset of itself.

\[
\begin{align*}
f \in S &\rightarrow S_0 \\
S_0 &\subseteq S \\
\forall s \in S_0. \ f \ s = s \\
\hline
\text{normalize } f \in S &\leftrightarrow S_0/=
\end{align*}
\]

\[
\begin{align*}
canonize s &= f \ s \\
\text{choose } s &= s
\end{align*}
\]

**4.4.2 Lemma:** Let \( S \) and \( S_0 \) be sets such that \( S_0 \subseteq S \). Also let \( f \in S \rightarrow S_0 \) be a function from \( S \) to \( S_0 \). Then \((\text{normalize } f)\) is a canonizer in \( S \leftrightarrow S_0/= \).

The canonize component is given by the function \( f \), and the choose component is the identity function. Together, the constraint on the behavior of \( f \) and the condition that \((S_0 \subseteq S)\) ensure that ReCanonize holds.

As an example showing how normalize is used, consider a canonizer to put substrings of a bigger string in sorted order. To keep the notation simple, we will describe the binary version; the generalization to an \( n \)-ary sort 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 with type
\[
f \in ((S_1 \cdot S_2) \cup (S_2 \cdot S_1)) \rightarrow (S_1 \cdot S_2)
\]
defined by the equations

\[ f(s_1 \cdot s_2) = (s_1 \cdot s_2) \]
\[ f(s_2 \cdot s_1) = (s_1 \cdot s_2) \]

where \( s_1 \in S_1 \) and \( s_2 \in S_2 \). It is easy to check that \( f \) satisfies the side condition in the typing rule for \textit{normalize} with \( S_0 = (S_1 \cdot S_2) \) because \( f \) maps strings in sorted order to themselves. The canonizer \((\text{sort } S_1 S_2)\) defined as \((\text{normalize } f)\) has type:

\[ (S_1 \cdot S_2 \cup S_2 \cdot S_1) \leftrightarrow S_1 \cdot S_2 \]

We use this canonizer (and a variant, discussed in Section 4.5) in many of our examples to canonize the representation of unordered data—e.g., XML attributes, BibTeX fields, etc.

**Columnize**

Many ad hoc formats require that long lines of text be wrapped by replacing a space character with a newline followed by several spaces. For example, the UniProt genomic database described in Section 4.6 requires that lines exceeding 75 characters be wrapped so that they do not spill over into the margin. The next primitive does this transformation:

\[
\begin{align*}
  n &\in \mathbb{N} & sp &\in \Sigma^* & nl &\in \Sigma^* \\
  (\Sigma^* \cdot nl \cdot \Sigma^*) &\cap S_0 = \emptyset \\
  S &\equiv [(sp \cup nl)/sp]S_0 \\
\end{align*}
\]

canonize \( n \) \( S_0 \) \( sp \) \( nl \) \( \in S \leftrightarrow 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 \( n \)

**4.4.3 Lemma:** Let \( n \) be a number, \( S_0 \subseteq \Sigma^* \) a language, and \( sp \in \Sigma^* \) and \( nl \in \Sigma^* \) strings such that \( nl \) does not occur in any string in \( S_0 \). Then \((\text{columnize } n \ S_0 \ sp \ nl)\) is a canonizer in \( S \leftrightarrow S_0 \) where \( S = [(sp \cup nl)/sp]S_0 \).

It takes as arguments a number \( n \), a set of strings \( S_0 \), a “space” string \( sp \), and a “newline” string \( nl \). Its \textit{canonize} component replaces every occurrence of newline with a space; the \textit{choose} component breaks lines longer than \( n \) as necessary by replacing spaces with newlines. The typing rule for \textit{columnize} requires that \( nl \) not appear in strings in \( S_0 \) and assigns to the entire canonizer the type \( S \leftrightarrow S_0 \) where \( S \) is obtained by widening \( S_0 \) so that \( nl \) may appear anywhere that \( sp \) may.

68
4.5 Typechecking

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 our language to build larger examples, we discovered a significant—and completely unexpected—side benefit: quotient lenses allow us to assign many bidirectional transformations coarser types than the strict lens laws permit, easing some serious tractability concerns.

The need for precise types stems from a fundamental choice in our design: put functions are total. Totality is attractive to users of lenses, because it guarantees that any valid view can be put back with any valid source. However, for exactly the same reason, totality makes it difficult to design lens primitives—the put function must do something reasonable with every valid view and source, and 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 algebra of types.

To be sure, working in a language with very precise types has many advantages. For example, Boomerang’s type checker, based on regular expressions, uncovered a subtle source of ambiguity in the UniProt ASCII format. But it also imposes burdens—programmers must write programs that satisfy a very picky type checker, and implementations must mechanize these precise analyses, often relying on algorithms that are computationally expensive. Fortunately, the increased flexibility of quotient lenses and canonizers can be exploited to loosen types and alleviate these burdens. We discuss three examples of this phenomenon in this section.

The first example involves the columnize transformation, which was defined as a primitive canonizer in Section 4.4. The mappings between long lines of text and blocks of well-wrapped lines are a bijection, so they trivially satisfy the lens laws. We could define columnize as a basic 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, and so would be quite tedious to write). However, the type of this lens, whose view type is 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) is horribly complicated and cumbersome—both for programmers and in the implementation.
We could loosen the type to match the one we gave to the *columnize* canonizer—i.e., to arbitrary blocks of text, including blocks containing “extra” newlines—but changing the type in this way also requires changing the *put* function in order to avoid violating the *G*3*P*2*V*4 law. In particular, if we take a concrete block of text containing some extra newlines, map it to an abstract line by *get*, and immediately map it back to a concrete block by *put*, then the strict version of *G*3*P*2*V*4 stipulates that all of the extra newlines must be restored exactly. Thus, the *put* function cannot ignore its concrete argument and insert the minimal number of newlines needed to avoid spilling over into the margin; it must also examine the concrete string and restore any extra newlines from it. Formulating *columnize* as a canonizer rather than a lens, avoids all of these complications 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 *columnize*, it is possible to define a basic lens version of *sort*. To sort \((S_1 \ldots S_k)\), we form the union of lenses that recognize the concatenations of permutations of the \(S_i\)s and apply the appropriate permutation to put them in sorted order. This lens has the behavior we want, but its type on the concrete side is the set of all concatenations of permutations of \(S_i\)s—a type whose size grows as the factorial of \(k\)! Representing this type in the implementation rapidly becomes impractical. Fortunately, the combinatorial blowup can be avoided by widening the concrete type to \((S_1 | \ldots | S_n)^*\). This type over 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 reasonable choice, which works well for many examples is to simply discard the extras and fill in missing ones with defaults.

The final example involves the duplication operator. As discussed above, it is possible to have duplication as a basic lens, but to satisfy *Put*/*Get* the type of strings in the view must include an equality constraint. By defining \(\text{dup}_1\) and \(\text{dup}_2\) as a quotient lens, we obtain a primitive with a much simpler—in fact, regular—type.

These examples show how quotient lenses ease certain aspects of typechecking. However, they complicate other aspects of typechecking because the typechecker needs to track equiva-
lence relations. In particular, 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, we need to verify that the equivalence on \(l\)'s view and the one on \(k\)'s source are identical.

In the rest of this section, we describe two different approaches to 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 too expensive to be useful in practice.

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, \texttt{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:

\[ \tau ::= \text{Identity} \mid \text{Any} \]

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 \texttt{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 to write many thousands of lines of demo applications. There are two reasons for this. First, it allows two quotient lenses to be composed, whenever the uses of \texttt{lquot} are all in the lens on the left and the uses of \texttt{rquot} on the right, a very common case. And second, it allows arbitrary quotient (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 used in Boomerang.
In theory, we can go further by replacing the Identity sort with a tag carrying an arbitrary finite state transduction \( f \)—i.e., a function computable by a finite state transducer (Berstel, 1979):

\[
\tau := \text{Fst of } f \mid \text{Any}
\]

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.

4.5.1 Definition: Let \( f \in A \rightarrow B \) be a rational function. Denote by \( \sim_f \) the relation \( \{(x, y) \in A \times A \mid f(x) = f(y)\} \).

4.5.2 Lemma: Let \( f \in A \rightarrow B \) and \( g \in A \rightarrow C \) be rational and surjective functions. Define a rational relation \( h \subseteq C \times B \) as \( (f \circ g^{-1}) \). Then \( \sim_g \subseteq \sim_f \) iff \( h \) is functional.

Proof: Let us expand the 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 \( f(a) = f(a') \) and so \( a \sim_f a' \). Since \( a \) and \( a' \) were arbitrary, we conclude that \( \sim_g \subseteq \sim_f \).

4.5.3 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 \( \sim_f = \sim_g \) iff 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 (Blattner, 1977).

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

4.6 Example

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/SwissProt 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:

\[
\text{<citation type="patent" date="1990-09-20" number="WO9010703"/>}
\]

In ASCII, they appear as RL lines:

\[
\text{RL Patent number WO9010703, 20-SEP-1990.}
\]

The bidirectional transformation between these formats is essentially bijective—the patent number can be copied verbatim from the attribute to the line, and the date just needs to 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 first seem.

A bad choice (the only choice available with just 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, since it 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 library code identified below.)

```ocaml
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. It is passed as an argument to an qdel instance that constructs the leading whitespace for the XML element in the reverse 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, the Xml.attr3_elt_no_kids function sorts the attributes to put them into this order. The space, comma, and dot lenses insert the indicated characters; escaped_pcdata handles unescaping of PCDATA; date 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

```xml
<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:

```ocaml
let oc_taxon : lens = Xml.pcdata_elt NL3 "taxon" esc_pcdata in
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; Euteleostomi;
OC Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini;
OC Catarrhini; Hominidae; Homo.
```

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

```ocaml
let oc_q : canonizer = columnize 75 (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; `stype`, used below, extracts the source part.)

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:
Using \textit{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 \textit{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

\begin{verbatim}
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;
\end{verbatim}

where the \texttt{Named isoforms} field in the second line is the count of the number of \texttt{Name} blocks that follow below. The Boomerang code that generates these lines uses \textit{dup}_2 and \textit{count} to generate the appropriate integer in the \textit{get} direction; in the reverse direction, it simply discards the integer generated by \textit{count}.

### 4.7 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 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.
Chapter 5

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 view with 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 are only able to 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 fine, but when the update breaks the association, it does not—it causes the put function to reintegrate information extracted from pieces of the source with completely unrelated pieces of the view.

Alignment is especially critical in the context of lenses because, unlike view update translators that work on explicit update operations (e.g., insert, delete, reorder, etc.), lenses are state based and therefore agnostic to the way that updates are expressed. As discussed in previous chapters, this design is well-motivated—it makes it easy to deploy lenses in a wide variety of application contexts—but because lenses are not supplied with the update operation performed on the view, they cannot use the update to align the original source and the updated view. Instead, they need to calculate an alignment using a function that is appropriate to the application at hand.
As an example illustrating the problems with alignment that come up with lenses, recall the composers lens and consider a source XML tree

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

and the corresponding ASCII view:

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

Notice that in going from source to view, the get component of the composers lens deletes the nationality of each composer (along with all of the XML formatting). In the put direction it restores nationalities positionally—i.e., by matching up the $i$th line in the view with the $i$th XML element in the source. For certain updates—e.g., when composers have been edited in place and perhaps added at the end of the view, but the order of lines has not changed—this simple policy does a good job. For example, if the update to the view replaces Sibelius’s death date with “1957”, corrects Briten’s name to from “Britten” to “Britten”, and adds a new line for Tansman, the put function will combine the new view

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

with the original source and yield an update XML tree that accurately reflects the modifications made to the view:

```xml
<composers>
    <composer>
        <name>Jean Sibelius</name>
        <lived>1865-1957</lived>
        <nationality>Finnish</nationality>
    </composer>
</composers>
```
On other updates, however, the behavior of the *put* function is highly unsatisfactory. If the update to the view breaks the positional association between lines in the source and view, the output will be mangled—e.g., when the view is obtained by making the same corrections to Sibelius’s death date and Britten’s name as above and *swapping* the order of Britten and Copland, combining

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

with the original source produces an output

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

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

This is a serious problem, and a pervasive one: it is triggered whenever a lens whose *get* function discards information is iterated over an ordered source and the update to the list of
items in the view does not preserve the positional correspondence between items in the source and view. It is a show-stopper for many of the applications we want to write using lenses.

What we would like, of course, is for the put function to align 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 put function that matched up composers by name would produce a result

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

where each nationality is restored to the appropriate composer. Unfortunately, neither basic lenses nor any of the other existing bidirectional languages provides the means to achieve this effect. Developing mechanisms that allow lenses to specify and use sophisticated alignment strategies is the goal of this chapter.

Our solution is to enrich lenses with a simple mechanisms for specifying alignment policies for the source and view. The idea is that the programmers should specify the locations of chunks in the source and view as well as the policy to use for aligning chunks. The put function can use the specified policy to compute an association—formally, a partial bijection—between the chunks in the original view and the updated view and combine this association with the association between chunks in the source and view realized by the lens to obtain an end-to-end alignment that connects chunks in the original source with chunks in the updated view. It can then use this alignment to put back corresponding chunks—i.e., according to the specified policy—together.

Here is a lens that has the desired behavior for the composers example:

```ocaml
let composer : lens =
  xml_elt "composer"
```

80
Compared to the earlier version of this program we have made two changes (both shaded in grey). First, we have enclosed both occurrences of the `composer` lens in angle brackets, indicating that each composer should be treated as a reorderable chunk. The `same_keys` “tag” specifies the policy for aligning chunks in the `put` direction. Second, we have wrapped the lens that copies the name of each composer to view with a special primitive `key`. This specifies that the portion of the view produced by this lens should be used as the key for the enclosing chunk when computing an alignment. Note that we do not actually demand that the key of each chunk be unique—i.e., these “keys” are not keys in the strict database sense. When several chunks in the view have the same key, the relative alignments of chunks having the same key goes by position. The net effect of these changes is that the `put` function aligns composers by name, as desired.

To make these new features behave as expected, we modify the framework of lenses in several ways. Semantically, we enrich the source and view types with notions of chunks and we add behavioral laws that capture the essential constraints on the handling of chunks—e.g., we stipulate that the `get` and `put` functions must carry chunks in the source through to chunks in the view, and vice versa. Using these laws, can derive many intuitive properties—e.g., that reorderings the chunks in the view will be propagated to corresponding reorderings on the source. We call these revised structures **resourceful lenses**.

Operationally, we change the type of the `put` function so that instead of taking the entire source as an argument, it takes a pair of structures: a “rigid complement” that represents the source information that must be handled positionally, and a “resource”, that represents the information in chunks, which can be freely reordered. We also add a new component to lenses called `res` (for “residue”) that extracts these structures from the source. These modifications to the lens architecture are depicted graphically in Figure 5.1. Separating the representation of rigidly ordered and reorderable information in this way provides a clean interface for using alignment information with lenses—by shuffling and discarding resource elements according to the alignment, we can pre-align the resource so that the source chunks will be matched up with
the specified chunks in the view.

Syntactically, we develop new primitives that allow programmers to specify the locations of chunks and alignment policies for chunks. We reinterpret each of the regular operators (union, concatenation, and Kleene star) as resourceful lenses, and we demonstrate that resourceful lenses are closed under composition. One of the main hallmarks of our design is that it is cleanly separates the treatment of alignment information from rest of the lens. Resourceful lenses can be instantiated with arbitrary alignment functions—the only property we require is that these functions return the identity alignment when the structures being aligned are identical. Accordingly, we describe several different algorithms for computing alignments and we show how these algorithms can be tuned using programmer-specified notions of “keys” and “thresholds”.

The rest of this chapter is organized as follows. Section 5.1 fixes some new notation for chunks that we will use to formalize resourceful lenses. Section 5.2 presents the semantic space of resourceful lenses. Section 5.3 describes syntax for resourceful lenses including: coercions to and from basic lenses, a combinator for indicating chunks, and resourceful versions of the regular, swap, and composition operators. Section 5.4 discusses alignment algorithms and several extensions to this core language. We conclude the chapter in Section 5.5.

5.1 Notation

Before we can define resourceful lenses precisely, we need a few additional pieces of notation.
Chunk Annotations

The types of resourceful lenses will be given by regular string languages decorated with extra annotations that indicate the locations of chunks. Let ⟨ and ⟩ be two fresh symbols not in Σ. A chunk-annotated string is a string over the extended alphabet (Σ∪{⟨, ⟩})* in which occurrences of ⟨ and ⟩ are balanced and non-nested:

5.1.1 Definition [Chunk-Annotated Strings]: The set of chunk-annotated strings is generated by the following grammar:

\[ w ::= u | \langle u \rangle | w \cdot w \]

where \( u \) ranges over ordinary strings in \( \Sigma^* \).

Note that every ordinary string is a chunk-annotated string and that chunks only appear at the top level—i.e., they cannot be nested. To describe sets of chunk-annotated strings, we will use chunk-annotated regular expressions:

5.1.2 Definition [Chunk-Annotated Regular Expression]: The set of chunk-annotated regular expressions is generated by the following grammar:

\[ A ::= \mathcal{R} | \langle \mathcal{R} \rangle | A | A \cdot A | A^* \]

where \( \mathcal{R} \) ranges over ordinary regular expressions.

Again, note that every ordinary regular expression is also a chunk-annotated expression, and that chunks only appear at the top level.

A chunk-annotated string can be converted to an ordinary string by erasing annotations:

5.1.3 Definition [Erase Function]: The erase function, written \([\cdot]\), maps chunk-annotated strings to ordinary strings by sending ⟨ and ⟩ to \( \epsilon \) and every other character to itself.

We lift \([\cdot]\) to regular expressions and languages in the obvious way, and we observe that for every chunk-annotated regular expression \( E \) we have \([L(E)] = L([E])\).

We will use chunk-annotations to “read off” the locations of chunks in ordinary strings. Some chunk-annotated languages, however, do not uniquely identify the chunks of strings in their erasure—e.g., we can interpret the in \((a·b)\) using the chunk-annotated regular expression...
\((a \cdot b | a \cdot \langle b \rangle)\) in two ways. We will rule out these situations by only considering chunk unambiguous languages:

5.1.4 Definition [Chunk Unambiguous]: A chunk-annotated regular language \(L\) is chunk unambiguous if and only if the erase function \(\lfloor \cdot \rfloor\) is injective on \(L\).

In what follows, we tacitly assume that all chunk-annotated regular languages under discussion are chunk unambiguous. (And when we define typing rules for our resourceful lens combinators, below, we will be careful to ensure it.)

We reference the chunks in a string by position, starting from 1. For the next few definitions, fix a chunk-annotated regular language \(L\) and let \(u\) be a string in \([L]\). To lighten the presentation, we will not explicitly mention \(L\) in the definitions that follow. It will always be clear from context.

5.1.5 Notation [Locations]: We write \(|u|\) for the number of chunks in \(u\) and \(\text{locs}(u)\) the set of locations of chunks in \(u\).

Observe that \(\text{locs}(u) = \{1, \ldots, |u|\}\).

5.1.6 Notation [Chunk Access and Update]: When \(x\) is a location in \(\text{locs}(u)\) and \(v\) is a string, we write \(u[x]\) for the chunk located at \(x\) in \(u\) and \(w[u:=v]\) for the string obtained from \(u\) by setting the chunk at \(x\) to \(v\).

In general, updating a chunk of a string in \(L\) may not preserve membership in \(L\). However, if membership in \(L\) is preserved by updating chunks with arbitrary strings in a given language \(S\), and if the chunks of each string in \(L\) all belong to \(S\), then we say that \(S\) is chunk compatible with \(L\):

5.1.7 Definition [Chunk Compatible]: A language \(S\) is chunk compatible with \(L\) if and only if for all strings \(u \in L\) and \(v \in S\) and every location \(x \in \text{locs}(u)\) we have \(u[x] \in S\) and \(u[x:=v] \in L\).

To reorder the chunks of a string, we apply a permutation:
5.1.8 Notation [Permutations]: We write \( \text{Perms}(u) \) for the set of permutations of chunks in \( u \) and \( (\varnothing \ u) \) for the string obtained by reordering chunks according to a permutation \( q \) in \( \text{Perms}(u) \).

The skeleton of a string \( u \) is the string that remains after replacing every chunk with \( \Box \), a fresh symbol not in \( \Sigma \).

5.1.9 Notation [Skeleton]: We write \( \text{skel}(u) \) the skeleton of \( u \).

Finite Maps

To represent resources we will use finite maps from locations (i.e., positive integers) to strings. Our notation for finite maps is as follows:

5.1.10 Notation [Empty Map]: We write \{||\} for the totally undefined map.

5.1.11 Notation [Singleton Map]: We write \{\( \{x \mapsto u\}\)\} for the singleton map that associates the location \( x \) to the string \( u \), and is otherwise undefined.

5.1.12 Notation [Map Length]: We write \( |r| \) for the largest element in the domain of \( r \).

Let \( r_1 \) and \( r_2 \) be finite maps. The map \( (r_1 ++ r_2) \) behaves like \( r_1 \) on locations in \( \text{dom}(r_1) \) and like \( r_2 \) (shifted down by \( |r_1| \)) on all other locations:

5.1.13 Definition [Map Concatenation]:

\[
(r_1 ++ r_2)(x) \triangleq \begin{cases} 
  r_1(x) & \text{if } x \leq |r_1| \\
  r_2(x - |r_1|) & \text{otherwise}
\end{cases}
\]

The \( \text{split}(n, r) \) function takes an integer \( n \) and a map \( r \) and splits it in two. The first map behaves like \( r \) restricted to locations less than or equal to \( n \) and the second map behaves like \( r \) shifted up by \( n \):

5.1.14 Definition [Map Split]:

\[
(\pi_1(\text{split}(n, r)))(x) = \begin{cases} 
  r(x) & \text{if } x < n \text{ and } x \in \text{dom}(r) \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

\[
(\pi_2(\text{split}(n, r)))(x) = \begin{cases} 
  r(x + n) & \text{if } (x + n) \in \text{dom}(r) \\
  \text{undefined} & \text{otherwise}.
\end{cases}
\]
Observe that \( \text{split}(|r_1|, r_1 ++ r_2) = (r_1, r_2) \). We describe sets of maps using the following notation:

5.1.15 Notation [Set of Maps]: We write \( \{N \mapsto S\} \) for the set of maps with codomain \( S \).

5.2 Resourceful Lens Semantics

With this notation in place, we are now ready to define the semantic space of resourceful lenses. Unlike the semantic spaces of basic and quotient lenses, which we described completely generically, in this chapter, to keep things simple we will assume that the source and view are strings from the start. (It would not difficult generalize the definition of resourceful lenses to arbitrary structures, but we would need to generalize the notions of chunks, skeletons, etc.)

Let \( S \) and \( V \) be languages of chunk-annotated strings, \( t \) a tag, \( k \) a basic lens, and \( C \) be a set of rigid complements such that \( k.S \) and \( k.V \) (i.e., the source and view type of \( k \)) are chunk compatible with \( S \) and \( V \) respectively. The basic lens \( k \) will be used to process the strings in chunks and the tag \( t \) will be used to determine the policy for aligning chunks. The set \( C \) of rigid complements is an arbitrary set of structures.

5.2.1 Definition [Resourceful Lens]: A resourceful lens \( l \) between \( S \) and \( V \) over \( C \) and \( (t : k) \) comprises functions

\[
\begin{align*}
l.\text{get} &\in [S] \rightarrow [V] \\
l.\text{res} &\in [S] \rightarrow C \times \{N \mapsto k.S\} \\
l.\text{put} &\in [V] \rightarrow C \times \{N \mapsto k.S\} \rightarrow [S] \\
l.\text{create} &\in [V] \rightarrow \{N \mapsto k.S\} \rightarrow [S]
\end{align*}
\]

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 \( r \in \{N \mapsto k.S\} \):

\[
\begin{align*}
l.\text{get} (l.\text{put} v (c, r)) &= v \quad \text{(PutGet)} \\
l.\text{get} (l.\text{create} v r) &= v \quad \text{(CreateGet)} \\
l.\text{put} (l.\text{get} s) (l.\text{res} s) &= s \quad \text{(GetPut)}
\end{align*}
\]
\[ \text{locs}(s) = \text{locs}(\text{l.get } s) \quad \text{(GetChunks)} \]

\[ c, r = \text{l.res } s \quad \text{(ResChunks)} \]

\[ \text{locs}(s) = \text{dom}(r) \]

\[ x \in (\text{locs}(v) \cap \text{dom}(r)) \quad \text{(ChunkPut)} \]

\[ (\text{l.put } v (c, r))[x] = k.\text{put } v[x] (r x) \]

\[ x \in (\text{locs}(v) \cap \text{dom}(r)) \quad \text{(ChunkCreate)} \]

\[ (\text{l.create } v r)[x] = k.\text{put } v[x] (r x) \]

\[ x \in (\text{locs}(v) - \text{dom}(r)) \quad \text{(NoChunkPut)} \]

\[ (\text{l.put } v (c, r))[x] = k.\text{create } v[x] \]

\[ x \in (\text{locs}(v) - \text{dom}(r)) \quad \text{(NoChunkCreate)} \]

\[ (\text{l.create } v r)[x] = k.\text{create } v[x] \]

\[ \text{skel}(v) = \text{skel}(v') \quad \text{(SkelPut)} \]

\[ \text{skel}(\text{l.put } v (c, r)) = \text{skel}(\text{l.put } v' (c, r')) \]

\[ \text{skel}(v) = \text{skel}(v') \quad \text{(SkelCreate)} \]

\[ \text{skel}(\text{l.create } v r) = \text{skel}(\text{l.create } v' r') \]

The set of all resourceful lenses from \( S \) to \( V \) over \( C \) and \( k \) is written \( S \xleftrightarrow{C;k} V \).

The first thing to notice about this definition is that although the types of the source and view are chunk-annotated languages, the functions that make up a resourceful lens operate on ordinary strings—i.e., the type of \( \text{get} \) is \([S] \rightarrow [V]\) not \( S \rightarrow V \). Second, note that the basic lens \( k \) that processes the strings in chunks is built into the semantics of resourceful lenses. This allows us to formulate behavioral laws such as \( \text{ChunkPut} \) and \( \text{NoChunkPut} \) that specify the handling of strings in chunks in terms of \( k \)'s behavior.

The \( \text{PutGet} \), \( \text{CreatePut} \), and \( \text{GetPut} \) laws are the same as the basic lens version (generalized because \( \text{put} \) now takes a rigid complement and a resource rather than the source itself).
The GetChunks law states that the chunks in the source must be carried through to chunks in the view. This rules out lenses whose view type advertises the presence of chunks but whose source type does not have chunks (and vice versa). The ResChunks law requires an analogous property for the resource generated by \( \text{res} \) from the source. Lenses that violate these laws would cause problems with the procedure for using alignments described at the beginning of this chapter—we would not necessarily be able to propagate alignments on the view back to alignments on the source. We do not state explicit PutChunks and CreateGet laws because they can be derived from GetChunks, PutGet, and CreateGet:

5.2.2 Lemma [PutChunks]: For every resourceful lens \( l \in S \rightleftharpoons_{C.t} V \), string \( v \in [V] \), skeleton \( c \in C \), and resource \( r \in \{|N \mapsto k.S|\} \) we have \( \text{locs}(l.\text{put} v (c, r)) = \text{locs}(v) \).

5.2.3 Lemma [CreateChunks]: For every resourceful lens \( l \in S \rightleftharpoons_{C.t} V \), string \( v \in [V] \), and resource \( r \in \{|N \mapsto k.S|\} \) we have \( \text{locs}(l.\text{create} v r) = \text{locs}(v) \).

The ChunkPut and ChunkCreate laws codify how resourceful lenses must use their resources. Formally, they stipulate that the \( i \)th chunk in the source produced by \( \text{Put} \) and \( \text{Create} \) must be the identical to the string produced by \( k.\text{put} \) with the \( i \)th view chunk and the string associated to \( i \) in the resource. The NoChunkPut and NoChunkCreate laws are similar. They specify the behavior of the lens in cases where the resource does not contain an item for the \( i \)th view chunk and it uses \( k.\text{create} \) instead of \( k.\text{put} \).

Finally, the SkelPut law stipulates that the skeleton of the updated source produced by \( \text{put} \) must only depend on the skeleton of the view, and not on any of the chunks in the view or the items in the resource. Among other things, this law is critical for ensuring that the union lens is well behaved.

The laws for resourceful lenses have a more operational flavor than the laws for basic and quotient lenses. This is unavoidable—the resourceful laws spell out the processing of individual chunks in intricate detail—but it is often useful to think in terms of higher-level, more declarative properties. To illustrate the semantics of resourceful lenses, let us prove one such property: that the \( \text{put} \) function of a resourceful lens translates reorderings on the view to corresponding reorderings on the source.
**5.2.4 Lemma [ReorderPut]:** For every resourceful lens $l \in S \overset{C.t:k}{\leftrightarrow} V$, string $v \in [V]$, rigid complement $c \in C$, resource $r \in \{\|N \to k.S\}$, and permutation $q \in \text{Perms}(v)$ we have: 
\[\oplus (l.\text{put } v (c, r)) = l.\text{put } (\oplus v) (c, r \circ q^{-1}).\]

As another illustration of the semantics of resourceful lenses, consider a coercion that takes a resourceful lens and packages it up with the interface of a basic lens. The $\lfloor \cdot \rfloor$ combinator takes a resourceful lens $l$ in $S \overset{C.t:k}{\leftrightarrow} V$ as input and constructs a basic lens $\lfloor l \rfloor$ in $\lfloor S \rfloor \overset{\cdot}{\leftrightarrow} \lfloor V \rfloor$. It realizes the procedure for using a resourceful lens described above in which the put function first pre-aligns the resource using an alignment computed between the chunks in the old view and new view, and then invokes the actual put function:

\[
\begin{align*}
l \in S & \overset{C.t:k}{\leftrightarrow} V \\
[l] & \in [S] \overset{\cdot}{\leftrightarrow} [V]
\end{align*}
\]

\[
\begin{align*}
\text{get } s &= l.\text{get } s \\
\text{res } s &= s \\
\text{put } v s &= l.\text{put } v (c, r \circ g) \\
&\quad \text{where } (c, r) = l.\text{res } s \\
&\quad \text{and } g = \text{align}(v, l.\text{get } s) \\
\text{create } v &= l.\text{create } v \{\emptyset\}
\end{align*}
\]

**5.2.5 Lemma:** Let $l \in S \overset{C.t:k}{\leftrightarrow} V$ be a resourceful lens. Then $[l]$ is a basic lens in $[S] \overset{\cdot}{\leftrightarrow} [V]$.

The basic get function is identical to the resourceful version. The put function, however is different. It takes a view $v$ and a source $s$ as arguments and calculates an alignment $g$ between the chunks in $v$ and the chunks in $s$ using the following five steps: First, it applies get to $s$ to recalculate the original view $v_0$ (in an implementation, $v_0$ could be cached to avoid computing it twice). Second, it applies res to $s$ to calculate a rigid complement $c$ and a resource $r$. Third, it applies the alignment function align to $v$ and $v_0$ calculate the alignment $g$—formally, a partial bijection between the locations of their chunks. The align function itself is controlled by the tag $t$—see Section 5.4. For now, let us assume that align is the function that matches chunks in the view having the same key (and that falls back to a positional strategy when there are multiple chunks with the same key). Fourth, it composes $g$ and $r$. Since $r$ is a finite map from locations to source chunks and $g$ is a partial bijection on locations, this has the effect of pre-aligning $r$ using $g$. Finally, it passes $v$, $c$ and $(r \circ g)$ to l.put, which produces the updated source. The basic
create function invokes \textit{l.create} with the view and the empty resource. Note that the \([l]\) coercion does not place any constraints on the alignment function \textit{align} except that it must return the identity alignment when its arguments are two identical strings (this property is needed to show that \([l]\) obeys GETPUT).

5.3 Resourceful Lens Primitives

This section describes syntax for resourceful lenses. We start with two simple primitives—one that converts from basic to resourceful lenses and another that provides a way for programmers to specify the locations of chunks. Next, we describe resourceful versions of each of the regular operators, generalizing their \textit{put} and \textit{create} functions to operate on rigid complements and resources instead of source strings. The last part of the section presents resourceful versions of the sequential composition and swap operators.

Raise Every basic lens can be used as a resourceful lens, by taking the rigid complement to be the entire source:

\[
\begin{array}{c}
k \in S' \iff V' \\
l \in S \iff V \\
\hat{l} \in S \overset{\text{C; } t}{\leadsto} V
\end{array}
\]

\[
\begin{array}{c}
get s = l.\text{get } s \\
\text{res } s = l.\text{res } s, \{ || \} \\
\text{put } v \ (c, r) = l.\text{put } c \\
\text{create } v \ r = l.\text{create } v
\end{array}
\]

5.3.1 Lemma: Let \(l \in S \iff V\) and \(k \in S' \iff V'\) be basic lenses. Then \(\hat{l}\) is a resourceful lens in \(S \overset{\text{C; } t}{\leadsto} V\).

Since \(l\) is a basic lens, the source and view types \(S\) and \(V\) are ordinary regular expressions that do not contain chunks. We pick the set of strings denoted by \(S\) as the set of rigid complements and let \(\text{res}\) be the function that maps a source string \(s\) to the the pair containing \(s\) and the empty resource \(\{ || \}\). The \textit{put} and \textit{create} functions simply ignore their resource argument and otherwise behave the same as the corresponding functions from \(l\). Note that the basic lens \(k\) mentioned in the type of \(\hat{l}\) can be an arbitrary basic lens. This will be important for other
combinators such as concatenation and union, whose typing rules stipulate that the basic lens mentioned in the types of their sublenses must be identical.

**Match**  The match combinator handles the processing of strings in chunks. It takes a tag \( t \) and a basic lens \( k \) as arguments and produces a resourceful lens \( \langle t : k \rangle \) that uses \( k \) to process the source and view in each direction. The tag \( t \) controls the \textit{align} function that is used to align chunks in coercing to a basic lens using \([ \cdot ]\).

\[
\frac{t \in T \quad k \in S \iff V}{\langle t : k \rangle \in \langle S \rangle \iff \langle V \rangle}
\]

- \textit{get \( s \)} = \( k \cdot \text{get} \ s \)
- \textit{res \( s \)} = \( \emptyset, \{ 1 \mapsto s \} \)

\[
\begin{align*}
\text{put } v & (\emptyset, r) = \left\{ \begin{array}{ll}
  k \cdot \text{put } v (r(1)) & \text{if } 1 \in \text{dom}(r) \\
  k \cdot \text{create} v & \text{otherwise}
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\text{create } v & \ r = \left\{ \begin{array}{ll}
  k \cdot \text{put } v (r(1)) & \text{if } 1 \in \text{dom}(r) \\
  k \cdot \text{create} v & \text{otherwise}
\end{array} \right.
\end{align*}
\]

**5.3.2 Lemma:** Let \( t \in T \) be a tag and \( k \in S \iff V \) be a basic lens. Then \( \langle t : k \rangle \) is a resourceful lens in \( \langle S \rangle \iff \langle V \rangle \).

The \textit{get} component of match simply passes off control to \( k \). Its \textit{res} function takes a source string \( s \) as input and produces a pair consisting of the rigid complement \( \emptyset \) and the resource \( \{ 1 \mapsto s \} \) as the resource. This forces the \textit{put} function to access all source information through the resource—the rigid complement \( \emptyset \) does not contain any information. There are two cases for \textit{put}: If the resource \( r \) contains an mapping for \( 1 \), then \textit{put} invokes \( l \cdot \text{put} \) on the view and \( r(1) \). Otherwise, it invokes \( l \cdot \text{create} \) on \( v \) and the empty resource \( \{ \} \). The \textit{create} function is identical to \textit{put}.

In examples, we often omit the tag and write “\( \langle k \rangle \)” instead of “\( \langle t : k \rangle \)”. The Boomerang parser silently inserts a default tag that aligns chunks by key, as discussed previously.

**Concatenation**  The next few primitives are resourceful versions of the regular operators. For the most part, these lense are straightforward generalizations of the corresponding basic lenses, but there are many interesting details—e.g., the concatenation and union lenses need to ensure that the resource from its left sublens can be safely passed to its right sublens and vice versa.
Lemma: Let \( l_1 \in S_1 \xleftarrow{C_1 \times t} V_1 \) and \( l_2 \in S_2 \xleftarrow{C_2 \times t} V_2 \) be resourceful lenses such that \( |S_1| \cdot |S_2| \) and \( |V_1| \cdot |V_2| \). Then \((l_1 \cdot l_2)\) is a resourceful lens in \((S_1 \cdot S_2) \xleftarrow{(C_1 \times C_2) \times t} (V_1 \cdot V_2)\).

5.3.3 Lemma: Let \( l_1 \in S_1 \xleftarrow{C_1 \times t} V_1 \) and \( l_2 \in S_2 \xleftarrow{C_2 \times t} V_2 \) be resourceful lenses such that \( |S_1| \cdot |S_2| \) and \( |V_1| \cdot |V_2| \). Then \((l_1 \cdot l_2)\) is a resourceful lens in \((S_1 \cdot S_2) \xleftarrow{(C_1 \times C_2) \times t} (V_1 \cdot V_2)\).

The get function splits the source string in two, applies the get components of its sublenses, and concatenates the results, just like basic concatenation. The res function splits the source string in two and uses \( l_1.res \) and \( l_2.res \) to compute rigid complements \( c_1 \) and \( c_2 \) and resources \( r_1 \) and \( r_2 \). It combines the complements into the pair \((c_1, c_2) \in C_1 \times C_2\) and the resources into the single resource \((r_1 ++ r_2)\). 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 \((r_1 ++ r_2)\) all belong to \( \{N \mapsto k.S\} \). This ensures that we can reorder the resource and pass portions of it to \( l_1 \) and \( l_2 \). The put function splits the view into \( v_1 \) and \( v_2 \), the rigid complement \((c_1, c_2) \) into \( c_1 \) and \( c_2 \), and the resource into \( r_1 \) and \( r_2 \) using split. Recall the equality \( \text{split}(x_1 r_1 ++ r_2) = (r_1, r_2) \), which is essential for ensuring the GetPut law. The create function is similar.

One reason that we require that the same basic lens \( k \) be used for every chunk is to ensure that the resource has a uniform type \( \{N \mapsto k.S\} \), as discussed above. We might be tempted to allow chunks to be processed using different lenses, as long as the lenses had equivalent types, but this would force us to settle for weaker properties. For instance, consider the lens
where $k_1$ and $k_2$ are basic lenses defined as follows:

$$\langle k_1 \rangle \triangleq (\text{copy } \langle ab \rangle) \in \{a, b\} \iff \{a, b\}$$

$$\langle k_2 \rangle \triangleq (a \leftrightarrow b \mid b \leftrightarrow a) \in \{a, b\} \iff \{a, b\}$$

and suppose that we put back the view “aa” (since $k_1$ and $l_2$ are both oblivious, the result of the put does not depend on the rigid complement or the resource). The put function will yield “ab” as the result. Now suppose that we swap the order of chunks in “aa”. Lemma 5.2.4 states that the result produced by put should be “ba”—i.e., the string obtained by swapping the chunks in “ab”. But this is not what happens. Swapping the chunks in “aa” is a no-op, so put produces the very same result, “ab”, as before. Thus, although it is tempting to allow resourceful lenses that use several different lenses to process chunks, we cannot do so without sacrificing properties such as Lemma 5.2.4.

Kleene Star

The Kleene star operator iterates resourceful lenses:

$$l \in S \xleftrightarrow{C,t,k} V \quad |S|^1 \quad |V|^1*$$

$$l^* \in S^* \xleftrightarrow{(C \text{ list}),t,k} V^*$$

get ($s_1 \cdots s_n$) $= (l.\text{get } s_1) \cdots (l.\text{get } s_n)$

res ($s_1 \cdots s_n$) $= [c_1, \ldots, c_n], (r_1++, \ldots, ++ r_n)$

where $c_i, r_i = l.\text{res } s_i$ for $i \in \{1, \ldots, n\}$

put ($v_1 \cdots v_n$) ($c, r$) $= s'_1 \cdots s'_n$

where $s'_i = \begin{cases} l.\text{put } v_i (c_i, r_i) & i \in \{1, \ldots, \min(n, m)\} \\ l.\text{create } v_i r_i & i \in \{m+1, \ldots, n\} \end{cases}$

and $[c_1, \ldots, c_m] = c$

and $r'_0 = r$

and $r_i, r'_i = \text{split}(|v_i|, r'_{i-1})$ for $i \in \{1, \ldots, n\}$

create ($v_1 \cdots v_n$) $r = (l.\text{create } v_1 r_1) \cdots (l.\text{create } v_n r_n)$

where $r'_0 = r$

and $r_i, r'_i = \text{split}(|v_i|, r'_{i-1})$ for $i \in \{1, \ldots, n\}$

5.3.4 Lemma: Let $l \in S \xleftrightarrow{C,t,k} V$ be a resourceful lens such that $|S|^1*$ and $|V|^1*$. Then $l^*$ is a resourceful lens in $S^* \xleftrightarrow{(C \text{ list}),t,k} V^*$.

The get component of the Kleene star lens is identical to the basic lens version. The res function splits the source string into a list of smaller strings and passes each element of this list to $l.\text{res}$. It then merges the rigid complements into a list $[c_1, \ldots, c_n] \in C \text{ list}$ and combines the
resources into a single resource \((r_1 + \ldots + r_n) \in \{N \mapsto k.S\}\). The \textit{put} function splits the view into a list of smaller strings \([v_1, \ldots, v_m]\) and uses the number of chunks in each \(v_i\) and \textit{split} to divide the resource \(r\) into resources \(r_1\) to \(r_m\)—one for each segment of the view. It invokes \textit{l.put} or \textit{l.create} on each \(v_i\) as in the basic lens version.

The Kleene star lens demonstrates why it is important that the \textit{create} function take a resource as an argument, and not just a view. Even when the list of rigid complements contains fewer elements than the list of substrings of the view, the resource may contain items that should be restored to chunks processed by \textit{l.create}. To illustrate, consider the following example:

```haskell
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 convert basic lenses to resourceful lenses as needed). The behavior of \(k.get\) is straightforward:

```haskell
test l.get "Xx,Yy,Zz" = "X,Y,Z"
```

But the behavior of \(k.put\) is more sophisticated: it restores the lower-case letters to each piece of the source by matching up the letters in the old and new new views:

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

To see how this works, let us trace the evaluation of the \textit{put} function in detail. First, note that we are using \(l\) with the interface of a basic lens—i.e., we pass its \textit{put} function strings rather than a string, a rigid complement, and a resource. The Boomerang implementation silently inserts the \([\cdot]\) coercion. Thus, \(1.put\) first uses \(1.get\) to calculate the old view \(x, y, z\) and \(1.res\) to calculate a rigid complement \(c\) and a resource \(r\) from the source string \(\text{x}, \text{y}, \text{z}\):

\[
c = ([\square], [\square, \square]) \\
r = \begin{cases} 
1 \mapsto (X, x) \\
2 \mapsto (Y, y) \\
3 \mapsto (Z, z)
\end{cases}
\]

Next, it aligns the old view “\(x, y, z\)” and new view “\(z, y, w, x\)” using by matching up chunks with the same key. This yields a partial matching \(g\) between chunks in the view, also depicted graphically below:
Next, it applies \( g \) to \( r \), yielding a pre-aligned resource \( r' \):

\[
\begin{align*}
r' = (r \circ g) = \\
\begin{cases}
4 \mapsto (X, x) \\
2 \mapsto (Y, y) \\
1 \mapsto (Z, z)
\end{cases}
\end{align*}
\]

Finally, it invokes the \textit{put} component of \( \llangle \) with the new view, the rigid complement \( c \) and the pre-aligned resource \( r' \). The effect is that the lower-case letters are restored to the chunk containing the corresponding upper-case letter, as in the example above. Notice that even though Kleene star lens uses \( \llangle \).\textit{put} for the third chunk in the updated view and \( \llangle \).\textit{create} for the fourth chunk, the third chunk “\( w \)” is created fresh because the resource \( r' \) is undefined on 3 while the lower-case letter “\( x \)” is restored to the fourth chunk because \( r' \) contains an item for 4.

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

\[
\begin{align*}
|S_1| \cap |S_2| = \emptyset & \quad |V_1| \cap |V_2| \subseteq |V_1 \cap V_2| \\
l_1 \in S_1 & \xrightarrow{C_1 \cdot t \cdot k} V_1 \\
l_2 \in S_2 & \xrightarrow{C_2 \cdot t \cdot k} V_2 \\
(l_1 \vert l_2) & \in (S_1 \cup S_2) \xrightarrow{(C_1 + C_2) \cdot t \cdot k} (V_1 \cup V_2)
\end{align*}
\]

\[
\begin{align*}
\text{get } s &= \begin{cases} l_1.\text{get } s \text{ if } s \in |S_1| \\ l_2.\text{get } s \text{ if } s \in |S_2| \end{cases} \\
\text{res } s &= \begin{cases} \text{Inl}(l_1.\text{res } s) \text{ if } s \in |S_1| \\ \text{Inr}(l_2.\text{res } s) \text{ if } s \in |S_2| \end{cases} \\
\text{put } v(c, r) &= \begin{cases} l_1.\text{put } v(c_1, r) \text{ if } v \in |V_1| \land c = \text{Inl}(c_1) \\ l_2.\text{put } v(c_2, r) \text{ if } v \in |V_2| \land c = \text{Inr}(c_2) \\ l_1.\text{create } v r \text{ if } v \notin |V_1| \land c = \text{Inl}(c_1) \\ l_2.\text{create } v r \text{ if } v \notin |V_2| \land c = \text{Inr}(c_1) \end{cases} \\
\text{create } v r &= \begin{cases} l_1.\text{create } v r \text{ if } v \in |V_1| \\ l_2.\text{create } v r \text{ if } v \notin |V_1| \end{cases}
\end{align*}
\]

**5.3.5 Lemma:** Let \( l_1 \in S_1 \xrightarrow{C_1 \cdot t \cdot k} V_1 \) and \( l_2 \in S_2 \xrightarrow{C_2 \cdot t \cdot k} V_2 \) be resourceful lenses such that \(|S_1| \cap |S_2| = \emptyset\) and \(|V_1| \cap |V_2| \subseteq |V_1 \cap V_2|\). Then \( (l_1 \vert l_2) \) is a resourceful lens in \( (S_1 \cup S_2) \xrightarrow{(C_1 + C_2) \cdot t \cdot k} (V_1 \cup V_2)\).
The `get` component of the resourceful union operator behaves just like the basic lens version. The `res` function selects one of \( l_1\text{.res} \) and \( l_2\text{.res} \) using the source string and puts the resulting rigid complement into a tagged sum, yielding \( \text{Inl}(c) \) in the \( l_1 \) case and \( \text{Inr}(c) \) in the \( l_2 \) case. The resource \( r \) is not tagged. As with concatenation, the typing rule for union stipulates that both branches must use the same basic lens \( k \) to process chunks. This ensures that a resource \( r \) generated using \( l_1\text{.res} \) can be safely passed to \( l_2\text{.put} \) and vice versa. The `put` function selects a branch in an analogous manner to the basic version of union: it first tries to select a branch using the view and falls back to the tag on the rigid complement when the view could be processed by either branch. The side condition mentioned in the typing rule for union requires \([V_1] \cap [V_2] \subseteq [V_1 \cap V_2]\). This ensures that \((V_1 \mid V_2)\) is chunk unambiguous—in particular, that strings in the intersection \((V_1 \cap V_2)\) have unique parses. It rules out languages such as \((a \cdot \langle b \rangle \mid \langle a \rangle \cdot b)\).

Interestingly, using the resource, the union lens is able to pass some source information between branches in the `put` direction. This recovers some of the functionality of “fixup functions” discussed in Chapter 3. To illustrate, consider the following definitions:

```haskell
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 \( k \) that 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:

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

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

**Composition**   The next primitives is a resourceful version of the sequential composition operators. Composition is interesting to consider as a resourceful lens because information can be discarded during both phases of the evaluation of the `get` function, so it must be restored during both phases of the evaluation of `put`. One way to achieve this is by merging the resources produced by each phase of computation into a single structure. This makes it easy to apply alignments calculated for the ultimate view (i.e., the view produced by the second lens)
to the reorderable information represented in the first resource—i.e., it allows alignments to be translated through composite transformations in a natural way.

\[
\begin{align*}
l_1 \in S & \xrightarrow{C_1 \cdot t_1 \cdot k_1} U \quad l_2 \in U \xrightarrow{C_2 \cdot t_2 \cdot k_2} V \\
(l_1; l_2) & \in S \xrightarrow{(C_1 \otimes C_2) \cdot t_2 \cdot (k_1 \cdot k_2)} V
\end{align*}
\]

5.3.6 Lemma: Let \( l_1 \in S \xrightarrow{C_1 \cdot k_1} U \) and \( l_2 \in U \xrightarrow{C_2 \cdot k_2} V \) be resourceful lenses. Then \((l_1; l_2)\) is a resourceful lens in \( S \xrightarrow{(C_1 \otimes C_2) \cdot (k_1 \cdot k_2)} V \).

The \textit{get} function applies \( l_1.get \) and \( l_2.get \) to the source in sequence, just like the basic lens version. The \textit{res} function first applies \( l_1.res \) to the source \( s \), yielding a rigid complement \( c_1 \) and a resource \( r_1 \), and then applies \( l_2.res \) to \((l_1.get \ s)\), yielding another rigid complement \( c_2 \) and resource \( r_2 \). It combines the rigid complements into a pair \((c_1, c_2)\) (we use angle brackets to distinguish them from the pairs produced as rigid complements by the concatenation lens) and merges the resources using \textit{zip}. The resource produced by \textit{zip} represents all of the reorderable information from the source and from the intermediate view in a single structure. In the \textit{put} direction, the composition takes a view \( v \), a rigid complement \((c_1, c_2)\), and a resource \( r \). It first unzips the resource, yielding resources \( r_1 \) and \( r_2 \), and then applies \( l_2.put \) to \( v, c_2, r_2 \), yielding an intermediate view \( u \). To finish the job, it applies \( l_1.put \) to \( u, c_1 \), and \( r_1 \), yielding the final source. The \textit{create} function is similar.

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

\begin{verbatim}
let k1 = del [0-9] . key (copy [A-Z]) . copy [a-z]
let k2 = del [A-Z] . key (copy [a-z])
let l1 = <k1> . (copy COMMA . <k1>)* ;
    <k2> . (copy COMMA . <k2>)*
\end{verbatim}
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 composition phase and the upper-case letter in the second phase:

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

In the `put` direction, it restores the information deleted in each phase:

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

The resource produced by `res` represents the letter and number deleted in each phase of computation together, so that when the resource is aligned using the view, the information from the first phase is also aligned.

The composition lens requires that the view type of `l₁` must be identical to the source type of `l₂`. In particular, it requires that 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 notions of chunks used in 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 `⌊·⌋`, which erases chunks, and then compose them as basic lenses. The tradeoff is that we lose all guarantees about the handling of alignments—although the individual lenses will be resourceful internally, the top-level lens will only be guaranteed to obey the basic lens laws.

**Swap** The final primitive discussed in this chapter is the swap operator. All of the lenses we have seen so far do not permute the order of chunks in going from the source to view—i.e., the `get` function carries the `i`th chunks in the source through to the `i`th chunk in the view. Adding swap breaks this property, which, among other things, causes problems with the composition operator just discussed—if the two phases of computation permute the order of chunks in different ways, then zipping the resources from each phase together and aligning them against the view does not make sense.

To recover the behavior we want in the presence of swap, we need to add a new component to lenses that keeps tracks of the permutation on chunks computed by the `get` function:

```
l.perm ∈ Π s : [S]. Perms(s)
```
It is straightforward to add perm to each of the lenses we have seen so far—e.g., the match lens returns the identity permutation on one chunk while the concatenation operator merges the permutations returned by its sublenses.

We also need to generalize the laws that specify the handling of chunks and resources in the put direction—the old versions no longer hold as stated when lenses may reorder chunks:

\[
\begin{align*}
\text{CHUNKPut} & : \quad x \in (\text{locs}(v) \cap \text{dom}(r)) \quad \Rightarrow \quad (l.\text{perm}(l.\text{put}(c, r)))(y) = x \\
\quad (l.\text{put}(c, r))[y] &= k.\text{put}[x](r(x)) \\
\text{CHUNKCreate} & : \quad x \in (\text{locs}(v) \cap \text{dom}(r)) \quad \Rightarrow \quad (l.\text{create}(v r))[y] = k.\text{put}[x](r(x)) \\
\text{NoChunkPut} & : \quad x \in (\text{locs}(v) - \text{dom}(r)) \quad \Rightarrow \quad (l.\text{put}(c, r))[y] = k.\text{create}(v x) \\
\text{NoChunkCreate} & : \quad x \in (\text{locs}(v) - \text{dom}(r)) \quad \Rightarrow \quad (l.\text{create}(v r))[y] = k.\text{create}(v x)
\end{align*}
\]

With these adjustments to the semantic framework, we can generalize the composition operator as follows. We first revise the res function to permute the resource produced by \(l_1\) using the inverse of the permutation on chunks realized by \(l_2\):

\[
(l_1; l_2).\text{res} s = (c_1, c_2), \text{zip}(r_1 \circ p_2^{-1}) r_2
\]

where

\[
\begin{align*}
& c_1, r_1 = l_1.\text{res} s \\
& c_2, r_2 = l_2.\text{res} u \\
& p_2 = l_2.\text{perm} u \\
& u = l_1.\text{get} s
\end{align*}
\]

This has the effect of putting the resource computed by \(l_1\) into the view order of \(l_2\). We also revise the definition of put to apply the permutation on chunks realigned by \(l_2\) to resource we pass to \(l_1\):

\[
(l_1; l_2).\text{put} v ((c_1, c_2), r) = l_1.\text{put} u (c_1, r_1 \circ p)
\]

where

\[
\begin{align*}
& r_1, r_2 = \text{unzip} r \\
& u = (l_2.\text{put}(v (c_2, r_2)) \\
& p_2 = l_2.\text{perm} u
\end{align*}
\]
This puts the items in the resource back into the view order of \( l_1 \).

Now we turn to the resourceful version of the swap lens itself:

\[
\begin{align*}
|S_1|_1|S_2| & \quad |V_2|_1|V_1| \\
|l_1 \in S_1 \xrightarrow{C_{1,t,k}} V_1| & \quad |l_2 \in S_2 \xrightarrow{C_{2,t,k}} V_2| \\
(l_1 \sim l_2) \in (S_1 \cdot S_2) \xrightarrow{(C_{2 \times C_1}) \cdot t \cdot k} (V_2 \cdot V_1)
\end{align*}
\]

\[\begin{align*}
\text{get} (s_1 \cdot s_2) &= (l_2 \text{.get} s_2) \cdot (l_1 \text{.get} s_1) \\
\text{res} (s_1 \cdot s_2) &= (c_2, c_1), (r_2 ++ r_1) \\
&\quad \text{where } c_1, r_1 = l_1 \text{.res } s_1 \\
&\quad \text{and } c_2, r_2 = l_2 \text{.res } s_2 \\
\text{perm} (s_1 \cdot s_2) &= (l_2 \text{.perm } s_2) \ast (l_1 \text{.perm } s_1) \\
\text{put} (v_2 \cdot v_1) (c, r) &= (l_1 \text{.put } v_1 (c_1, r_1)) \cdot (l_2 \text{.put } v_2 (c_2, r_2)) \\
&\quad \text{where } c_1, c_2 = c \\
&\quad \text{and } r_2, r_1 = \text{split}(|v_2|, r) \\
\text{create} (v_2 \cdot v_1) r &= (l_1 \text{.create } v_1 r_1) \cdot (l_2 \text{.create } v_2 r_2) \\
&\quad \text{where } r_1, r_2 = \text{split}(|v_2|, r)
\end{align*}\]

5.3.7 Lemma: Let \( l_1 \in S_1 \xrightarrow{C_{1,t,k}} V_1 \) and \( l_2 \in S_2 \xrightarrow{C_{2,t,k}} V_2 \) be lenses with \( |S_1|_1|S_2| \) and \( |V_1|_1|V_2| \). Then \( (l_1 \sim l_2) \) is a resourceful lens in \( (S_1 \cdot S_2) \xrightarrow{(C_{2 \times C_1}) \cdot t \cdot k} (V_2 \cdot V_1) \).

The get, res, put, and create components are all straightforward generalizations of the concatenation lens. The perm component of swap merges the permutations \( q_1 \) and \( q_2 \) computed using \( l_1 \) and \( l_2 \) using the \((\ast \ast)\) operator, which behaves as follows:

\[
(q_2 \ast \ast q_1)(i) = \begin{cases} 
q_1(i) + |q_2| & \text{if } i \leq |q_1| \\
q_2(i - |q_1|) & \text{otherwise}
\end{cases}
\]

5.4 Alignments

So far, our discussion of resourceful lenses has mostly focused on the mechanisms for specifying chunks and for using alignment information to propagate updates to views back to sources. We have not, however, said much about where alignments actually come from, except to say that they are controlled by the tag argument to the match combinator. In this section, we describe the alignment functions we have implemented in Boomerang, we show how to combine multiple alignment policies in the same lens, and we describe mechanisms for tuning alignments using programmer-specified notions of keys and thresholds.
Keys

In many situations, we do not want to consider all of the information contained in the chunks when we compute an alignment. For example, consider the following definitions:

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

The `get` function takes a list of comma-sparated chunks containing a number, an upper-case letter, and a lower-case letter, deletes the number, and copies the letters. The `put` function uses the default alignment policy which, perhaps surprisingly, aligns chunks by position in this case:

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

The reason that it uses positional alignment is that the default alignment policy only considers the portions of chunks that are designated as “keys” when computing an alignment, and the `k` lens does not designate any keys—i.e., every chunk has the empty string as its key. Here is an alternate version of the lens that uses the upper-case letter as the key for the enclosing chunk:

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

If we evaluate the same `put` as above, the numbers will be restored to each chunk by key instead by position:

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

Note that we have not designated the lower-case letter as a key. This means that we can edit the lower-case letters without changing the alignment:

```haskell
test l.put "Cx,By,Az" into "1Aa,2Bb,3Cc" = "3Cx,2By,1Az"
```

The ability to designate some, all, or none of the view as a key is a simple but important feature that we use often examples.

Our implementation has two primitives for specifying keys. The `key` combinator takes a lens as an argument and designates the entire view as a key while the `nokey` combinator designates none of the view as a key. Semantically, these combinator do not change the behavior of the lenses they enclose—i.e., outside of a chunk, the lenses `k`, `key k`, and `nokey k` all behave the same. Instead, they add (or remove) annotations to the type of the view that indicate which parts of the view should be counted as a part of the key. Before we align two lists of chunks, we use the annotations on the type of the view to calculate a key for each chunk. We then align the lists of chunks using keys instead of the actual contents.
Species

Because alignment is a fundamentally heuristic operation, the ability to select between different alignment strategies is important in practice. One of the strengths of the resourceful lens framework is that correctness does not hinge on special properties of the alignment function—the only property we require is that return the identity alignment whenever its arguments are identical. This flexibility gives us enormous latitude in designing alignment functions to use with resourceful lenses. Our implementation currently supports several different alignment functions:

- **Positional**: The alignments matches up chunks by position in the two lists of chunks. If one list has more chunks than the other, the extra items at the end of the list are not aligned.

- **Bush-like**: The alignment minimizes a cost function defined as the sum of the edit distances between the keys of aligned chunks and the sizes of the keys of unaligned chunk.

- **Diff-like**: The alignment minimizes the same cost function as the bush-like heuristic, but does not consider alignments with “crossing” edges. This heuristic corresponds to the longest common subsequence, so it can be implemented quite efficiently.

- **Operation-based**: The alignment is obtained from the actual update operation performed to the view.

(Strictly speaking, the operation-based strategy is not an alignment function; we include it in this list because alignments can also be provided to lenses directly.) These strategies are depicted graphically in Figure 5.2, and the first three strategies are illustrated in the following examples:
let l = key \[A-Z\] . del \[0-9\]
test <pos : l>*.put "BCA" into "A1B2C3" = "B1C2A3"
test <dif : l>*.put "BCA" into "A1B2C3" = "B2C3A0"

To select a particular “species” to use to align chunks, programmers supply the appropriate tag to the match combinator—e.g., the \(\text{pos}\) tag selects positional alignment, \(\text{bsh}\) uses bush-like, and \(\text{dif}\) uses diff-like. When combined with \(\text{key}\) annotations, these species allow programmers to specify many useful alignment policies.

### Multiple Chunks

All of the resourceful lenses we have seen so far all have a single kind of chunk. However, it is often convenient to define resourceful lenses that operate on several different kinds of chunks where the alignment of different kinds of chunks is handled separately in the \(\text{put}\) direction. As an example, consider the following definition, which introduces a lens with two different kinds of chunks identified by tags.

```haskell
let INT : regexp = \[0-9\]+ let FLOAT : regexp = INT . "." . INT
let l1 : lens = key \[A-Z\] . ("=" . INT) <-> ":int"
let l2 : lens = key \[A-Z\] . ("=" . FLOAT) <-> ":float"
let l : lens = <tag "t1" : l1> | <tag "t2" : l2>
let k : lens = l . (copy "," . l)*
```

In the \(\text{get}\) direction, \(k\) takes a non-empty list of comma-separated chunks containing either an letter and an integer or a letter and a floating point number, copies the letter, deletes the number, and inserts a string recording whether the chunk contained an integer or a float:

```haskell
test k.get "A=1,A=1.0,B=2,B=0.5" = "A:int,A:float,B:int,B:float"
```

In the \(\text{put}\) direction, the tags \(t1\) and \(t2\) specify that the lens should handle integer and floating point chunks separately, even when their keys overlap. For example, if we reorder the chunks so that all of the floating point chunks come before the integer chunks

```haskell
test k.put "A:float,B:float,A:int,B:int" into "A=1,A=1.0,B=2,B=0.5" = "A=1.0,B=0.5,A=1,B=2"
```

the lens restores the numbers to each chunk. If we had defined \(k\) using only one kind of chunk, the behavior of \(\text{put}\) would be different, aligning chunks by key in the \(\text{put}\) direction but ignoring the distinction between integer chunks and float chunks:

```haskell
let l' : lens = < 11 | 12 >
let k' : lens = l' . (copy "," . l')*
```

```haskell
test k'.put "A:float,B:float,A:int,B:int" into "A=1,A=1.0,B=2,B=0.5" = "A=0.0,B=0.0,A=0,B=0"
```
This results in integer chunks being matched up with float chunks (and vice versa), so instead of numbers from the old source being restored to the new source, each of the new source chunks are created with defaults.

Generalizing the type system to accommodate tags is straightforward: rather than having a single basic lens in its type, each lens has a finite map from tags to basic lenses in its type (i.e., a lens for each tag). We change the typing rules for combinators such as union and concatenation, to check that the maps for the sublenses agree on the basic lens on all tags where both maps are defined. Likewise, we generalize resources from finite maps from locations to strings to finite maps from tags to maps from locations to strings. Finally, we compute alignments tag-wise—i.e., for each tag, we use the alignment function corresponding to the species of the chunk under that tag.

**Thresholds**

Some of the alignment strategies just discussed—in particular, set-like and diff-like—match up chunks by minimizing the global cost of an alignment. These strategies work well in cases where the keys of chunks have been edited. However, in many situations, it is important to not match up chunks that are “too different”, even if aligning those chunks contributes to the minimal cost alignment. For example, in the following program, where the keys are three characters long

```plaintext
let l : lens = key [A-Z]{3} . del [0-9]
test <bsh : l>*.put "DBDCCCAAA" into "AAA1BBB2CCC3" = "DBD2CCC3AAA1"
```

we might prefer to create the chunk for DBD rather than matching it up with the BBB2 source chunk. Unfortunately, the bush-like heuristic does align these chunks because the cost of a two-character edit is less than the cost of a newly created chunk. To achieve the behavior we want, we tune the bush-like species with a threshold annotation, as illustrated in the following example:

```plaintext
let l : lens = key [A-Z]{3} . del [0-9]
test <$sim 50 : l>*.put "DBDCCCAAA" into "AAA1BBB2CCC3" = "DBD0CCC3AAA1"
```

The \$sim species takes an integer \(n\) as an argument. It behaves like bush-like, but only matches chunks whose keys have not been edited by more than \(\left(100 - n\right)\)%—e.g., the \$bsh species is actually defined as \($\sim 0\$\). In this example, the lens does not align DBD with the BBB2 from
the source because the view is more than 50\% different. The diff-like species also supports thresholds. Tuning alignment functions using thresholds controls the behavior of the heuristic, and makes it more predictable—e.g., we often use it when we need to align totally unstructured blocks of text.

5.5 Summary

Resourceful lenses extend the simple mechanisms of basic lenses with new constructs for handling ordered data. These features make it possible to handle situations where the update to the view involves a reordering. Semantically, we redesign the architecture of lenses to separate the handling of rigidly ordered and reorderable data and we add new laws ensuring that the components of lenses use alignment information correctly. This design allows us to instantiate resourceful lenses with arbitrary alignment heuristics. 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, and we add new primitives for specifying and tuning alignment policies.
Chapter 6

Secure Lenses

“One who wishes to keep a secret must hide the fact that he possesses one.”

—Johannes Wolfgang von Goethe

One of the primary uses of views in databases is as a mechanism for controlling access to sensitive information. By forcing users to access data sources via a security view that only exposes public information, data administrators ensure that secrets will not be leaked, even if the users mishandle the view or are even malicious. Security views are a robust mechanism, making it impossible for users to expose the source data hidden by the view, and they are flexible: since they are implemented as arbitrary programs, they can be used to enforce extremely 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, a collaborative data sharing system based on Wikipedia 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

1Strictly speaking, the user of the view may still be able to gain some knowledge of the hidden parts of the source by examining the view (Miklau and Suciu, 2007)—i.e., views do not provide privacy—but they cannot access any of data hidden by the view directly.
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), we would like the users of these views to be able to make updates—e.g., to correct errors or add new information—and have their changes be propagated back to the original document.

Unfortunately, all of 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 would be to delete the corresponding section of the original document. But while doing so faithfully reflects the edit made to the view—formally, it satisfies the \texttt{PVG} \texttt{E} law—it is not necessarily what we want: if the section in the original document contains additional classified data in nested subsections, then deleting the section is almost surely unacceptable—users 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 lenses obey the \texttt{PVP} \texttt{PUT} law. However, as discussed in Chapter 3, requiring that every lens be very well behaved is a draconian restriction that rules out many transformations needed in practice—e.g., union and Kleene star.

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 develop a new semantic space of \textit{secure lenses}, in which the types of lenses not only describe the sets of structures manipulated by the components of lenses, but also capture the notion that certain parts of the source and view represent endorsed 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 endorsed 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 a non-interference property for the put function as an additional behavioral law. This law ensures that if the update to the view does not change high-integrity data in the view then the put function does not modify high-integrity data in the source.

We then develop security-enhanced variants of each of our string lens combinators. These 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, since regular expressions describe data schemas at a high level of precision.

Of course, confidentiality is also interesting in the context of security views: 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 for Boomerang. Thus, our actual type system tracks flows of information in two directions, ensuring confidentiality in the forward direction and integrity in the reverse direction.

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 final technical section of this chapter, we extend secure lenses with dynamic checks that allow us to test for and detect these situations. These lenses use a combination of static types and dynamic tests to establish the same essential security properties and, in many cases, can be assigned more flexible types.

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 framework for building reliable and updatable security views.

2. We design the syntax and semantics of \textit{security-annotated regular expressions}, which describe sets of strings as well as equivalence relations that encode confidentiality and integrity policies.

3. We reinterpret each of our \textit{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.

\section{Example}

To warm up, let us examine 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 string is an electronic calendar in which certain appointments, indicated by “*”, are intended to be private.

\begin{quote}
*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)
\end{quote}

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 \textit{BUSY} and, at the same time, we also want to erase the location of every appointment.

\begin{quote}
08:30 BUSY  
10:00 Meeting with Brett  
12:00 PLClub Seminar  
15:00 BUSY
\end{quote}

Or, perhaps, we want to go a step further and erase private appointments completely.

\begin{quote}
10:00 Meeting with Brett  
12:00 PLClub Seminar
\end{quote}

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.
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 P.posterG. 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 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.
As these examples demonstrate, to manage security views using lenses reliably, we need 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 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 in Boomerang as follows:

```plaintext
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 endorsed by decorating the bare regular expressions with annotations. Here is a type in which the private appointments are endorsed, as indicated by annotations of the form $(R:E)$, but the public appointments are tainted, as indicated by annotations of the form $(R:T)$:

```plaintext
( (SPACE·TIME·DESC·LOCATION·NEWLINE): T
| (ASTERISK·TIME·DESC·LOCATION·NEWLINE): E)*
```

As described in the next section, 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 endorsed data. In this case, since the private appointments are endorsed, 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:

\[
((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOCATION}\cdot\text{NEWLINE}) \cdot T | (\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOCATION}\cdot\text{NEWLINE}) \cdot T)^*)
\]

\[
<(\text{TIME}\cdot\text{DESC}\cdot\text{NEWLINE}) | (\text{TIME}\cdot\text{BUSY}\cdot\text{NEWLINE})>: T)^*
\]

With this type, 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.

```boomerang
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. 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 \(l\dot{\text{put}}\) to pairs of these in order. We defer an investigation of secure resourceful lenses to future work.

Here is an example of the sort of property we will be able to show using the secure lens framework developed in this chapter:
6.1.1 Lemma: The redact lens is a secure lens at the following type:

\[ ((\text{SPACE}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOCATION}\cdot\text{NEWLINE})\cdot\text{T}) \mid (\text{ASTERISK}\cdot\text{TIME}\cdot\text{DESC}\cdot\text{LOCATION}\cdot\text{NEWLINE})\cdot\text{E})^* \]

\[ \iff ((\text{TIME}\cdot\text{DESC}\cdot\text{NEWLINE})\cdot\text{T} \mid (\text{TIME}\cdot\text{BUSY}\cdot\text{NEWLINE})\cdot\text{E})^* \]

6.2 Semantics

The basic lenses ensure some fundamental sanity conditions on the handling of data in the source and view, but, as we saw in the preceding section, to use 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—stipulating that the put function must not taint endorsed (high integrity) source data and the get function must not leak secret (high confidentiality) data—see Figure 6.1.

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 mostly work with two-point lattices \( \mathcal{P} = \{\mathcal{P}, \mathcal{S}\} \) (for “public” and “secret”) with \( \mathcal{P} \sqsubseteq \mathcal{S} \) and \( \mathcal{Q} = \{\mathcal{E}, \mathcal{T}\} \) (for “endorsed” and “tainted”) with \( \mathcal{E} \sqsubseteq \mathcal{T} \).

\[
\mathcal{P} = \begin{cases} \mathcal{S} \\ \mathcal{P} \end{cases} \quad \mathcal{Q} = \begin{cases} \mathcal{T} \\ \mathcal{E} \end{cases}
\]
Although we call endorsed data “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.) Our results generalize to arbitrary finite lattices in a straightforward way.

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, \( \sim \) and \( \approx \) will be equality, while \( \sim \) and \( \approx \) will be coarser relations that identify sources and views containing the same public and endorsed parts, respectively. These equivalences capture confidentiality and integrity policies for the data.

6.2.1 Definition [Secure Lens]:  
A secure lens \( l \) has the same components as a basic lens

\[
\begin{align*}
  l.\text{get} &\in S \rightarrow V \\
  l.\text{put} &\in V \rightarrow S \rightarrow S \\
  l.\text{create} &\in V \rightarrow S
\end{align*}
\]

that obey the following laws for every \( s \) in \( S \), \( v \) in \( V \), and \( k \) in \( \mathcal{Q} \) or \( \mathcal{P} \) as appropriate:

\[
\begin{align*}
  l.\text{get} (l.\text{put} v s) &= v & \text{(PutGet)} \\
  l.\text{get} (l.\text{create} v) &= v \quad & \text{(CreateGet)} \\
  v \approx_k l.\text{get} s \quad & \text{(GetPut)} \\
  l.\text{put} v s \approx_k s \\
  s \sim_k s' \quad & \text{(GetNoLeak)}
\end{align*}
\]

The \text{PutGet} and \text{CreateGet} laws here are identical to the basic lens version that we saw in Chapter 3 and express the same fundamental constraint: updates to views must be translated exactly.
The GetPut law for secure lenses, however, is different. It ensures the integrity of source data, expressed as a non-interference condition on the put function. Formally, it requires that if the original view (i.e., the one computed from the original source) and the new view 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 new view are related by \( \approx_E \) — i.e., they agree on the endorsed data — then GetPut guarantees that the new source will also agree with the original source on endorsed data. Note that we recover the basic lens law GetPut when \( \approx_k \) is equality, as it typically is for \( \approx_T \).

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 \( Q \) 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 \( \approx_E \). The owner of the source only performs the put if the test succeeds.

Secure lenses obey a variant of the PutPut law capturing a notion of lenses that are very well behaved on endorsed data:

**6.2.2 Lemma:** Secure lenses admit the following inference rule:

\[
\begin{align*}
    v' \approx_k l.\text{get } s & \approx_k v \\
    l.\text{put } v' (l.\text{put } v \ s) & \approx_k l.\text{put } v' \ s
\end{align*}
\]

(PutPutEndorsed)

If \( \approx_k \) is equality (as it typically is for \( \approx_T \)) then PutPutEndorsed reduces to the GetPut law: it says that applying put (twice) to the view obtained by invoking get on the source yields the original source (both times). If, however, \( \approx_k \) relates strings that agree on endorsed data (as it typically does for \( \approx_E \)) then PutPutEndorsed implies that put must preserve the endorsed 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 may be discarded as 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 $\sim_\mathcal{P}$ relates two sources, GetNoLeak ensures that the views computed from those sources also agree on public data. Secure lenses provide a confidentiality guarantee that can be understood without having to look at the lens program.\footnote{We treat confidentiality and integrity as orthogonal—almost, see Section 6.5—so users can also choose $\sim_\mathcal{P}$ 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.} In the next section, we present a declarative language for security annotations that can be used to describe many such equivalences.

### 6.3 Annotated Regular Expressions

The types of our secure string lens combinators are regular expressions annotated with labels drawn from the two lattices of security labels. In this section, we define the precise syntax and semantics of these 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’ll 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.

#### 6.3.1 Definition [Security-Annnotated Regular Expression]:

The set of 

security-annotated regular expressions

over $\Sigma$ and $\mathcal{K}$ is the smallest set generated by the following grammar

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

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 \( \sim_k \subseteq (L(R) \times L(R)) \) 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 \( L(R) \)—e.g., we will write \( u \in R \) instead of \( u \in L(R) \).

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 and 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, \( \text{hash}(R) \) the function that transforms strings in \( L(R) \) by rewriting every character to \( # \),

\[
\text{hash}(R)(u) \triangleq \underbrace{# \cdots #}_{\text{\# times}}
\]

and \( \text{mark}(R, k) \) a relation that marks inaccessible characters:

\[
\begin{align*}
\text{mark}(\emptyset, k) & \triangleq \{\} \\
\text{mark}(u, k) & \triangleq \{(u, u)\} \\
\text{mark}(R_1 \cdot R_2, k) & \triangleq \text{mark}(R_1, k) \cdot \text{mark}(R_2, k) \\
\text{mark}(R_1 | R_2, k) & \triangleq \text{mark}(R_1, k) \& (L(R_1) - L(R_2)) \\
& \quad \cup \text{mark}(R_2, k) \& (L(R_2) - L(R_1)) \\
& \quad \cup \text{mark}(R_1, k) \& \text{mark}(R_2, k) \\
\text{mark}(R_1^*, k) & \triangleq \text{mark}(R_1, k)^* \\
\text{mark}(R_1; j, k) & \triangleq \begin{cases} \text{mark}(R_1, k) & \text{if } k \supseteq j \\ \text{hash}(R_1) & \text{otherwise} \end{cases}
\end{align*}
\]
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 \( L \):

\[ Q \cap L \equiv \{(u, v) \mid (u, v) \in Q \land u \in L\} \]

To handle strings described by both expressions, we use an intersection operator that merges markings

\[ Q_1 \cap Q_2 \equiv \{(u, \text{merge}(v_1, v_2)) \mid (u, v_i) \in Q_i\}, \]

where:

\[
\begin{align*}
\text{merge}(\epsilon, \epsilon) &= \epsilon \\
\text{merge}(\#, v_1, \# - v_2) &= \# \cdot \text{merge}(v_1, v_2) \\
\text{merge}(-v_1, \# - v_2) &= \# \cdot \text{merge}(v_1, v_2) \\
\text{merge}(c \cdot v_1, c \cdot v_2) &= c \cdot \text{merge}(v_1, v_2).
\end{align*}
\]

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:

6.3.2 Definition [Well-Formed Security-Annotated Regular Expression]: \( R \) is well formed iff every subexpression of the form \( R_1 \cdot R_2 \) is unambiguously concatenable \((L(R_1), L(R_2))\) and every subexpression of the form \( R^* \) is unambiguously iterable \((L(R)^\dagger)\).

6.3.3 Proposition: If \( R \) is well formed, then \( \text{mark}(R, k) \) is a function.

In what follows, we will tacitly assume that all annotated expressions under discussion are well formed. (And when we define typing rules for our secure lens combinators, below, we will be careful to ensure well-formedness.)
Let \( \text{erase} \) be the function on \( (\Sigma \cup \{\#\}) \) that copies characters in \( \Sigma \) and erases \# symbols. We define \( \sim_k \) as the relation induced by marking and then erasing:

\[
\begin{align*}
\text{hide}_k(u) & \triangleq \text{erase}(\text{mark}(R, k)(u)) \\
\sim_k & \triangleq \{(u, v) \mid \text{hide}_k(u) = \text{hide}_k(v)\}
\end{align*}
\]

It is easy to see that \( \sim_k \) is an equivalence relation.

6.3.4 Lemma: Let \( R_1 \) and \( R_2 \) be well-formed annotated regular expressions over a finite lattice \( K \). It is decidable whether \( R_1 \) and \( R_2 \) are equivalent.

Proof sketch: Equivalence for the regular languages \( L(R_1) \) and \( L(R_2) \) is straightforward. Moreover, each relation \( \sim_k \) is induced by \( \text{hide}_k(\cdot) \), which is definable as a rational function—a class for which equivalence is decidable (Berstel, 1979, Chapter IV, Corollary 1.3).

As examples to illustrate the semantics, consider a two-point lattice \( (\{P, S\}, \sqsubseteq) \) with \( P \sqsubseteq S \). and take \( R_1 \) to be the annotated expression \([a-z]:S\). Then for every string \( u \) in \( L(R_1) \) we have \( \text{mark}(R_1, P)(u) = \# \), and so \( \text{hide}_P(u) = \epsilon \), and \( \sim_P \) is the total relation. For the annotated relation \( R_1^* \), the equivalence \( \sim_P \) is again the total relation because every \( u \) in \( L(R_1^*) \) maps to a sequence of \# symbols by \( \text{mark}(R_1^*, P) \), and so \( \text{hide}_P(u) = \epsilon \). More interestingly, for \( R_2 \) defined as

\[
([a-z]:P)\cdot([0-4]:S) \mid ([a-z]:P)\cdot([5-9]:S),
\]

and any string \( c\cdot n \) in \( L(R_2) \) we have \( \text{mark}(R_2, P)(c\cdot n) = c\# \) and so \( \text{hide}_P(c\cdot n) = c \). It follows that \( cn \sim_P c'n \) iff \( c = c' \). Finally, for \( R_2^* \) the equivalence \( \sim_P \) identifies \( (c_1\cdot n_1 \cdots c_i\cdot n_i) \) and \( (d_1\cdot m_1 \cdots d_j\cdot m_j) \) iff \( i = j \) 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 \( \text{mark} \) 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 \( \text{mark} \) 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 the rest of this chapter.

119
6.4 Secure Lens Combinators

Having identified the semantic space of secure lenses and defined the syntax and semantics of annotated regular expressions, we now turn to syntax, developing defining secure versions of our core set of lens combinators. The functional components of these secure lenses are identical to their basic lens versions, but their typing rules are enhanced with an information-flow analysis that guarantees the secure lens laws.

Copy The simplest lens, \( \text{copy } E \), takes a well-formed annotated regular expression as an argument. It copies strings belonging to \( E \) in both directions.

\[
\begin{align*}
\text{E well-formed} \\
\text{copy } E \in E \leftrightarrow E
\end{align*}
\]

\[
\begin{align*}
\text{get } s & = s \\
\text{put } v s & = v \\
\text{create } v & = v
\end{align*}
\]

6.4.1 Lemma: Let \( E \in \mathcal{R} \) be a security-annotated regular expression. Then \( \text{(copy } E \text{)} \) is a secure lens in \( E \leftrightarrow E \).

The \( \text{(copy } E \text{)} \) lens trivially obeys the secure lens laws because the equivalence relations for confidentiality and integrity are identical for the source and view.

Const The \( \text{const} \) 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 the unique element of \( F \) in the \( \text{get} \) direction, and restores the discarded source string in the reverse direction.

\[
\begin{align*}
\text{E, F well-formed} & \quad |\mathcal{L}(F)| = 1 \\
\text{const } E F \in E \leftrightarrow F
\end{align*}
\]

\[
\begin{align*}
\text{get } s & = \text{representative}(F) \\
\text{put } v s & = s \\
\text{create } v & = \text{representative}(E)
\end{align*}
\]

6.4.2 Lemma: Let \( E \) and \( F \) be well-formed security-annotated regular expressions such that \( |\mathcal{L}(F)| = 1 \). Then \( \text{(const } E F \text{)} \) is a secure lens in \( E \leftrightarrow F \).
Typically $F$ will just be a bare string $u$, but occasionally it will be useful to decorate it with integrity labels (e.g., see the discussion following the union combinator below). The typing rule for $\text{const}$ places no additional labels on the source and view types. This is safe: the $\text{get}$ function maps every string in $E$ to $\text{representative}(F)$, so $\text{GetNoLeak}$ holds trivially. The $\text{put}$ restores the source exactly—including any high-integrity data—so $\text{GetPut}$ also holds trivially.

**Union**  The union combinator uses some new notation, which is explained below.

\[
\begin{align*}
(S_1 \cap S_2) &= \emptyset \\
l_1 \in S_1 \iff V_1 & \quad l_2 \in S_2 \iff V_2 \\
q &= \bigvee \{k \mid \text{k min obs. } V_1 \neq V_2 \land V_1 \land V_2 \text{ agree} \} \\
p &= \bigvee \{k \mid \text{k min obs. } (S_1 \cap S_2) = \emptyset \} \\
(l_1 \mid l_2) &\in (S_1 \mid S_2) : q \iff (V_1 \mid V_2) : p
\end{align*}
\]

\[
\begin{align*}
\text{get } s &= \begin{cases} 
  l_1.\text{get } s & \text{if } s \in S_1 \\
  l_2.\text{get } s & \text{if } s \in S_2
\end{cases} \\
\text{put } v \ s &= \begin{cases} 
  l_1.\text{put } v \ s & \text{if } s \in S_1 \land v \in V_1 \\
  l_2.\text{put } v \ s & \text{if } s \in S_2 \land v \in V_2 \\
  l_1.\text{create } v & \text{if } s \in S_2 \land v \in (V_1 - V_2) \\
  l_2.\text{create } v & \text{if } s \in S_1 \land v \in (V_2 - V_1)
\end{cases} \\
\text{create } v &= \begin{cases} 
  l_1.\text{create } v & \text{if } v \in V_1 \\
  l_2.\text{create } v & \text{if } v \in (V_2 - V_1)
\end{cases}
\end{align*}
\]

**6.4.3 Lemma:** Let $l_1 \in S_1 \iff V_1$ and $l_2 \in S_2 \iff V_2$ be secure lenses such that $(S_1 \cap S_2) = \emptyset$. Then $(l_1 \mid l_2)$ is a secure lens in $(S_1 \mid S_2) : q \iff (V_1 \mid V_2) : p$ where $q = \bigvee \{k \mid \text{k min obs. } V_1 \neq V_2 \land V_1 \land V_2 \text{ agree} \}$ and $p = \bigvee \{k \mid \text{k min obs. } (S_1 \cap S_2) = \emptyset \}$.

Like the basic lens version, the union lens uses a membership test on the source string to select a lens in the $\text{get}$ 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:

\[
\begin{align*}
l_1 & \triangleq [0-4]: S \leftrightarrow A \in ([0-4]: S) \iff A \\
l_2 & \triangleq [5-9]: S \leftrightarrow B \in ([5-9]: S) \iff B
\end{align*}
\]

121
We might be tempted to assign it the type obtained by taking the unions of the source and view types of the smaller lenses:

\[(l_1 \mid l_2) \in ([0-4]:S \mid [5-9]:S) \xleftrightarrow{\mathit{get}} (A \mid B)\]

But this would be wrong: the \texttt{get} function leaks information about which branch was selected, as demonstrated by the following counterexample to \texttt{GetNoLeak}. By the semantics of annotated regular expressions, we have \(0 \sim_P 5\), since \texttt{hide}_P\texttt{maps both to the empty string. But:}

\[(l_1 \mid l_2).\texttt{get} 0 = A \not\sim_B B = (l_1 \mid l_2).\texttt{get} 5\]

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 \((l_1 \mid l_2)\) and the two-point lattice, \(S\) 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 \(P\) is finite). Let \(k\) be a label in \(P\). We say that \(k\) \textit{observes} \((S_1 \cap S_2) = \emptyset\) iff 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 \(k\) \textit{observes} \((S_1 \cap S_2) = \emptyset\) iff the codomains of the rational function \(\text{hide}_k(\cdot)\) 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 \textit{put} direction, the union lens selects a lens using membership tests on the source and the view (the test on the view takes priority, with the test on the source breaking ties). Here we need to consider the integrity of the source data, since modifying the view can result in \(l_2\) being used for the \texttt{put} function even though \(l_1\)’s \texttt{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:

\[ l_1 \triangleq (\text{del } [0-4]: E) \cdot (\text{copy } [A-Q]: T) \in ([0-4]: E \cdot [A-Q]: T) \]

\[ l_2 \triangleq (\text{del } [5-9]: E) \cdot (\text{copy } [F-Z]: T) \in ([5-9]: E \cdot [F-Z]: T) \]

This lens does not have secure lens type obtained by taking the union of the source and view types

\[ (l_1 \mid l_2) \in ([0-4]: E \cdot [A-Q]: T) \mid ([5-9]: E \cdot [F-Z]: T) \]

\[ \iff ([A-Q]: T \mid [F-Z]: T) \]

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 \( z \approx_{E} A \), since \( \text{hide}_{E} \) maps both to the empty string. But

\[ (l_1 \mid l_2).\text{put } Z 0A = 5Z \not{\approx}_{E} 0A \]

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 \( Q \) that observe that \( V_1 \) and \( V_2 \) are not identical—e.g., for the lens above, \( T \). For technical reasons—to ensure that \( v \in V_1 \) and \( s \in S_1 \) and \( v \approx_{k}^{(S_1 | S_2)} (l_1 \mid l_2).\text{get } s \) implies \( v \approx_{k}^{S_1} l_1.\text{get } s \)—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. Then, since both lenses are capable of handling the entire view type, the same lens is always selected for put as was selected for get. For example, the union of

\[ l_1 \triangleq (\text{del } [0-4]: E) \cdot (\text{copy } [A-Z]: T) \in ([0-4]: E \cdot [A-Z]: T) \iff ([A-Z]: T) \]

\[ l_2 \triangleq (\text{del } [5-9]: E) \cdot (\text{copy } [A-Z]: T) \in ([5-9]: E \cdot [A-Z]: T) \iff ([A-Z]: T) \]

123
does have the type:

\[ ([0-4]:E\cdot[A-Z]:T) \mid ([5-9]:E\cdot[A-Z]:T) \leftrightarrow [A-Z]:T \]

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 \( E \). Annotating with \( E \), the least element in \( Q \), is semantically equivalent to having no annotation at all.

**Concatenation**  Somewhat surprisingly, the concatenation lens also has an interesting typing rule as a secure lens.

\[
\begin{align*}
  l_1 &\in S_1 \leftrightarrow V_1 \quad S_1 \downarrow S_2 \\
  l_2 &\in S_2 \leftrightarrow V_2 \quad V_1 \downarrow V_2 \\
  q &= \bigvee \{ k \mid k \text{ min obs. } V_1 \downarrow V_2 \} \\
  p &= \bigvee \{ k \mid k \text{ min obs. } S_1 \downarrow S_2 \} \\
  l_1 \cdot l_2 &\in (S_1 \cdot S_2):q \leftrightarrow (V_1 \cdot V_2):p
\end{align*}
\]

![Concatenation Typing Rule](image)

**6.4.4 Lemma:** Let \( l_1 \in S_1 \leftrightarrow V_1 \) and \( l_2 \in S_2 \leftrightarrow V_2 \) such that \( S_1 \downarrow S_2 \) and \( V_1 \downarrow V_2 \). Then \( (l_1 \cdot l_2) \) is a secure lens in \( (S_1 \cdot S_2):q \leftrightarrow (V_1 \cdot V_2):p \) where \( q = \bigvee \{ k \mid k \text{ min obs. } V_1 \downarrow V_2 \} \) and \( p = \bigvee \{ k \mid k \text{ min obs. } S_1 \downarrow S_2 \} \).

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 \( a_0 \) to \( \lambda \) and \( a_1 \) to \( a \), and a lens \( l_2 \) that maps \( b_0 \) to \( b \) and \( b_1 \) to \( b \), where all of the source data is private except for the \( i \), which is public:

\[
\begin{align*}
  l_1 &\triangleq ((a:S)\cdot(1:P) \leftrightarrow \lambda) \mid ((a:S)\cdot(0:S) \leftrightarrow a) \in (a:S\cdot(0:S \mid 1:P)) \leftrightarrow (\lambda \mid a) \\
  l_2 &\triangleq ((b:S)\cdot(1:P) \leftrightarrow b) \mid ((b:S)\cdot(0:S) \leftrightarrow b) \in (b:S\cdot(0:S \mid 1:P)) \leftrightarrow (b \mid b)
\end{align*}
\]

The concatenation of \( l_1 \) and \( l_2 \) does not have the type obtained by concatenating their source and view types,

\[
l_1 \cdot l_2 \in ((a:S\cdot(0:S \mid 1:P))\cdot(b:S\cdot(0:S \mid 1:P))) \leftrightarrow ((\lambda \mid a)\cdot(\lambda \mid b))\]

124
because the get function exposes the way that the source string was split, as demonstrated by a
counterexample to GetNoLeak:

\[ a_1b_0 \sim_p a_0b_1 \]

but \((l_1 \cdot l_2).\text{get } a_1b_0 = A b \not\sim_p a_B = (l_1 \cdot l_2).\text{get } a_0b_1.\]

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 \((S_1 \cdot S_2)\) iff 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 effectively compute 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.

\[
\begin{align*}
l \in S \iff V & \quad S^* \iff V^* \\
q = \lor \{ k \mid k \text{ min obs. } V^* \} & \\
p = \lor \{ k \mid k \text{ min obs. } S^* \} & \\
l^* \in (S^*): q \iff (V^*): p
\end{align*}
\]

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

**6.4.5 Lemma:** Let \( l \in S \iff V \) be a secure lens such that \( S^* \) and \( V^* \). Then \( l^* \) is a secure lens
in \((S^*): q \iff (V^*): p\) where \( q = \lor \{ k \mid k \text{ min obs. } V^* \} \) and \( p = \lor \{ k \mid k \text{ min obs. } S^* \} \).

125
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:

\[ l \triangleq A : S \leftrightarrow B : P \quad \in \quad A : S \leftrightarrow B : P \]

It is not the case that

\[ l^* \in (A : S)^* \leftrightarrow (B : P)^*, \]

as demonstrated by the following counterexample to GetNoLeak:

\[ AAA \sim_P AA \]

but

\[ l^*.get AAA = BBB \not\sim_P BB = l^*.get BB. \]

The problem is that \textit{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

\[ l \triangleq [0-9] : E \leftrightarrow A : T \in [0-9] : E \leftrightarrow A : T \]

has type

\[ l^* \in ([0-9] : E)^* \leftrightarrow (A : T)^*, \]

as demonstrated by the following counterexample to GetPut:

\[ A \approx_E AAA = l^*.get 123 \]

but

\[ l^*.put A 123 = 1 \not\approx_E 123. \]

Here the problem is that the update shortens the length of the view, which causes the iteration operator to discard endorsed 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 \( V \). These labels can be computed from annotated regular expressions using elementary constructions.

\textbf{Sequential Composition} \quad The sequential composition lens has a straightforward type.

\[
\begin{array}{c}
l_1 \in S \leftrightarrow T \quad l_2 \in T \leftrightarrow V \\
\hline
l_1; l_2 \in S \leftrightarrow V
\end{array}
\]

\[
\begin{array}{c}
g s = l_2.get \ (l_1.get s) \\
p v s = l_1.put \ (l_2.put v \ (l_1.get s)) \ s \\
c v = l_1.create \ (l_2.create v)
\end{array}
\]
6.4.6 Lemma: Let \( l_1 \in S \leftrightarrow T \) and \( l_2 \in T \leftrightarrow V \) be secure lenses. Then \((l_1;l_2)\) is a secure lens in \( S \leftrightarrow V \).

It requires that the view type of the first lens and the source type of the second be identical. As is usual for composition, this is essential for ensuring the secure lens laws.

Filter The secure version of the \textit{filter} lens allows us to hide information in a list of source items.

\[
\begin{align*}
E, F \text{ well-formed} & \quad (\mathcal{L}(E) \cap \mathcal{L}(F)) = \emptyset \quad (\mathcal{L}(E) \mathbin{|} \mathcal{L}(F))^\dagger \\
q & = \bigvee \{k \mid k \text{ min obs. } E^\dagger \} \\
p & \sqsupseteq \bigvee \{k \mid k \text{ observes } E^\dagger F \text{ and } F^\dagger E\} \\
\text{filter } E F & \in (E:q \mathbin{|} F:p)^* \leftarrow \rightarrow E^*
\end{align*}
\]

get \((s_1 \cdots s_n)\) = string\_filter \(E \ (s_1 \cdots s_n)\)
put \((v_1 \cdots v_n) (s_1 \cdots s_m)\) = string\_unfilter \(F \ (v_1 \cdots v_n) \ (s_1 \cdots s_m)\)
create \((v_1 \cdots v_n)\) = \((v_1 \cdots v_n)\)

6.4.7 Lemma: Let \( E \) and \( F \) be well-formed security-annotated regular expressions such that \((\mathcal{L}(E) \cap \mathcal{L}(F)) = \emptyset \) and \((\mathcal{L}(E) \mathbin{|} \mathcal{L}(F))^\dagger \). Then \((\text{filter } E F)\) is a secure lens in \((E:q \mathbin{|} F:p)^* \leftarrow \rightarrow E^*\) where \(q = \bigvee \{k \mid k \text{ min obs. } E^\dagger \}\) and \(p \sqsupseteq \bigvee \{k \mid k \text{ observes } E^\dagger F \text{ and } F^\dagger E\}\).

The typing rule for \textit{filter} captures the fact that none of the \(F\)s are leaked to the view, and so the \(F\)s in the source can be assigned any confidentiality label (that observes the unambiguous concatenability of \(E\)s and \(F\)s). Since observers with clearance lower than \(p\) cannot distinguish source strings that differ only in the \(F\)s, it is simple to show \textsc{GetNoLeak}: two source strings are related by \(\sim_p\) exactly when their filterings—i.e., the views computed by \textit{get}—are related by \(\sim_p\). In the reverse direction, we need to escalate the integrity label on the \(E\)s by the join of the minimal labels observing that \(E\) is unambiguously iterable. However, the \(F\)s are restored exactly, so their integrity level does not need to be escalated.

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.

\[
\begin{align*}
l \in S \leftrightarrow V \quad q \in Q \quad p \in P \\
l \in (S:q) \leftrightarrow (V:p)
\end{align*}
\]
6.4.8 Lemma: Let \( l \in S \leftrightarrow V \) be a secure lens and let \( q \in Q \) and \( p \in P \) be labels. Then \( l \) is also a secure lens in \( (S;q) \leftrightarrow (V;p) \).

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.

6.5 Dynamic Secure Lenses

Using the static type system to track tainted source data is effective, but conservative—it forces us to label source data as tainted if the \( \text{put} \) function ever produces a tainted result, even if there are many inputs for which it does not. In this section, we explore the idea of augmenting lenses with dynamic tests to check whether \( \text{put} \) can preserve the integrity of the endorsed data in the source for a particular view and source. This generalization makes it possible for lenses to make very fine-grained decisions about which views to accept and which to reject, and allows us to assign relaxed types to many of our lens primitives while still retaining strong guarantees about integrity.

At the same time that we extend lenses with these dynamic tests, we also address a subtle interaction between confidentiality and integrity that we have ignored thus far. 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 endorsed 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 \( C \subseteq P \times Q \) be a set of clearances. A user with clearance \((j,k)\) is allowed to access data at confidentiality level \( j \) and modify data tainted at integrity level \( k \). We extend
lenses with a new function

\[ l.\text{safe} \in C \to \mathcal{L}(V) \to \mathcal{L}(S) \to \mathbb{B} \]

that returns \( \top \) iff a user with clearance \((j, k)\) can safely \textit{put} a particular view and source back together. We replace the hypothesis that \( v \approx_k s \) in the \texttt{GetPUT} law with \textit{safe}, requiring, for all \((j, k) \in C \) and \( s \in S \) and \( v \in V \) that

\[
\frac{l.\text{safe} \ (j, k) \ v \ s}{l.\text{put} \ v \ s \approx_k s}
\]

and we revise the protocol for propagating updates to the view accordingly: before allowing the user of a view to invoke \textit{put}, the owner of the source checks that the original and updated views are safe for the user's clearance.

As discussed above, these \textit{safe} functions, which are arbitrary functions, can reveal information about source data. We therefore add a new law stipulating that \textit{safe} must not reveal confidential data, formulated as a non-interference property for every \((j, k) \in C\), every \( s, s' \in S\), and \( v, v' \in V\):

\[
\frac{v \sim_j v' \quad s \sim_j s'}{l.\text{safe} \ (j, k) \ v \ s = l.\text{safe} \ (j, k) \ v' \ s'}
\]

\text{(SafeNoLeak)}

For technical reasons—to prove that the \textit{safe} component of the sequential composition operator, which is defined in terms of the \textit{put} function of one of its sublenses, satisfies \textit{SafeNoLeak}—we also need a law stipulating that the \textit{put} function must be non-interfering for all \((j, k) \in C\) and for all \( s, s' \in S\) and \( v, v' \in V\):

\[
\frac{v \sim_j v' \quad s \sim_j s'}{l.\text{safe} \ (j, k) \ v \ s = l.\text{safe} \ (j, k) \ v' \ s'}
\]

\text{(PutNoLeak)}

With these refinements, we can now present revised versions of each of our secure string lens combinators.

\textbf{Copy} For \textit{copy} the \textit{safe} function checks that the new view and original source agree on \( k\)-trusted data.
6.5.1 Lemma: Let $E$ be a well-formed security-annotated regular expression. Then $(\text{copy } E)$ is a secure lens in $E \leftrightarrow F$.

To ensure that $\text{safe}$ does not leak information, we add a hypothesis to the typing rule requiring that $\sim_j$ must refine $\approx_k$ for every $(j, k) \in C$. This condition captures the essential interaction between the confidentiality and integrity lattices.

**Const** For $\text{const}$, the view type is a singleton, so there is only one possible update—a no-op. Hence, we choose a $\text{safe}$ function that always returns $\text{true}$.

6.5.2 Lemma: Let $E$ and $F$ be well-formed security-annotated regular expressions such that $|\mathcal{L}(F)| = 1$. Then $(\text{const } E \ F)$ is a secure lens in $E \leftrightarrow F$.

**Concatenation** For the concatenation lens, we choose a $\text{safe}$ 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 $\text{PutNoLeak}$ and $\text{SafeNoLeak}$. 

130
\[ l_1 \in S_1 \leftrightarrow V_1 \quad S_1 \cdot S_2 \]
\[ l_2 \in S_2 \leftrightarrow V_2 \quad V_1 \cdot S_2 \]
\[ p = \bigvee \{ k \mid k \text{ min obs. } S_1 \cdot S_2 \} \]
\[ l_1 \cdot l_2 \in (S_1 \cdot S_2) \leftrightarrow (V_1 \cdot V_2):p \]

**Lemma**: Let \( l_1 \in S_1 \leftrightarrow V_1 \) and \( l_2 \in S_2 \leftrightarrow V_2 \) such that \( S_1 \cdot S_2 \) and \( V_1 \cdot V_2 \). Then \((l_1 \cdot l_2)\) is a secure lens in \((S_1 \cdot S_2) \leftrightarrow (V_1 \cdot V_2):p\) where \( p = \bigvee \{ k \mid k \text{ min obs. } S_1 \cdot S_2 \} \).

**Union** For the union lens, the \textit{safe} function tests whether the source and view can be processed by the same sublens. (Additionally, because \textit{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.)

\[ (S_1 \cap S_2) = \emptyset \]
\[ l_1 \in S_1 \leftrightarrow V_1 \quad l_2 \in S_2 \leftrightarrow V_2 \]
\[ p = \bigvee \{ k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset \} \]
\[ l_1 \cdot l_2 \in (S_1 \cap S_2) \leftrightarrow (V_1 \cap V_2):p \]

\[ \text{safe} (j, k) v_1 v_2 s_1 s_2 = \]
\[ j \text{ observes } (S_1 \cap S_2) = \emptyset \text{ and } V_1 \& V_2 \text{ agree} \]
\[ \land \begin{cases} l_1 \cdot \text{safe} (j, k) v_1 s & \text{if } v \in V_1 \land s \in S_1 \\ l_2 \cdot \text{safe} (j, k) v_2 s & \text{if } v \in V_2 \land s \in S_2 \\ \bot & \text{otherwise} \end{cases} \]

**Kleene Star** For the Kleene star lens, \textit{safe} checks that the view is the same length as the one generated from the source. Because \textit{safe} can be used to determine the length of the source, we

\[ p = \bigvee \{ k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset \} \].
require that \( j \) observe the unambiguous concatenability of \( S \) and \( V \) (which implies that \( j \) can distinguish strings of different lengths).

\[
\begin{align*}
  l \in S & \leftrightarrow V \\
  p = \bigvee \{ k \mid k \min \text{obs. } S^* \} \\
  l^* \in S^* & \leftrightarrow (V^*):p
\end{align*}
\]

\[\text{safe} (j, k) (v_1 \cdots v_n) (s_1 \cdots s_m) = \]
\[j \text{ observes } S^* \text{ and } V^* \]
\[\land n = m \land l.\text{safe} (j, k) v_i s_i \text{ for } i \in \{1, \ldots, n\}\]

\[6.5.5 \text{ Lemma: Let } l \in S \leftrightarrow V \text{ be a secure lens such that } S^* \text{ and } V^*. \text{ Then } l^* \text{ is a secure lens in } (S^*) \leftrightarrow (V^*):p \text{ where } p = \bigvee \{ k \mid k \min \text{obs. } S^* \}.\]

**Sequential Composition** For sequential composition, the \( \text{safe} \) function requires the conditions implied by \( l_1 \)'s \( \text{safe} \) function on the intermediate view computed by \( l_2 \)'s \( \text{put} \) on the view and the original source.

\[
\begin{align*}
  l_1 \in S & \leftrightarrow T \\
  l_2 \in T & \leftrightarrow V \\
  l_1; l_2 \in S & \leftrightarrow V
\end{align*}
\]

\[\text{safe} (j, k) s v = l_1.\text{safe} (j, k) (l_2.\text{put } v (l_1.\text{get } s)) s\]

\[6.5.6 \text{ Lemma: Let } l_1 \in S \leftrightarrow T \text{ and } l_2 \in T \leftrightarrow V \text{ be secure lenses. Then } (l_1;l_2) \text{ is a secure lens in } S \leftrightarrow V.\]

Note that the composition operator is the reason we need the \( \text{PutNoLeak} \) law and that \( \text{SafeNoLeak} \) needs to require that \( \text{safe} \) be non-interfering in both its source and view arguments (rather than just its source argument). We could relax these conditions by only requiring \( \text{PutNoLeak} \) of lenses used as the second argument to a composition operator and the full version of \( \text{SafeNoLeak} \) of lenses used as the first argument. This would give us yet more flexibility in designing \( \text{safe} \) 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 \( k \)-endorsed data. Additionally, to ensure that *safe* does not leak information about the source, *safe* also checks that \( j \) observes the way the way that \( E \)s and \( F \)s are split in the source, as well as the unambiguous iterability of \( E \).

\[
E, F \text{ well-formed } \quad \mathcal{L}(E) \cap \mathcal{L}(F) = \emptyset \quad (\mathcal{L}(E) \mid \mathcal{L}(F))^!\*
\]
\[
p \supseteq \lor \{ k \mid k \text{ observes } E \cdot F \text{ and } F \cdot E \}
\]
\[
\forall (j, k) \in C. \; \sim^E_j \subseteq \approx^E_k
\]

*safe* \((j, k)(v_1 \cdots v_n)(s_1 \cdots s_m) =
\]
\[
j \text{ observes } E \cdot F \text{ and } F \cdot E \quad \land \quad j \text{ and } k \text{ observe } E^{!*}
\]
\[
\land (v_1 \cdots v_n) \approx^E_k (\text{string_filter } E (s_1 \cdots s_m))
\]

6.5.7 Lemma: Let \( E \) and \( F \) be well-formed security-annotated regular expressions such that \((\mathcal{L}(E) \cap \mathcal{L}(F)) = \emptyset \) and \((\mathcal{L}(E) \mid \mathcal{L}(F))^!\*\). Then \((\text{filter } E \; F)\) is a secure lens in \((E \mid F; p)^* \leftrightarrow E^*\) where \( p \supseteq \lor \{ k \mid k \text{ observes } E \cdot F \text{ and } F \cdot E \}\).

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. It would be interesting investigate the tradeoffs along these axes in the future.

6.6 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 leak source information hidden by the view, and reliable, since they come equipped with strong integrity guarantees.
Chapter 7

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—i.e., writing substantial lens programs only using low-level combinators would be extremely tedious. We don’t do this. Instead, we have embedded the lens combinators into a full-blown programming language called Boomerang. This chapter highlights the main features of Boomerang, focusing on high-level syntax, typechecking, and engineering issues.

7.1 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 components of the program, and so on. Boomerang is organized as a functional language, so it comes equipped with rich mechanisms for programming with lenses.

More concretely, Boomerang is based on the polymorphic $\lambda$-calculus, also called System F (Girard, 1972; Reynolds, 1974), extended with subtyping, dependent types, and refinement types. Figure 7.1 presents the core syntax of the language. Since the language is based on a
\[ m ::= \text{module} \ x = d^* \]

\[ d ::= \]
\[ \quad \text{type} \ a^* \ x = (| x \text{ of } t|)+ \]
\[ \quad \text{let} \ p : t = e \]
\[ \quad \text{test} \ e = e \]
\[ \quad \text{test} \ e : t \]

\[ t ::= \]
\[ \quad \text{forall} \ a -> t \]
\[ \quad \text{'a} \]
\[ \quad x:t -> t \]
\[ \quad (x:t \text{ where } t) \]
\[ \quad t * t \]
\[ \quad t^* X \]
\[ \quad \text{unit} | \text{int} | \text{bool} | \text{char} | \text{string} | \text{regexp} | \text{canonizer} | \text{lens} \]

\[ e ::= \]
\[ \quad x \]
\[ \quad k \]
\[ \quad \text{fun} \ (x:t) : t -> e \]
\[ \quad e e \]
\[ \quad \text{fun} \ a -> e \]
\[ \quad e\{e\} \]
\[ \quad \text{let} \ p : t = e \text{ in } e \]
\[ \quad \text{match} \ e \text{ with } (| p -> e|)+ : t \]
\[ \quad e , e \]
\[ \quad < (e:)? e > \]
\[ \quad e \ o e ? \]

\[ p ::= \_ | x | k | p,p | X p \]

\[ k ::= \text{'}c\text{'} | \text{"s"} | n | b | () | [-?c-c] \]

\[ o ::= \langle-> | 1 | . | - | * | + | ? | \{n,n\}? | ; | \& | - | \&\& | || | < | > | <= | >= \]

---

Figure 7.1: Core Boomerang Syntax
standard formalism, we will highlight its main features by example, rather than giving a full formal definition (see Pierce 2002, Chapter 23 for a textbook description of System F).

A Boomerang program is a functional program over base types \texttt{string}, \texttt{regexp}, \texttt{canonizer}, \texttt{lens}, etc. The standard library includes primitives for each lens combinator. To use a lens defined in a Boomerang program to manipulate string data, we first evaluate the program to obtain a lens value, and then apply the lens to the string itself. As an example, recall the \texttt{xml_elt} helper used to define the composers lens. It takes a string \texttt{tag} and a lens \texttt{body} as arguments and builds a lens that transforms XML elements named \texttt{tag} using \texttt{body} to process the children of the element:

\begin{verbatim}
let xml_elt (tag:string) (body:lens) : lens =
  del WHITESPACE*
  . del ("<" . tag . ">")
  . body
  . del WHITESPACE*
  . del ("</" . tag . ">")
\end{verbatim}

Having defined \texttt{xml_elt}, we can easily instantiate it with appropriate arguments obtain lenses for processing particular elements. For example,

\begin{verbatim}
let name : lens = xml_elt "name" (copy (ALPHA . SPACE . ALPHA) )
let lived : lens = xml_elt "lived" (copy (YEAR . DASH . YEAR) )
\end{verbatim}

handle strings of the form \texttt{<name>Jean Sibelius</name> and <lived>1865-1957</lived>} respectively. Compare the versions of these lenses written in terms of \texttt{xml_elt} to the programs we would need to write using combinators:

\begin{verbatim}
let name : lens =
  del WHITESPACE*
  . del "<name>
  . copy (ALPHA . SPACE . ALPHA)
  . del WHITESPACE*
  . del "</name>
\end{verbatim}

and

\begin{verbatim}
let lived : lens =
  del WHITESPACE*
  . del "<lived>
  . copy (YEAR . DASH . YEAR)
  . del WHITESPACE*
  . del "</lived>
\end{verbatim}

It should be clear that the first versions are simpler to write, reason about, and maintain. Moreover, we can assign them intuitive names, allowing the programmer can work at an appropriate level of abstraction. For example, to build a lens that handles the name and dates for a composer, we can simply write,
which reflects the essential nature of the transformation, rather than

```plaintext
del WHITESPACE*
  . del "<name>"
  . copy (ALPHA . SPACE . ALPHA)
  . del WHITESPACE*
  . del "</name>"
  . ins (COMMA . SPACE)
  . del WHITESPACE*
  . del "<lived>"
  . copy (YEAR . DASH . YEAR)
  . del WHITESPACE*
  . del "</lived>"
```

which exposes all of the low-level details of the lens and obscures the actual transformation being done on data.

Boomerang has many other features designed to simplify lens development. A simple module system provides a way to separate definitions into distinct name spaces:

```plaintext
module M =
  let l : lens = copy [A-Z] . del [0-9]
  let x : string = "A"
  let y : string = "1"
end
```

Built-in tests gives programmers an easy way to test the behavior of their program during development and also provides helpful documentation:

```plaintext
test (M.l).get (M.x . M.y) = M.x
```

Unit tests can also be used to check errors:

```plaintext
test (M.l).get M.x = error (* type error: M.x not in (stype M.l) *)
```

User-defined data types, polymorphism, and pattern matching, make it possible to write lenses parameterized on data structures such as lists. As an example, let us construct a generic lens for escaping characters as strings. To represent the escape codes for a particular format, we will use lists. The Boomerang List module contains the standard polymorphic definition lists:

```plaintext
type 'a t = Nil | Cons of 'a * 'a t
```

We can represent the escape codes for XML PCDATA as the following list:

```plaintext
let xml_escs : (char * string) List.t =
  #
```
The characters represent symbols that need to be escaped and the strings represent the escaped versions. 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 expression that describes the set of unescaped characters using a fold:

```ocaml
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,

```ocaml
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
```

is defined by folding down the list of escape codes and, at each step, taking the union of the accumulated lens and the rewriting between the unescaped character and the concatenation of the escape character and the escaped string—e.g., (\'<\' <-> '\&' \'<\' \"lt;\")\). The initial lens supplied to the fold copies characters that do not need to be escaped. By iterating escape_char using Kleene star, we obtain a lens that escapes arbitrary strings, not just single characters:

```ocaml
let escape (escs : (char * string) List.t) : lens = (escape_char escs)*
```

The `xml_esc` lens is a straightforward instantiation of `escape` with the escape character `&` and `xml_esc`:

```ocaml
let xml_esc : lens = escape xml_escs
```

test xml_esc.get "Duke Ellington & His Orchestra" = "Duke Ellington &amp; His Orchestra"

For comparison, here is the same lens written directly using combinators:

```ocaml
let xml_esc : lens =
  ( '>\' <-> "&gt;\"
  \'<\' <-> "&lt;\"
  '\&' <-> "&amp;\"
  copy [\'<\&] )*\n```

This is not so bad, but it only handles XML PCDATA. If we need to escape strings in CSV files, we need to write another lens from scratch whose structure is essentially the same:
let csv_esc : lens =
  ( ',' <-> "\\,"
   '\n' <-> "\\n"
   '\\' <-> "\\\\"
   copy [^,\n\\] )*

By defining escape generically, we avoid having to repeat the definition.

Most functional languages are used to describe general-purpose computations. However, Boomerang is a language specifically designed for transforming strings. As a result, we have equipped it with a number of features aimed at making it easier to describe strings, regular expressions, and functions on strings. For example, rather than handling regular expressions by an external library as in many other languages, standard POSIX notation is built into the syntax of the language. This lets programmers manipulate regular expressions directly, rather than having to wrap them up into string literals and pass them off to a library function (this quickly gets complicated when the strings need to be escaped). For example:

    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 a string belongs to the language denoted by a regular expression.)

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 type checker automatically resolves overloaded symbols and selects the appropriate operator. This allows programmers to describe transformations using whatever syntax is most convenient. For example:

    test 'a' . 'b' : string
    test "a" . "b" : string
    test [a] . [b] : regexp
    test (copy "a" . copy "b") : lens

Boomerang also recognizes the following subtype relationships between base types:

char <: string <: regexp <: lens

This simplifies many programs—e.g., strings can be directly treated like regular expressions and lenses. When the subsumption rule is used in the typechecker, it inserts explicit coercions to promote run-time values (Breazu-Tannen et al., 1991). For example, to concatenate a character with a string (here, the . resolves to the string concatenation operation),

    test 'a' . "b" = "ab"
the typechecker first inserts a coercion that converts ‘a’ to a string, and then performs the actual concatenation on strings—i.e., the expression essentially desugars into the following more explicit version:

```haskell
test string_concat (string_of_char 'a') "b" = "ab"
```

More interesting—and more useful—is the coercion from regular expressions to lenses: it uses `copy` to produce a lens that copies strings belonging to the regular expression from source to view. This turns out to be quite convenient—omitting `copy` makes the description of many lens programs much shorter. As examples to illustrate subtyping in Boomerang, consider the following:

```haskell
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 ensures that the treatment of subtyping is coherent (Breazu-Tannen et al., 1991).

### 7.2 Typechecking

Boomerang comes equipped with a very expressive type system: in addition to the standard types found in the polymorphic \( \lambda \)-calculus—sums, products, functions, and polymorphism—it includes dependent function types and refinement types. A dependent function type “\( x : t_1 \rightarrow t_2 \)” generalizes the ordinary function type “\( t_1 \rightarrow t_2 \)” by allowing the \( t_2 \) to depend on the value of the argument supplied for \( x \). This feature most useful in the presence of refinement types. A refinement type “\( (x : t \text{ where } p) \)” further constraints values of type \( t \) by requiring that the predicate \( p \) be satisfied.

Together, these precise types are capable of expressing extremely detailed properties of programs. In Boomerang, their primary use is for encoding the typing rules for our lens combinators. For example, here is the type of the concatenation lens as declared in the Boomerang standard library:

```haskell
test lens_concat :
(11:lens ->
 (12:lens where splittable 11.stype 12.stype
   && splittable 11.vtype 12.vtype) ->
 (lens in (11.stype . 12.stype) <-> (11.vtype . 12.vtype)))
```
It requires that the source and and view types each be unambiguously concatenable and guarantees that the source and view types of the lens it constructs are the concatenations of the respective types from \( l_1 \) and \( l_2 \). Refinement types make it possible to state the requirement on the types of the two lenses while dependent function types make it possible for the refinement on \( l_2 \) to refer to \( l_1 \). The \textit{splitable} function used in the refinement on \( l_2 \) is a binary predicate that tests if two regular expressions are unambiguously concatenable. The notation used in the return type, “\( \text{lens in } S \leftrightarrow V \)”, desugars to an ordinary refinement type “\( (l : \text{lens where } l.\text{stype} = S \&\& l.\text{vtype} = V) \)” where \( l \) is fresh. Another example is the union lens:

\[
\text{test lens_union} :
(11 : \text{lens} \rightarrow
(12 : \text{lens where disjoint } 11.\text{stype} 12.\text{stype}) \rightarrow
(\text{lens in } (11.\text{stype} \mid 12.\text{stype}) \leftrightarrow (11.\text{vtype} \mid 12.\text{vtype})))
\]

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:

\[
\text{test default} :
(1 : \text{lens} \rightarrow
((\text{string in } l.\text{vtype}) \rightarrow (\text{string in } l.\text{stype})) \rightarrow
(\text{lens in } l.\text{stype} \leftrightarrow l.\text{vtype}))
\]

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 “\( \text{string in } R \)” desugars to “\( (x : \text{string where matches } R x) \)” where \( x \) is fresh. The \textit{get}, \textit{put} and \textit{create} functions, which extract the component functions of a lens have the following types,

\[
\text{test get} : (1 : \text{lens} \rightarrow (\text{string in } l.\text{stype}) \rightarrow (\text{string in } l.\text{vtype}))
\]
\[
\text{test put} : (1 : \text{lens} \rightarrow (\text{string in } l.\text{vtype}) \rightarrow (\text{string in } l.\text{vtype}) \rightarrow (\text{string in } l.\text{stype}))
\]
\[
\text{test create} : (1 : \text{lens} \rightarrow (\text{string in } l.\text{vtype}) \rightarrow (\text{string in } l.\text{stype}))
\]

which check that the strings supplied as the source and view have the correct type.

An early version of Boomerang did not support dependent and refinement types. Instead, the language only had the types of standard polymorphic \( \lambda \)-calculus—the native function implementing each lens primitive had to check the conditions mentioned in the typing rule for the combinator. Because of the two-tier evaluation strategy used with lenses in Boomerang—evaluate the functional program to obtain a lens value, and then apply this lens to strings—this approach was safe. That is, the conditions mentioned in the typing rule of a lens were always verified before we used the lens. However, as we began to develop larger libraries of lens code, we
discovered a practical flaw in this approach—because it checked typing rules in native code, errors were reported quite late, and it was very inconvenient for programmers to have to trace back through their program to discover the source of the error.

As an example illustrating this problem, recall the `escape_char` function defined above:

```ocaml
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 - (unescape 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 does the escaping. Because it is defined using union, there is a subtle constraint on the type of `escs`—the list of escape codes must not contain repeated characters. For example, consider the following instantiation of `escape_char`:

```ocaml
escape_char #{char * string}[('&',"&amp;"); ('<','&lt;'); ('<','&lt;')]
```

where `<` appears twice in the list of unescaped characters. Evaluating this expression will trigger an error because the source types of the two lenses combined using union lens in the last iteration of the fold,

'<' <-> "&lt;"

and

( '<' <-> "&" | ' '<-> "&amp;" | copy (ANYCHAR - ["&<"] )

are not disjoint. However, the union lens is not a good place to report the error to the programmer—it forces them to trace backward through the entire execution of their program to determine the cause of the error. This is so bad for this program, which is a simple fold, but rapidly becomes impractical as programs grow in size. It also breaks modularity—in general, programmers will need to examine code referenced from other modules when things go wrong.

Our original motivation for adding refinement and dependent types to Boomerang was to provide programmers with a way to express the precise conditions needed to guarantee that evaluation of the program will not go wrong. These types make it possible to detect errors early, and not just in the constructors for lenses. For example, they can assign `escape_char` a type type that accurately reflects the properties needed of the list of escape codes:

```ocaml
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 is the obvious predicate on lists of characters:

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

With these definitions, the error in the above example will be detected early—i.e., as soon as escape_char is applied to a list of non-disjoint escape codes.

Boomerang’s typechecker is implemented in the hybrid style, using contracts (Flanagan, 2006; Wadler and Findler, 2007; Findler and Felleisen). That is, the static typechecker uses a coarse analysis to rule out obviously ill-formed programs and dynamic assertions to verify the constraints expressed by precise types. Greenberg, Pierce, and Weirich (2009) are currently developing the foundations of this hybrid approach.

7.3 Implementation

We have developed a prototype implementation of the Boomerang language. This system includes an interpreter for the surface language, native implementations of the core basic, resourceful, and quotient lens combinators, and several generic lens libraries for handling escaping, lists, sorting, and XML. (The type system of 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 concatenations and iterations are unambiguous. We have implemented an efficient regular expression library in OCaml based on Brzozowski derivatives Brzozowski (1964). The library makes heavy use of hash consing and memoization to avoid recomputing results, and a clever algorithm for deciding ambiguity due to Møller (2001).

Almost all of the examples typeset in a typewriter font in this dissertation have been generated from a literate source file and checked against the Boomerang implementation.
7.4 Augeas

Lenses has recently been adopted in industry. Red Hat Linux, Inc., has released a tool for managing operating system configurations called Augeas that is directly based on Boomerang (Lutterkort, 2008). The main use of lenses in Augeas is to map textual configuration files of the kind typically found under the `/etc` directory in most Unix systems to simplified tree structures that are easily to manipulate using scripts. The language that Augeas programmers used to write lenses is directly based on an early version of Boomerang—it uses the same set of core combinators, and the same surface syntax (based on OCaml), and the an early version of the mechanisms described in Chapter 5 for handling ordered data. It also extends the language with some new combinators for indicating the tree structure of the view.

Here is a lens developed by Pinson (2009) that builds an updatable view over preference files for 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
```

and the view are trees (or in this case, sequences of trees) representing the same 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 simple ASCII notation for trees nodes are indicated using curly braces, and ever node has a label (which can be the empty string), an optional value, and a sequence of children. For example, the subtree
"Pin"   = "release"
"a" = "backports"

has the label “Pin”, value “release”, and a single child.

Here is the definition of the lens that computes this view in Augeas’s lens language:

```ocaml
module AptPreferences =
  autoload xfm
(* helpers *)
let colon = del /[ \t]*/ ": "
let eol = del /[\t]*\n/ 
let value_to_eol = store /[([^\t\n].*[^\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 Augeas program uses many of the same primitives as Boomerang—e.g., the \(\text{del}, (.), (1),\) and (\(\ast\)) lenses—as well as some new primitives for building the tree structure of the view. The \((\text{store } E)\) primitive matches a string described by \(E\) and stores it as the value of the enclosing subtree. The \((\text{key } E)\) primitive matches a string described by \(E\) and stores it as the label of the enclosing subtree. The \((\text{seq } x)\) primitive provides a way to generate sequential labels—it labels the enclosing subtree with the next value from a counter identified by \(x\). The \([l]\) primitive builds a tree node. It uses the \(\text{key}\) or \(\text{seq}\) primitive in \(l\) (which must be unique) to generate the label, the \(\text{store}\) primitive (which also must be unique) to generate the value, and \(l\) to generate the children. For example, the \(\text{record}\) lens, declared above as

```ocaml
let record = [ seq "record" . entries+ ]
```

generates a single tree labeled by the current value of the \(\text{record}\) counter and containing a non-empty list of children, each generated by \(\text{entries}\).
Augeas contributors have developed a large number of lenses—one for each of the following configuration file formats:

- aliases.aug
- exports.aug
- logrotate.aug
- puppet.aug
- sudoers.aug
- aptpreferences.aug
- fstab.aug
- monit.aug
- rsyncd.aug
- syslog.aug
- aptsources.aug
- gdm.aug
- ntp.aug
- samba.aug
- utilaug
- bbhosts.aug
- group.aug
- openvpn.aug
- services.aug
- vsftpd.aug
- crontab.aug
- grub.aug
- pam.aug
- shellvars.aug
- webmin.aug
- darkice.aug
- hosts.aug
- passwd.aug
- slapd.aug
- xinetd.aug
- dhclient.aug
- inittab.aug
- phpvars.aug
- spacevars.aug
- yum.aug
- dpkg.aug
- interfaces.aug
- postfix_main.aug
- squid.aug
- dput.aug
- limits.aug
- postfix_master.aug
- sshd.aug

Augeas is also starting to be used more widely—e.g. in the Puppet configuration management tool (Pup, 2009) and in the Netcf network interface configuration tool (Lutterkort, 2009).

### 7.5 Grammars

This final describes an extension to Boomerang we have recently implemented. Boomerang’s functional infrastructure goes a long way toward making high-level lens programming convenient. But, ultimately, it still forces programmers to describe lenses using combinators. For some transformations—in particular lenses that rearrange data in going from source to view—this is tedious. This section describes an extension to Boomerang for describing lenses using grammars. Rather than having to use combinators to massage source data into the correct position in the view, programmers can bind pieces of the source to variables and place data directly in the view. Our design for grammars is directly inspired by XSugar (Brabrand et al., 2008). To illustrate, here is the composer lens written as a grammar:

```plaintext
let composer : lens =
  grammar composer ::= 
    WS
    "<composer>" WS
    "<name>" n:(key (ALPHA . SPACE . ALPHA)) "</name>" WS
    "<lived>" d:(YEAR . DASH . YEAR) "</lived>" WS
    "<nationality>" ALPHA "</nationality>" WS
    "</composer>" <->
    n COMMA SPACE d
  end
```

The grammar contains a single production named `composer` with 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 produces the view by concatenating \( n \) and \( d \) with a comma and a 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:

```ml
let composer_list : lens =
  grammar composer_list ::=
    c:< composer > <-> c
    | c:< composer > cs:composer_list <-> c NEWLINE 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 a lens that does the complete transformation:

```ml
let composers : lens =
  grammar composers ::= 
    "<composers>" WS "</composers>" <-> EPSILON
    | "<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 describe the lens using combinators or grammars is a matter of taste. However, the advantages of the grammar approach become more evident when we want 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 3, we can do this using the swap lens, but the combinator program becomes much more complicated—we need to place the swap carefully to lift the dates over the names:

```ml
let composer : lens =
xml_elt "composer"
 ( ( xml_elt "name" (copy (ALPHA . SPACE . ALPHA) )
- ( xml_elt "lived" (copy (YEAR . DASH . YEAR) )
  . ins (COMMA . SPACE) )
- xml_elt "nationality" ( default (del ALPHA) (fun (x:string) -> "Unknown") ) )
```

This approach rapidly becomes complicated in situations where the transformation reorders multiple pieces of information in the source (and even using the \( n \)-ary generalization of swap, `permute`, described in Chapter 3). By contrast, the grammar version of the lens can be easily modified to obtain the behavior we want—we just need to invert the order of the variables \( n \) and \( d \) on the right-hand side of the rule, replacing “\( n \) COMMA SPACE \( d \)” with “\( d \) COMMA SPACE \( n \)”. Thus, grammars and variables provide a much more natural way to describe many transformations on strings—in particular, lenses 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 compilation that maps productions to combinator expressions. Formally, the syntax of grammars is given by the following extension to the Boomerang syntax shown in Figure 7.1:

That is, a grammar \( g \) is a list of productions; a production \( p \) consists of a name \( x \) and a set of rules (separated by \( \mid \)); 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.

148
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

\[ a_1 \ldots a_k \leftrightarrow b_1 \ldots b_l \quad \text{or} \quad a_1 \ldots a_k \ x : e_{k+1} \leftrightarrow b_1 \ldots b_l \ x \]

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 \texttt{composer} production compiles to the following lens:

\begin{verbatim}
permute
  #{int}[1;2;3;4;5]
  #{lens}{ del ( WS . "<composer>" . WS . "<name>" )
     ; key (ALPHA . SPACE . ALPHA)
     ; "</name>" . WS . "<lived>" <-> COMMA . SPACE
     ; copy (YEAR . DASH . YEAR)
     ; del ( "</lived>" . WS . "<nationality>" . ALPHA .
     "</nationality>" . WS . "</composer>" ) }
\end{verbatim}

The list of integers represents a permutation (the “\#{int}” is a type argument that instantiates the polymorphic list type). 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 \texttt{composer_list} production yields the following:

\[ \texttt{composer_list ::= (permute \{int\}[1] \{lens\}(< composer >))} \]

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 \( x_i \):

\[ x_i = (|_{x_i \in S_1}) ^ k \cdot (|_{x_i \in S_2}) ^ l \]
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 well-typed lenses for grammars that are “locally unambiguous” in this sense.

### 7.6 Summary

The Boomerang language provides convenient high-level notation for programming with lenses and an expressive type system for establishing correctness. In our experience, these features make it possible to develop lens programs of substantial size. We have developed a number of lenses for real-world data formats including electronic address books, calendars, bibliographies, and scientific data. Our design has also been validated in industry: the Augeas tool uses a language directly based on Boomerang. Finally, we have also developed an extension to the language makes it possible to describe lenses using grammars instead of combinators.
Chapter 8

Related Work

This chapter discusses previous work on view update and bidirectional languages. The foundational structures described in this dissertation—lenses and their associated behavioral laws—are not completely new: similar structures that have been studied for decades in the database community. The idea of programming languages that can be interpreted in two directions has also been explored previously in the programming language community.

Broadly speaking, our work on lenses has several features that distinguish it from previous work. One is that lenses transform whole states rather than “update functions”. Another is that they treat well-behavedness as a form of type assertion. Lenses are also novel in addressing the issues of totality—i.e., offering static guarantees that propagating updates will not fail at run time. On the linguistic side, lenses appear to be the first language to take totality as a primary goal (while connecting the language with a formal semantic foundation, choosing primitives that can be combined into composite lenses where totality is guaranteed by construction), and the first to emphasize types—i.e., compositional reasoning about well-behavedness and totality—as an organizing design principle.

8.1 Foundations

The foundations of view update translation were studied intensively by database researchers in the late ’70s and ’80s. This thread of work is closely related to our semantics of lenses. We discuss here the main similarities and differences between our work and these classical ap-
approaches to view update—in particular Dayal and Bernstein’s notion (1982) of “correct update translation,” Bancilhon and Spyrotas’s (1981) notion of “update translation under a constant complement,” Gottlob, Paolini, and Zicari’s “dynamic views” (1988), and the basic view update and “relational triggers” mechanisms offered by commercial database systems such as Oracle (Fogel and Lane, 2005; Lorentz, 2005).

The view update problem concerns translating updates on a view into “reasonable” updates on the underlying database. It is helpful to structure the discussion by breaking this broad statement of the problem into more specific questions: First, what is a “reasonable” translation of an update? Second, how should we handle the possibility that, for some updates, there may be no reasonable way of translating its effect to the underlying source? And third, how do we deal with the possibility that for some updates there may be 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 in fact there is a remarkable degree of agreement in the literature, with most approaches adopting one of two basic positions. The stricter of these is enunciated in Bancilhon and Spyrotas’s (1981) notion of complement of a view, which must include at least all information missing from the view. When a complement is fixed, there exists at most one update of the database that reflects a given update on the view while leaving the complement unmodified—i.e., that “translates updates under a constant complement.” The constant complement approach has influenced numerous later works in the area, including recent papers by Lechtenbörger (2003) and Hegener (2004).

The other, more permissive, definition of “reasonable” is elegantly formulated by Gottlob, Paolini, and Zicari, who call it “dynamic views” (1988). They present a general framework and identify two special cases, one being formally equivalent to Bancilhon and Spyrotas’s constant complement translators and the other—which they advocate on pragmatic grounds—being their own dynamic views.

Our notion of lenses adopts the same, more permissive, attitude towards reasonable behavior of update translation. Indeed, modulo some technical refinements: the set of well-behaved lenses is isomorphic to the set of dynamic views in the sense of Gottlob, Paolini, and Zicari. Moreover, the set of very well-behaved lenses is isomorphic to the set of translators under constant complement in the sense of Bancilhon and Spyrotas.
Dayal and Bernstein’s (1982) seminal theory of “correct update translation” also adopts the more permissive position on “reasonableness.” Their notion of “exactly performing an update” corresponds, intuitively, to our \texttt{PutGet} law.

The pragmatic tradeoffs between these two perspectives on reasonable update translations are discussed by Hegner (1990; 2004), who introduces the term \textit{closed view} for the stricter constant complement approach and \textit{open view} for the looser approach adopted by dynamic views and in the present work. Hegner adopts a closed-world approach, but notes that both choices may have pragmatic advantages in different situations. In particular, the open-view approach is useful when the users are aware that they are actually using a view as a convenient way to edit an underlying database, while closed-world is preferable when users need views that are robust abstractions, even this means that the set of possible updates to the view must be restricted.

Hegner (2004) also formalizes an additional condition (which has also been noted by others—e.g., by Dayal and Bernstein) called \textit{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.

Commercial database systems such as Oracle (Fogel and Lane, 2005; Lorentz, 2005), Microsoft’s SQL Server Server (2005), and IBM’s DB2 (IBM, 2004) typically provide two quite different mechanisms for building updatable views. First, very simple views defined using select, project, and a very restricted form of join (i.e., where the key attributes of one relation are a subset of those in the other) are considered \textit{inherently updatable}. For these, the notion of reasonableness is essentially the constant complement position. Alternatively, programmers can support updates to arbitrary views by adding \textit{relational triggers} that are 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 the section was how to deal with the possibility that there are no reasonable translations for some update. The simplest response is just to let the translation of an update fail, if it sees that its effect is going to be unreasonable. The advantage of this approach is that we can determine reasonableness on a case-by-case basis, allowing translations that usually give reasonable results but that might fail under rare condi-
tions. The disadvantage is that we lose the ability to perform offline updates to the view—we need the underlying database in order to test if a particular view update is allowed. Another possibility is to restrict the set of operations to just the ones that can be guaranteed to correspond to reasonable translations; this is the position taken by most papers in the area. A different approach—the one we have taken in this 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, for a particular choice of complement, there will be at most one translation. Hegner’s additional condition of monotonicity (Hegner, 2004) 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 (2001). Buneman, Khanna, and Tan (2002) 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.

One way of understanding the main idea explored in this dissertation is that it provides programmers with a way to describe the update policy at the same time as the view is defined—i.e., by enriching the primitives of the language so that the programmer can pick a reasonable policy for propagating updates to views.

8.2 Programming Languages

At the level of syntax, different forms of bidirectional programming have been explored across a surprisingly diverse range of communities, including programming languages, databases, program transformation, constraint-based user interfaces, and quantum computing. One useful way of classifying these languages is by the “shape” of the semantic space in which their trans-
formations live. We identify three major classes:

**Bidirectional languages** pair a *get* function of type $S \rightarrow V$ with a *put* function of type $V \rightarrow S \rightarrow S$. In general, the *get* function can project away some information from the source, which must then be restored by the *put* function.

**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 only that the work performed by any function to produce a given output can be undone by applying the function “in reverse” working backwards from this output to produce the original input. Here, there is no separate *put* function at all: instead, the *get* function itself is constructed so that each step can be reversed.

In the first class, the work that is fundamentally most similar to ours is Meertens’s formal treatment of *constraint maintainers* for constraint-based user interfaces (1998). Meertens’s semantic setting is actually even more general: he takes *get* and *put* to be relations, not just functions, and his constraint maintainers are 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. He considers constraint maintainers for ordered data such as lists, but here adopts a rather different point of view from our approach in resourceful lenses, focusing on constraint maintainers that work with structures not directly but in terms of the “edit scripts” that might have produced them—i.e., he switches from a state-based to an operation-based treatment at this point.
Bidirectional languages capable of duplicating data in the get direction, either by explicit combinators or implicitly by non-linear uses of variables, have been the focus of recent work by the Programmable Structured Documents group at Tokyo.

Early work by Mu, Hu, and Takeichi on “injective languages” for view-update-based structure editors (2004b) 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. A primary concern is using the view-to-view transformations to simultaneously restore invariants within the source view as well as update the source. For example, an 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 not only must the change be propagated to the source, it must also add a new element to the second list in the view. It is easy to see that PutGet cannot hold if the view is modified by put. Similarly, they assume that edits to the view mark modified fields as “updated.” These marks are removed when the put lens computes the modifications to the concrete view—another change to the view that violates PutGet. Consequently, to support invariant preservation within the abstract view, and to support edit lists, their transformations only obey a much weaker variant of PutGet called PutGetPut (see Chapter 4).

Another line of work by Hu, Mu, and Takeichi (2004; 2006) applies a bidirectional programming language quite closely related to ours to the design of “programmable editors” for structured documents. As in the earlier work by Mu, Hu, and Takeichi (2004b), they support preservation of local invariants in the put direction. Here, 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 the work by Mu, Hu, and Takeichi (2004b), the notion of well behavedness must be 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 (Liu et al., 2007). 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 a condition proposed in the journal version of a paper by Hu, Mu, and Takeichi (2004). 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 q-lens law and their plus update preservation. (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.)

Languages for Bijective Transformations

An active thread of work in the program transformation community concerns program inversion and inverse computation—see, for example, Abramov and Glück (2000; 2002) and many other papers cited there. Program inversion (Dijkstra, 1979) derives the inverse program from the forward program. Inverse computation (McCarthy, 1956) 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 (1997) 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 (1998). Their get functions involve parsing, while put functions involve unparsing. Again, to avoid
ambiguous abstract updates, they restricted themselves to \textit{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 (Brabrand et al., 2008) 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 7 was inspired by XSugar. A similar language biXid (Kawanaka and Hosoya, 2006) also specifies bidirectional conversions between pairs of XML documents. However, unlike XSugar, biXid allows ambiguous productions and non-linear variables.

The PADS system (Fisher and Gruber, 2005) generates a data type, parser, and pretty printer for an ad-hoc data from a single, declarative description of the data 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, separated lists, etc. Recent work on PADS has focused on developing mechanisms for learning data descriptions automatically (Fisher et al., 2008). Kennedy’s pickling combinators (2004) describe serializers and deserializers. Benton (2005) and Ramsey (2003) each proposed systems for mapping between the values in a host language and the run-time values manipulated by an embedded interpreter.

JT (Ennals and Gay, 2007) synchronizes programs written in different high level languages, such as C and Jekyll, an extension of C with features from ML. JT relies on a notion of distance to decide how to propagate modifications, allowing the detection of non local edits such as the swap of two functions. The synchronization seems to work well in many cases but there is no claim that the semantics of the synchronized programs are the same.

In the functional programming community, Wadler’s notion of \textit{views} (1987), extends algebraic pattern matching to abstract data types. Programmers supply explicit \textit{in} and \textit{out} functions to map between views and the underlying structures they are generated from.

\textbf{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 (1961) observed that non-injective functions were logically irreversible, and that this irreversibility requires the generation and dissipation of some heat per machine cycle. Bennet (1973) demonstrated that this irreversibility was not inevitable by constructing a reversible Turing machine, showing that thermodynamically reversible computers were plausible. Baker (1992) 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 (2004a).

8.3 Databases

Research on view update translation in the database literature 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. By contrast, we have designed a new language in which the definitions of get and put go hand-in-hand. Our approach also goes beyond classical work in the relational setting by directly transforming and updating arbitrary structures rather than just relations.

Work by Bohannon, Pierce, and Vaughan (2006) extends the lens framework described here to obtain lenses that operate natively on relational data. 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 (2003) articulates a similar perspective on translating relational updates.

Masunaga (1984) 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 (1985) 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. He later (1986) 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. Building on this foundation, Barsalou, Siambela, Keller, and Wiederhold (1991) described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa (1985) presented a design tool for exploring the effects of choosing a view update policy. This tool shows the update translation for update requests supplied by the user; by considering all possible valid sources, the tool predicts whether the desired update would in fact be reflected back into the view after applying the translated update to the concrete database.

Miller et al. (2001) describe Clio, a system for managing heterogeneous transformation and integration. Clio provides a tool for visualizing two schemas, specifying correspondences between fields, defining a mapping between the schemas, and viewing sample query results. They only consider the get direction of our lenses, but their system is somewhat mapping-agnostic, so it might eventually be possible to use a framework like Clio as a user interface for incremental lens programming.

Atzeni and Torlone (1997; 1996) described a tool for translating views and observed that if one can translate any source view to and from a meta-model (shared view), one then gets bidirectional transformations between any pair of sources. They limited themselves to mappings
where the source and view are isomorphic.

Complexity bounds have also been studied for various versions of the view update inference problem. Cosmadakis and Papadimitriou (1984) 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 (2002) 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 (2001), and Braganholo, Davidson, and Heuser (2003) and others studied the problem of updating relational databases “presented as XML.” Their solution requires a 1:1 mapping between XML view elements and objects in the database, to make updates unambiguous.

Tatarinov, Ives, Halevy, and Weld (2001) 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 (1991) 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 (Gupta et al., 1993)—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.
8.4 Model Transformations

In the model-driven approach to software development, formal models are used to derive implementations from specifications. In systems with many kinds of models developers need to maintain complex relationships among the models and code—e.g., refinement of design models to code and the 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 (Czarnecki and Helsen, 2006). Bidirectional model transformations are of particular interest if the related artifacts can be edited independently (Antkiewicz and Czarnecki, 2008). Formalisms such as triple graph grammars can be used to describe bidirectional transformations between models (Schürr, 1995). Recently, Stevens (2008a; 2008b; 2007) has applied lenses to model transformations. Similar ideas have also been pursued by Xiong, Liu, Hu, Zhao, Takeichi, and Mei (2007) and Hidaka, Hu, Kato, and Nakano (2009).

8.5 Security

Views have long been used in relational systems as a mechanism for enforcing confidentiality. They were first proposed as a security mechanism for XML data by Stoica and Farkas (2002) and were later studied extensively by Fan and his colleagues in a series of papers (2004; 2006; 2007). The key difference between previous work on security views and the framework proposed in Chapter 6, of course, is support for updates. Additionally, 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 (2004) 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., in the Intellipedia system discussed in Chapter 6, the regraded documents need 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 (1977) and has since been applied in a variety of languages, including Jif (Myers, 1999), a secure variant of Java, and FlowCaml (Pottier and Simonet, 2003), a secure variant of OCaml. The excellent survey article by Sabelfeld and Myers (2003) gives a general overview of the entire area and provides 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 (Benzaken et al., 2003). Foster, Green, and Tannen proposed a mechanism for ensuring non-interference properties of tree transformations using a semantics that propagates dynamic provenance annotations (Foster et al., 2008a). The Fable language also propagates security labels dynamically (Swamy et al., 2008; Corcoran et al., 2007). 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 (Cheney et al., 2007).

Integrity can be treated as a formal dual to confidentiality, as was first noted by Biba (1977). 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 (2003), 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 (Wall et al., 2000). Wassermann and Su proposed a more powerful approach based on a dynamic analysis of generated strings that tracks tainted data in PHP scripts (2007). Shankar, Talwar, Foster, and Wagner (2001) 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 (2009).
Chapter 9

Summary and Future Work

This dissertation demonstrates that bidirectional programming languages are an effective and elegant way of defining updatable views. Starting with foundations, we proposed lenses as a theoretical foundation for bidirectional languages. We then developed a concrete language for building lenses that operate on strings, with natural syntax based on the regular operators. We studied the special complications that arise when lenses are used with data containing extraneous details, ordered data, and confidential data. We described the practical implementation of our ideas in Boomerang.

These results do not come close to finishing the story on bidirectional languages. On the contrary, our work can be extended in many directions. We discuss several possibilities in this section.

9.1 Data Model

All of the lenses discussed in this dissertation manipulate strings. However, the semantic framework of lenses is completely generic—it can be instantiated with arbitrary structures. Basic lenses have been been applied to tree-structured (Foster et al., 2007b) and relational data (Bohannon et al., 2006). We are interested in continuing to instantiate the framework with other structures—e.g., complex values, objects, graphs, etc. Particular topics of interest include extending concepts such as chunks, which are central to the semantics of resourceful lenses in these richer settings. Another challenge will be developing type systems for that are powerful
enough to express the conditions necessary to ensure well behavedness.

9.2 Syntax

At the level of syntax, our investigation has focused exclusively on combinators. This low-level approach to language design has worked well as we explore fundamental issues (e.g., how standard constructs such as conditionals, products, iteration operators, etc. should work as lenses), but programmers find writing lenses in “point-free” style unnatural. We are interested in developing new lens languages based on more familiar forms of syntax. One promising idea is to start from the nested relational calculus (NRC) (Buneman et al., 1995). NRC has a rich data model that can represent many different structures and has a programming model that includes a powerful collection primitives based on comprehensions. The key technical challenge will be developing the machinery for interpreting NRC bidirectionally. We have several ideas for making progress: a semantic approach, similar to the one we have pursued in Boomerang, and an operational approach based on explicitly constructing complements—i.e., structures that contain all of the source information not reflected in the view.

A related topic is exploring the connections between view update and view maintenance. The goal of view maintenance systems is to propagate source updates to the view efficiently. Intuitively, many updates only affect a small portion of the view, so they can be translated to small incremental updates. Complements closely resemble the “trace” artifacts that have been proposed in the context of self-adjusting computation (Acar et al., 2006). 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 mechanisms for efficiently maintaining views constructed using lenses.

9.3 Optimization

Another area for further work is algebraic optimization. We are interested in developing a theory of our lens combinators that could serve as the basis for an optimizing compiler. For example, \((l_1^*; l_2^*)\) and \((l_1;l_2)^*\) behave the same as basic lenses (when both are well typed), but the second lens should run substantially faster. We would also like to explore streaming lenses.
This is motivated by large examples such as our lens for scientific data, where the size of sources is on the order of 1GB! We would like to develop a variant of the iteration combinator whose `get` function processes elements one at a time, rather than operating on a string representing the whole sequence. Similarly, the `put` function would operate on elements of the view one at a time. To optimize memory requirements, we would like to explore lenses that maintain minimal complements.

### 9.4 Security

Our secure lenses use static and dynamic mechanisms to ensure security properties of lenses. We would like to explore connections with other dynamic approaches—e.g., languages that propagate dynamic labels (Zheng and Myers, 2007; Shroff et al., 2007) as well as provenance metadata (Buneman et al., 2001; Foster et al., 2008a). We hope that these languages will suggest mechanisms for enforcing security properties at finer levels of granularity than our current, static, approach can track. We would also like to explore declassification operators (Myers and Liskov, 1997), quantitative measures of information flow (McCamant and Ernst, 2008), and formal notions of privacy (Miklau and Suciu, 2007) in the context of lenses.


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.


Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions.


172


173


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

\[
E \in \mathcal{R} \\
\text{copy } E \in [E] \iff [E]
\]

3.3.1 Lemma: Let \( E \in \mathcal{R} \) be a regular expression. Then \( \text{(copy } E \text{)} \) is a basic lens in \([E] \iff [E]\).

Proof:

- **GetPut**: Let \( e \) be a string in \([E]\). We calculate as follows

\[
\begin{align*}
\text{(copy } E \text{).put ((copy } E \text{).get } e \text{)} e \\
&= \text{(copy } E \text{).put } e \phantom{\text{.get } e} \quad \text{by definition (copy } E \text{).get} \\
&= e \phantom{\text{.get } e} \quad \text{by definition (copy } E \text{).put}
\end{align*}
\]

and obtain the required result.

- **PutGet**: Let \( e \) and \( e' \) be strings in \([E]\). We calculate as follows

\[
\begin{align*}
\text{(copy } E \text{).get ((copy } E \text{).put } e' \text{)} e \\
&= \text{(copy } E \text{).get } e' \phantom{\text{.put } e'} \quad \text{by definition (copy } E \text{).put} \\
&= e' \phantom{\text{.get } e'} \quad \text{by definition (copy } E \text{).get}
\end{align*}
\]

and obtain the required result.
CreateGet: Let \( e \) be a string in \([E]\). We calculate as follows

\[
\text{(copy } E\text{).get ((copy } E\text{).create } e) = (\text{copy } E\text{).get } e = e
\]

by definition \((\text{copy } E\text{).create}\) and \((\text{copy } E\text{).get}\)

and obtain the required result, which completes the proof.

\[
E \in \mathcal{R} \quad [E] \neq \emptyset \quad u \in \Sigma^* \\
\text{const } E \ u \in [E] \iff \{u\}
\]

3.3.2 Lemma: Let \( E \in \mathcal{R} \) be a regular expression and \( u \in \Sigma^* \) a string. Then \((\text{const } E \ u)\) is a basic lens in \([E] \iff \{u\}\).

Proof:

GetPut: Let \( e \in [E] \). We calculate as follows

\[
\text{(const } E \ u\text{).put ((const } E \ u\text{).get } e) \text{ e} = \text{(const } E \ u\text{).put } u \text{ e} \quad \text{by definition (const } E \ u\text{).get} \\
= e \quad \text{by definition (const } E \ u\text{).put}
\]

and obtain the required result.

PutGet: Let \( u \in \{u\} \) and \( e \in [E] \). We calculate as follows

\[
\text{(const } E \ u\text{).get ((const } E \ u\text{).put } u \text{ e)} \text{ e} = \text{(const } E \ u\text{).get } e \quad \text{by definition (const } E \ u\text{).put} \\
= u \quad \text{by definition (const } E \ u\text{).get}
\]

and obtain the required result.

CreateGet: Let \( u \in \{u\} \). We calculate as follows

\[
\text{(const } E \ u\text{).get ((const } E \ u\text{).create } u) = \text{(const } E \ u\text{).get } \text{representative}(E) \quad \text{by definition (const } E \ u\text{).create} \\
= u \quad \text{by definition (const } E \ u\text{).get}
\]

and obtain the required result, which completes the proof.
3.3.3 Lemma: Let \( l \in S \leftrightarrow V \) be a basic lens and \( f \in V \to S \) a function from \( S \) to \( V \). Then \( \text{default} \ l \ f \) is a basic lens in \( S \leftrightarrow V \).

Proof:

\( \text{GetPut:} \) Immediate by \text{GetPut} for \( l \).

\( \text{PutGet:} \) Immediate by \text{PutGet} for \( l \).

\( \text{CreateGet:} \) Immediate by \text{PutGet} for \( l \).

\[ \begin{array}{c|c}
S_1 \cdot S_2 & V_1 \cdot V_2 \\
\hline
l_1 \in S_1 \leftrightarrow V_1 & l_2 \in S_2 \leftrightarrow V_2 \\
(l_1 \cdot l_2) \in (S_1 \cdot S_2) \leftrightarrow (V_1 \cdot V_2)
\end{array} \]

3.3.4 Lemma: Let \( l_1 \in S_1 \leftrightarrow V_1 \) and \( l_2 \in S_2 \leftrightarrow V_2 \) be basic lenses such that \((S_1, S_2)\) and \((V_1, V_2)\). Then \((l_1 \cdot l_2)\) is a basic lens in \((S_1 \cdot S_2) \leftrightarrow (V_1 \cdot V_2)\).

Proof:

\( \text{GetPut:} \) 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

\[ (l_1 \cdot l_2).\text{put} ((l_1 \cdot l_2).\text{get} \ s) \]

\[ = (l_1 \cdot l_2).\text{put} ((l_1 \cdot l_2).\text{get} \ (s_1 \cdot s_2)) \ (s_1 \cdot s_2) \quad \text{by definition } s_1 \text{ and } s_2 \]

\[ = (l_1 \cdot l_2).\text{put} \ ((l_1.\text{get} \ s_1) \cdot (l_2.\text{get} \ s_2)) \ (s_1 \cdot s_2) \quad \text{by definition } (l_1 \cdot l_2).\text{get} \]

\[ = (l_1.\text{put} \ (l_1.\text{get} \ s_1) \ s_1) \cdot (l_2.\text{put} \ (l_2.\text{get} \ s_2) \ s_2) \quad \text{by definition } (l_1 \cdot l_2).\text{put} \text{ with cod}(l_1.\text{get}) = V_1 \text{ and cod}(l_2.\text{get}) = V_2 \text{ and } V_1 \cdot V_2 \]

\[ = (s_1 \cdot s_2) \quad \text{by GetPut for } l_1 \text{ and } l_2 \]

\[ = s \quad \text{by definition } s_1 \text{ and } s_2 \]

and obtain the required equality.
Lemma: 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

\[
\begin{align*}
(l_1 \cdot l_2) & . \text{get} \ ((l_1 \cdot l_2) . \text{put} \ v \ s) \\
= \ & (l_1 \cdot l_2) . \text{get} \ ((l_1 \cdot l_2) . \text{put} \ (v_1 \cdot v_2) \ (s_1 \cdot s_2)) \quad \text{by definition } v_1, v_2, s_1, \text{and } s_2 \\
= \ & (l_1 \cdot l_2) . \text{get} \ ((l_1 . \text{put} \ v_1 \ s_1) \cdot (l_2 . \text{put} \ v_2 \ s_2)) \quad \text{by definition } (l_1 \cdot l_2) . \text{put} \\
= \ & (l_1 . \text{get} \ (l_1 . \text{put} \ v_1 \ s_1)) \cdot (l_2 . \text{get} \ (l_2 . \text{put} \ v_2 \ s_2)) \quad \text{by definition } (l_1 \cdot l_2) . \text{get} \text{ with } \text{cod}(l_1 . \text{put}) = S_1 \\
& \quad \text{and } \text{cod}(l_2 . \text{put}) = S_2 \text{ and } S_1 \cdot S_2 \\
= \ & (v_1 \cdot v_2) \quad \text{by PutGet for } l_1 \text{ and } l_2 \\
= \ & v \quad \text{by definition } v_1 \text{ and } v_2
\end{align*}
\]

and obtain the required equality.

CreateGet: Similar to the proof for PutGet.

\[\begin{array}{c}
\begin{array}{c}
S_1 \cap S_2 = \emptyset \\
l_1 \in S_1 \iff V_1 \\
l_2 \in S_2 \iff V_2 \\
(l_1 | l_2) \in (S_1 \cup S_2) \iff (V_1 \cup V_2)
\end{array}
\end{array}\]

3.3.5 Lemma: Let \( l_1 \in S_1 \iff V_1 \) and \( l_2 \in S_2 \iff V_2 \) be basic lenses such that \( (S_1 \cap S_2) = \emptyset \). Then \( (l_1 | l_2) \) is a basic lens in \( (S_1 \cup S_2) \iff (V_1 \cup V_2) \).

Proof:

GetPut: Let \( s \in (S_1 \cup S_2) \). We analyze two cases.

Case \( s \in S_1 \): We calculate as follows

\[
\begin{align*}
(l_1 | l_2) . \text{put} \ ((l_1 | l_2) . \text{get} \ s) & \ s \\
= \ & (l_1 | l_2) . \text{put} \ (l_1 . \text{get} \ s) \ s \quad \text{by the definition of } (l_1 | l_2) . \text{get} \text{ with } s \in S_1 \\
= \ & l_1 . \text{put} \ (l_1 . \text{get} \ s) \ s \quad \text{by the definition of } (l_1 | l_2) . \text{put} \text{ with } \text{cod}(l . \text{get}) = V_1 \\
& \quad \text{and } s \in S_1 \\
= \ & s \quad \text{by PutGet for } l_1
\end{align*}
\]

and obtain the required equality, which finishes the case.
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

\[
(l_1 \mid l_2).\text{get}((l_1 \mid l_2).\text{put} v s)
\]
\[
= (l_1 \mid l_2).\text{get}(l_1.\text{put} v s)
\]
\[
= (l_1 \mid l_2).\text{get}(l_1.\text{create} v)
\]
\[
= l_1.\text{get}(l_1.\text{create} v)
\]
\[
= v
\]

by definition $(l_1 \mid l_2).\text{put}$ with $v \in V_1$ and $s \in S_1$

by definition $(l_1 \mid l_2).\text{get}$ with $\text{cod}(l_1.\text{put}) = S_1$

by PutGet for $l$

and obtain the required equality.

Case $v \in V_1$ 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

\[
(l_1 \mid l_2).\text{get}((l_1 \mid l_2).\text{put} v s)
\]
\[
= (l_1 \mid l_2).\text{get}(l_1.\text{create} v)
\]
\[
= l_1.\text{get}(l_1.\text{create} v)
\]
\[
= v
\]

by definition $(l_1 \mid l_2).\text{put}$ with $v \in (V_1 - V_2)$ and $s \in S_2$

by definition $(l_1 \mid l_2).\text{get}$ with $\text{cod}(l_1.\text{create}) = S_1$

by CreateGet for $l$

and obtain the required equality.

Case $v \in (V_2 - V_1)$ and $s \in S_1$: Symmetric to the previous case, using $l_2$ instead of $l_1$.

**CreateGet**: Let $v \in (V_1 \cup V_2)$. We analyze two cases.

Case $v \in V_1$: We calculate as follows

\[
(l_1 \mid l_2).\text{get}((l_1 \mid l_2).\text{create} v)
\]
\[
= (l_1 \mid l_2).\text{get}(l_1.\text{create} v)
\]
\[
= l_1.\text{get}(l_1.\text{create} v)
\]
\[
= v
\]

by definition $(l_1 \mid l_2).\text{create}$ with $v \in V_1$

by definition $(l_1 \mid l_2).\text{get}$ with $\text{cod}(l_1.\text{create}) = S_1$

by CreateGet for $l$

and obtain the required equality.

Case $v \in (V_2 - V_1)$: Symmetric to the previous case. □
3.3.6 Lemma: Let \( l \in S \iff V \) be a basic lens such that \( S^l \) and \( V^l \). Then \( l^* \) is a basic lens in \( S^* \iff V^* \).

Proof:

- **GetPut**: Let \( s \in S^* \). As \( S^l \) 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

\[
\begin{align*}
l^* \cdot \text{put} (l^* \cdot \text{get} s) s &= l^* \cdot \text{put} \left( l^* \cdot \text{get} (s_1 \cdots s_n) \right) (s_1 \cdots s_n) \\
&= l^* \cdot \text{put} \left( (l \cdot \text{get} s_1) \cdots (l \cdot \text{get} s_n) \right) (s_1 \cdots s_n) \quad \text{by definition } l^* \cdot \text{get} \\
&= l \cdot \text{put} (l \cdot \text{get} s_1) s_1 \cdots l \cdot \text{put} (l \cdot \text{get} s_n) s_n \quad \text{by definition } l^* \cdot \text{put} \text{ with } V^l \text{ and } \text{cod}(l \cdot \text{get}) = V \\
&= s_1 \cdots s_n \\
&= s \quad \text{by definition } s_1 \text{ to } s_n
\end{align*}
\]
and obtain the required equality.

- **PutGet**: Let \( v \in V^* \) and \( s \in S^* \). As \( V^l \) there exist unique strings \( v_1 \) to \( v_n \) in \( V \) such that \( v = (v_1 \cdots v_n) \). Similarly, as \( S^l \) 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

\[
\begin{align*}
l^* \cdot \text{get} (l^* \cdot \text{put} v s) &= l^* \cdot \text{get} \left( l^* \cdot \text{put} (v_1 \cdots v_n) \right) (s_1 \cdots s_m) \\
&= l^* \cdot \text{get} \left( s_1' \cdots s_n' \right) \quad \text{by the definition of } l^* \cdot \text{put} \\
&\quad \text{where } s_i' = \begin{cases} l \cdot \text{put} v_i & i \in \{1, \ldots, \min(n, m)\} \\ l \cdot \text{create} v_i & i \in \{m + 1, \ldots, n\} \end{cases} \\
&= (l \cdot \text{get} s_1') \cdots (l \cdot \text{get} s_n') \quad \text{by the definition of } l^* \cdot \text{get} \text{ with } V^l \text{ and } \text{cod}(l \cdot \text{put}) = \text{cod}(l \cdot \text{create}) = V \\
&= v_1 \cdots v_n \quad \text{by PutGet and CreateGet for } l \\
&= v \quad \text{by the definition of } v_1 \text{ to } v_n
\end{align*}
\]
and obtain the required equality.

- **CreateGet**: Similar to the proof for PutGet. \(\square\)
3.3.7 Lemma: Let \( l_1 \in S \iff U \) and \( l_2 \in U \iff V \) be basic lenses. Then \( (l_2;l_2) \) is a basic lens in \( S \iff V \).

Proof:

\[ \begin{align*}
    l_1 \in S &\iff U \quad l_2 \in U \iff V \\
    (l_1;l_2) \in S &\iff V
\end{align*} \]

\[ \begin{align*}
    (l_1;l_2).put ((l_1;l_2).get s) s &\quad \text{by definition \( (l_1;l_2).get \)} \\
    = (l_1;l_2).put (l_2.get (l_1.get s)) s &\quad \text{by definition \( (l_1;l_2).put \)} \\
    = l_1.put (l_2.put (l_2.get (l_1.get s)) (l_1.get s)) s &\quad \text{by definition \( (l_1;l_2).put \)} \\
    = l_1.put (l_1.get s) s &\quad \text{by GetPut for \( l_2 \)} \\
    = s &\quad \text{by GetPut for \( l_1 \)}
\end{align*} \]

and obtain the required equality.

\[ \begin{align*}
    (l_1;l_2).get ((l_1;l_2).put v s) &\quad \text{by definition \( (l_1;l_2).put \)} \\
    = (l_1;l_2).get (l_1.put (l_2.put v (l_1.get s)) s) &\quad \text{by definition \( (l_1;l_2).put \)} \\
    = l_2.get (l_1.get (l_1.put (l_2.put v (l_1.get s)) s)) &\quad \text{by definition \( (l_1;l_2).get \)} \\
    = l_2.get (l_2.put v (l_1.get s)) &\quad \text{by PutGet for \( l_1 \)} \\
    = v &\quad \text{by PutGet for \( l_2 \)}
\end{align*} \]

and obtain the required equality.

\[ \begin{align*}
    \text{CreateGet: Similar to the proof for PutGet.} \quad \Box
\end{align*} \]

\[ \begin{align*}
    [E] \cap [F] = \emptyset &\quad ([E] \cup [F])^* \\
    \text{filter \( E \; F \) \in ([E] \cup [F])^* \iff [E]^*}
\end{align*} \]

3.3.8 Lemma: Let \( E \) and \( F \) be regular expressions satisfying \( ([E] \cap [F]) = \emptyset \) and \( ([E] \cup [F])^* \).

Then \( \text{filter \( E \; F \) is a basic lens in \( ([E] \cup [F])^* \iff [E]^*} \)

Proof:
\[ \text{GetPut: Let } s \in ([E] \cup [F])^* \text{ be a string. As } ([E] \cup [F])^{1*}, \text{ there exist unique strings } s_1 \text{ to } s_n \text{ such that } s = (s_1 \cdots s_n). \text{ Let } (e_1, \cdots, e_k) \text{ denote the subsequence of } (s_1, \ldots, s_n) \text{ that belong to } E \text{ and let } (f_1, \cdots, f_l) \text{ denote the subsequence of } (s_1, \ldots, s_n) \text{ that belong to } F. \text{ Using these definitions, we calculate as follows} \]

\[
\begin{align*}
\text{GetPut} & = (\text{filter } E \ F).\text{put} (\text{filter } E \ F).\text{get } s \ s \\
& = (\text{filter } E \ F).\text{put} ((\text{filter } E \ F).\text{get } (s_1 \cdots s_n)) (s_1 \cdots s_n) \quad \text{by definition } s_1 \text{ to } s_n \\
& = (\text{filter } E \ F).\text{put} (\text{string滤器 } E (s_1 \cdots s_n)) (s_1 \cdots s_n) \quad \text{by definition } (\text{filter } E \ F).\text{get} \\
& = (\text{filter } E \ F).\text{put} (e_1 \cdots e_k) (s_1 \cdots s_n) \quad \text{by definition string滤器} \\
& = \text{string滤器 } F (e_1 \cdots e_k) (s_1 \cdots s_n) \quad \text{by definition } (\text{filter } E \ F).\text{put} \\
& = (s_1 \cdots s_n) \quad \text{by definition string滤器} \\
& = s \quad \text{by definition } s_1 \text{ to } s_n \\
\end{align*}
\]

and obtain the required equality.

\[ \text{PutGet: Let } s \in ([E] \cup [F])^* \text{ and } e \in [E]^* \text{ be strings. As } ([E] \cup [F])^{1*}, \text{ there exist unique strings } s_1 \text{ to } s_n \text{ such that } s = (s_1 \cdots s_n) \text{ and } e_1 \text{ to } e_k \text{ such that } e = (e_1 \cdots e_k). \text{ Using these definitions, we calculate as follows} \]

\[
\begin{align*}
\text{PutGet} & = (\text{filter } E \ F).\text{get} ((\text{filter } E \ F).\text{put } e \ s) \\
& = (\text{filter } E \ F).\text{get} ((\text{filter } E \ F).\text{put } e_1 \cdots e_k (s_1 \cdots s_n)) \quad \text{by definition } s_1 \text{ to } s_n \text{ and } e_1 \text{ to } e_k \\
& = (\text{filter } E \ F).\text{get} (\text{string滤器 } F (e_1 \cdots e_k) (s_1 \cdots s_n)) \quad \text{by definition } (\text{filter } E \ F).\text{put} \\
& = \text{string滤器 } F (\text{string滤器 } F (e_1 \cdots e_k) (s_1 \cdots s_n)) \quad \text{by definition } (\text{filter } E \ F).\text{get} \\
& = (e_1 \cdots e_k) \quad \text{by definition string滤器} \\
& = e \quad \text{by definition } e_1 \text{ to } e_k \\
\end{align*}
\]

and obtain the required equality.

\[ \text{CreateGet: Similar to the proof for PutGet.} \]

\[
\begin{array}{c|c}
S_1 \cdot S_2 & V_2 \cdot V_1 \\
\hline
l_1 \in S_1 & l_2 \in S_2 \quad \text{by definition } 1 \\
l_1 \sim l_2 \in (S_1 \cdot S_2) & (V_1 \cdot V_2) \quad \text{by definition } 2
\end{array}
\]

\[ \text{3.3.9 Lemma: Let } l_1 \in S_1 \iff V_1 \text{ and } l_2 \in S_2 \iff V_2 \text{ be basic lenses such that } (S_1 \cdot S_2) \text{ and } (V_2 \cdot V_1). \text{ Then } (l_1 \sim l_2) \text{ is a basic lens in } (S_1 \cdot S_2) \iff (V_1 \cdot V_2). \]

\[ \boxed{186} \]
Proof:

▶ 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

$$(l_1 \sim l_2).put ((l_1 \sim l_2).get s) s$$

$$= (l_1 \sim l_2).put ((l_1 \sim l_2).get (s_1 \cdot s_2)) (s_1 \cdot s_2) \quad \text{by definition } s_1 \text{ and } s_2$$

$$= (l_1 \sim l_2).put ((l_2.get s_2)\cdot(l_1.get s_1)) (s_1 \cdot s_2) \quad \text{by definition } (l_1 \sim l_2).get$$

$$= (l_1.put (l_1.get s_1) s_1)\cdot(l_2.put (l_2.get s_2) s_2) \quad \text{by definition } (l_1 \sim l_2).put \text{ with } V_2 \cdot V_1$$

and $\text{cod}(l_1.get) = V_1$

and $\text{cod}(l_2.get) = V_2$

$$= (s_1 \cdot s_2) \quad \text{by GetPut for } l_1 \text{ and } l_2$$

$$= s \quad \text{by definition } s_1 \text{ and } s_2$$

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

$$(l_1 \sim l_2).get ((l_1 \sim l_2).put v) s$$

$$= (l_1 \sim l_2).get ((l_1 \sim l_2).put (v_2 \cdot v_1) (s_1 \cdot s_2)) \quad \text{by definition } v_2, v_1, s_1, \text{ and } s_2$$

$$= (l_1 \sim l_2).get ((l_1.put v_1 s_1)\cdot(l_2.put v_2 s_2)) \quad \text{by definition } (l_1 \sim l_2).put$$

$$= (l_2.get (l_2.put v_2 s_2))\cdot(l_1.get (l_1.put v_1 s_2)) \quad \text{by definition } (l_1 \sim l_2).get \text{ with } S_1 \cdot S_2$$

and $\text{cod}(l_1.put) = S_1$

and $\text{cod}(l_2.put) = S_2$

$$= (v_2 \cdot v_1) \quad \text{by PutGet for } l_2 \text{ and } l_1$$

$$= v \quad \text{by definition } v_2 \text{ and } v_1$$

and obtain the required equality.

▶ CreateGet: Similar to the proof for PutGet. 

\[\square\]
Quotient Lens Proofs

4.2.1 Lemma: Let \( l \in S \iff V \) be a basic lens. Then \( \text{lift} l \) is a quotient lens in \( S/\sim \iff V/\sim \).

Proof:

- **GetEquiv**: Let \( s \in S \) and \( s' \in S \) be strings such that \( s = s' \). We immediately have \( ((\text{lift} l).\text{get} s) = ((\text{lift} l).\text{get} s') \).
- **PutEquiv**: Let \( v \in V \) and \( v' \in V \) be strings such that \( v = v' \) and, likewise, let \( s \in S \) and \( s' \in S \) be strings such that \( s = s' \). We immediately have \( ((\text{lift} l).\text{put} v s) = ((\text{lift} l).\text{get} v' s') \).
- **CreateEquiv**: Let \( v \in V \) and \( v' \in V \) be strings such that \( v = v' \). We immediately have \( ((\text{lift} l).\text{create} v) = ((\text{lift} l).\text{create} v') \).
- **GetPut**: Immediate by (the basic lens version of) \( \text{GETPUT} \) for \( l \).
- **PutGet**: Immediate by (the basic lens version of) \( \text{PUTGET} \) for \( l \).
- **CreateGet**: Immediate by (the basic lens version of) \( \text{CREATEGET} \) for \( l \).

\[
\begin{array}{cccc}
q & \in & S & \leftrightarrow U/\sim_U \\
l & \in & U/\sim_U & \iff V/\sim_V \\
s & \sim_S s' & \iff q.\text{canonize} s \sim_U q.\text{canonize} s' \\
l\text{quot} q l & \in & S/\sim_S & \iff V/\sim_V \\
\end{array}
\]

4.2.3 Lemma: Let \( q \in S \iff U/\sim_U \) be a canonizer and \( l \in U/\sim_U \iff V/\sim_V \) be a quotient lens. Then \((l\text{quot} q \ l)\) is a quotient lens in \( S/\sim_S \iff V/\sim_V \) where \( s \sim_S s' \) if and only if \( q.\text{canonize} s \sim_U q.\text{canonize} s' \).

Proof:

- **GetEquiv**: Let \( s \) and \( s' \) be strings in \( S \) such that \( s \sim_S s' \). By the definition of \( \sim_S \) we have

\[
q.\text{canonize} s \sim_U q.\text{canonize} s'
\]
Using this equivalence, we calculate as follows

\[(l\text{put } q \text{ } l).\text{get } s \]
\[= l.\text{get } (q.\text{canonize } s) \quad \text{by definition } (l\text{put } q \text{ } l).\text{get} \]
\[\sim_V l.\text{get } (q.\text{canonize } s') \quad \text{by GetEquiv for } l \]
\[= (l\text{put } q \text{ } l).\text{get } s' \quad \text{by definition } (l\text{put } q \text{ } l).\text{get} \]

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

\[(l\text{put } q \text{ } l).\text{put } v \text{ } s \sim_S (l\text{put } q \text{ } l).\text{put } v' \text{ } s' \]

by showing that

\[q.\text{canonize } ((l\text{put } q \text{ } l).\text{put } v \text{ } s) \sim_U q.\text{canonize } ((l\text{put } q \text{ } l).\text{put } v' \text{ } s') \]

By the definition of \( \sim_S \) we have

\[q.\text{canonize } s \sim_U q.\text{canonize } s' \]

Using this fact, we calculate as follows

\[q.\text{canonize } ((l\text{put } q \text{ } l).\text{put } v \text{ } s) \]
\[= q.\text{canonize } (q.\text{choose } (l.\text{put } v \text{ } (q.\text{canonize } s))) \quad \text{by definition } (l\text{put } q \text{ } l).\text{put} \]
\[\sim_U l.\text{put } v \text{ } (q.\text{canonize } s) \quad \text{by ReCanonize for } q \]
\[\sim_U l.\text{put } v' \text{ } (q.\text{canonize } s') \quad \text{by PutEquiv for } l \]
\[\sim_U q.\text{canonize } (q.\text{choose } (l.\text{put } v' \text{ } (q.\text{canonize } s'))) \quad \text{by ReCanonize for } q \]
\[= q.\text{canonize } ((l\text{put } q \text{ } l).\text{put } v' \text{ } s') \quad \text{by definition } (l\text{put } q \text{ } l).\text{put} \]

The required equivalence follows using transitivity and the definition of \( \sim_S \). (Throughout the rest of this dissertation, we will silently use elementary facts about equivalence relations, such as the transitivity of \( \sim_U \).)

**CreateEquiv:** Similar to the proof for PutEquiv.

**GetPut:** Let \( s \in S \) be a string. We will prove that

\[(l\text{put } q \text{ } l).\text{put } ((l\text{put } q \text{ } l).\text{get } s) \text{ } s \sim_S s \]
by showing that

\[ q.\text{canonize} ((l\text{quot } q \text{ l}).\text{put} ((l\text{quot } q \text{ l}).\text{get } s) s) \sim_U q.\text{canonize } s \]

We calculate as follows, abbreviating \( q.\text{canonize} \) as \( cn \)

\[
\begin{align*}
    cn ((l\text{quot } q \text{ l}).\text{put} ((l\text{quot } q \text{ l}).\text{get } s) s) \\
    &= cn (q.\text{choose} (l.\text{put} ((l\text{quot } q \text{ l}).\text{get } s) (cn s))) \quad \text{by definition} (l\text{quot } q \text{ l}).\text{put} \\
    \sim_U l.\text{put} ((l\text{quot } q \text{ l}).\text{get } s) (cn s) \quad \text{by \text{ReCanonize} for } q \\
    &= l.\text{put} (l.\text{get} (cn s)) (cn s) \quad \text{by definition} (l\text{quot } q \text{ l}).\text{get} \\
    \sim_U cn s \quad \text{by \text{GetPut} for } l
\end{align*}
\]

and obtain the required equivalence.

\textbf{PutGet:} Let \( v \in V \) and \( s \in S \) be strings. By \text{ReCanonize} for \( q \) we have

\[ q.\text{canonize} (q.\text{choose} (l.\text{put} v (q.\text{canonize } s))) \sim_U l.\text{put} v (q.\text{canonize } s) \]

Using this equivalence, we calculate as follows, abbreviating \( q.\text{canonize} \) as \( cn \)

\[
\begin{align*}
    (l\text{quot } q \text{ l}).\text{get} ((l\text{quot } q \text{ l}).\text{put } v s) \\
    &= l.\text{get} (cn (q.\text{choose} (l.\text{put} v (cn s)))) \quad \text{by definition} (l\text{quot } q \text{ l}).\text{get} \text{ and } (l\text{quot } q \text{ l}).\text{put} \\
    \sim_V l.\text{get} (l.\text{put} v (cn s)) \quad \text{by \text{GetEquiv} for } l \\
    \sim_V v \quad \text{by \text{PutGet} for } l
\end{align*}
\]

and obtain the required equivalence.

\textbf{CreateGet:} Analogous to the previous case (using \text{CreateGet} for \( l \) instead of \text{PutGet}). \qed

\[
\begin{array}{c}
    l \in S/\sim_S \iff U/\sim_U \quad q \in V \leftrightarrow U/\sim_U \\
    v \sim_V v' \iff q.\text{canonize } v \sim_U q.\text{canonize } v' \quad r\text{quot } l \ q \in S/\sim_S \iff V/\sim_V
\end{array}
\]

4.2.4 Lemma: Let \( l \in S/\sim_S \iff U/\sim_U \) be a quotient lens and \( q \in V \leftrightarrow U/\sim_U \) a can-
onizer. Then \( r\text{quot } l \ q \) is a quotient lens in \( S/\sim_S \iff V/\sim_V \) where \( v \sim_V v' \) if and only if \( q.\text{canonize } v \sim_U q.\text{canonize } v' \).
Proof:

**GetEquiv:** Let \( s \in S \) and \( s' \in S \) be strings such that \( s \sim_S s' \). We will prove that

\[
(rquot l q).get \ s \sim_V (rquot l q).get \ s'
\]

by showing that

\[
q.canonize ((rquot l q).get \ s) \sim_U q.canonize ((rquot l q).get \ s')
\]

We calculate as follows

\[
q.canonize ((rquot l q).get \ s)
= q.canonize (q.choose (l.get \ s)) \quad \text{by definition} \ (rquot l q).get
\sim_U l.get \ s \quad \text{by ReCANONIZE for } q
\sim_U l.get \ s' \quad \text{by GetEquiv for } l
\sim_U q.canonize (q.choose (l.get \ s')) \quad \text{by ReCANONIZE for } q
= q.canonize ((rquot l q).get \ s') \quad \text{by definition} \ (rquot l q).get
\]

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

\[
q.canonize \ v \sim_U q.canonize \ v'
\]

Using this fact, we calculate as follows

\[
(rquot l q).put \ v \ s
= l.put (q.canonize \ v) \ s \quad \text{by definition} \ (rquot l q).put
\sim_S l.put (q.canonize \ v') \ s' \quad \text{by PutEquiv for } l
= (rquot l q).put \ v' \ s' \quad \text{by definition} \ (rquot l q).put
\]

and obtain the required equivalence.

**CreateEquiv:** Similar to the proof for PutGet.

**GetPut:** Let \( s \in S \). We calculate as follows

\[
(rquot l q).put \ ((rquot l q).get \ s) \ s
= l.put (q.canonize (q.choose (l.get \ s))) \ s \quad \text{by definition} \ (rquot l q).get \text{ and } (rquot l q).put
\sim_S l.put (l.get \ s) \ s \quad \text{by ReCANONIZE for } q \text{ and PutEquiv for } l
\sim_S s \quad \text{by GetPut for } l
\]
and obtain the required equivalence.

**PutGet**: Let \( v \in V \) and \( s \in S \). We will show that

\[
(r\quot l \ q).\text{get} \ ((r\quot l \ q).\text{put} \ v \ s) \sim_V v
\]

by showing that

\[
q.\text{canonize} \ ((r\quot l \ q).\text{get} \ ((r\quot l \ q).\text{put} \ v \ s)) \sim_U q.\text{canonize} \ v
\]

We calculate as follows

\[
q.\text{canonize} \ ((r\quot l \ q).\text{get} \ ((r\quot l \ q).\text{put} \ v \ s))
\]

\[
= q.\text{canonize} \ (q.\text{choose} \ (l.\text{get} \ ((r\quot l \ q).\text{put} \ v \ s))) \quad \text{by definition (r\quot l \ q).get}
\]

\[
\sim_U l.\text{get} \ ((r\quot l \ q).\text{put} \ v \ s) \quad \text{by ReCanonize for } q
\]

\[
= l.\text{get} \ (l.\text{put} \ (q.\text{canonize} \ v) \ s) \quad \text{by definition (r\quot l \ q).put}
\]

\[
\sim_U (q.\text{canonize} \ v) \quad \text{By PutGet for } l
\]

and obtain the desired equivalence.

**CreateGet**: Similar to the proof for PutGet.

4.2.5 **Lemma**: Let \( q \in V \leftrightarrow U/\sim_U \) 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 \( V \leftrightarrow U/\sim_U' \).

**Proof:**

**ReCanonize**: Let \( u \in U \) be a string. As \( q \in V \leftrightarrow U/\sim_U \) we have \( q.\text{canonize} \ (q.\text{choose} \ u) \sim_U u \). Since \( \sim_U \) refines \( \sim_U' \) we immediately have that \( q.\text{canonize} \ (q.\text{choose} \ u) \sim_U' u \), as required.

4.2.6 **Lemma**: Let \( l_1 \in S/\sim_S \leftrightarrow U/\sim_U \) and \( l_2 \in U/\sim_U \leftrightarrow V/\sim_V \) be quotient lenses. Then \( (l_2;l_2) \) is a quotient lens in \( S/\sim_S \leftrightarrow V/\sim_V \).
Proof:

- **GetEquiv:** Let \( s \in S \) and \( s' \in S \) be strings such that \( s \sim_S s' \). We calculate as follows

\[
(l_1;l_2).\text{get } s = l_2.\text{get } (l_1.\text{get } s) \quad \text{by definition } (l_1;l_2).\text{get}
\]

\[
\sim_V l_2.\text{get } (l_1.\text{get } s') \quad \text{by GetEquiv for } l_1 \text{ and } l_2
\]

\[
= (l_1;l_2).\text{get } s' \quad \text{by definition } (l_1;l_2).\text{get}
\]

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 \( s \sim_S s' \). We calculate as follows

\[
(l_1;l_2).\text{put } v s = l_1.\text{put } (l_2.\text{put } s (l_1.\text{get } s)) s \quad \text{by definition } (l_1;l_2).\text{put}
\]

\[
\sim_S l_1.\text{put } (l_2.\text{put } v' (l.\text{get } s')) s' \quad \text{by GetEquiv for } l_1 \text{ and PutEquiv for } l_1 \text{ and } l_2
\]

\[
= (l_1;l_2).\text{put } v' s' \quad \text{by definition } (l_1;l_2).\text{put}
\]

and obtain the required equivalence.

- **CreateEquiv:** Similar to the proof for PutGet.

- **GetPut:** Let \( S \in S \) be a string. We calculate as follows

\[
(l_1;l_2).\text{put } ((l_1;l_2).\text{get } s) s = l_1.\text{put } (l_2.\text{put } (l_2.\text{get } (l_1.\text{get } s)) (l_1.\text{get } s)) s \quad \text{by definition } (l_1;l_2).\text{get} \text{ and } (l_1;l_2).\text{put}
\]

\[
\sim_S l.\text{put } (l.\text{get } s) s \quad \text{by GetPut for } l_2 \text{ and PutEquiv for } l_1
\]

\[
\sim_S s \quad \text{by GetPut for } l
\]

and obtain the required equivalence.

- **PutGet:** Let \( v \in V \) and \( s \in S \). We calculate as follows

\[
(l_1;l_2).\text{get } ((l_1;l_2).\text{put } v s) = l_2.\text{get } (l_1.\text{get } (l_1.\text{put } (l_2.\text{put } v (l.\text{get } s)) s)) \quad \text{by definition } (l_1;l_2).\text{get} \text{ and } (l_1;l_2).\text{put}
\]

\[
\sim_V l_2.\text{get } (l_2.\text{put } v (l_1.\text{get } s)) \quad \text{by PutGet for } l_1 \text{ and GetEquiv for } l_2
\]

\[
\sim_V v \quad \text{by PutGet for } l_2
\]

and obtain the required equivalence.

- **CreateGet:** Similar to the proof for PutGet.
Lemma: Let \( l \in S/\sim_S \iff U/\sim_U \) be a quotient lens. Then \((\text{canonizer of lens } l)\) is a canonizer in \( S \iff U/\sim_U \).

Proof:

- **ReCanitize:**

  Let \( u \in U \) be a string. We calculate as follows, abbreviating \((\text{canonizer of lens } l)\) as \( q \)

  \[
  q.\text{canonize}(q.\text{choose } u) = l.\text{get}(l.\text{create } u)
  \]
  by definition \( q.\text{canonize} \) and \( q.\text{choose} \)

  \[
  \sim_U u
  \]
  by CREATEGET for \( l \)

  and obtain the required equivalence.

\[\square\]

\[\begin{array}{ccc}
S_1 \cdot S_2 & V_1 \cdot V_2 \\
l_1 \in S_1/\sim_S & V_1/\sim_V \\
l_2 \in S_2/\sim_S & V_2/\sim_V \\
\sim_S = \sim_{S_1} \cdot \sim_{S_2} & \sim_V = \sim_{V_1} \cdot \sim_{V_2} \\
l_1 \cdot l_2 \in (S_1 \cdot S_2)/\sim_S & (V_1 \cdot V_2)/\sim_V
\end{array}\]

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 \((S_1, l_1)\) and \((V_1, l_2)\). Then \((l_1 \cdot l_2)\) is a quotient lens in \((S_1 \cdot S_2)/\sim_S \iff (V_1 \cdot V_2)/\sim_V\) where \(\sim_S \triangleq (\sim_{S_1} \cdot \sim_{S_2})\) and \(\sim_V \triangleq (\sim_{V_1} \cdot \sim_{V_2})\).

Proof:

- **GetEquiv:** Let \( s \in (S_1 \cdot S_2) \) and \( s' \in (S_1 \cdot S_2) \) be strings such that \( s \sim_S s' \). As \( S_1 \cdot S_2 \) there exist unique strings \( s_1 \in S_1 \) and \( s_2 \in S_2 \) and \( s'_1 \in S_1 \) and \( s'_2 \in S_2 \) such that \( s = (s_1 \cdot s_2) \) and \( s' = (s'_1 \cdot s'_2) \). Also, 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 and definitions, we calculate as follows

\[(l_1 \cdot l_2).\text{get } s\]
\[= (l_1.\text{get } s_1) \cdot (l_2.\text{get } s_2) \quad \text{by definition } (l_1 \cdot l_2).\text{get}\]
\[\sim_V (l_1.\text{get } s'_1) \cdot (l_2.\text{get } s'_2) \quad \text{by GETEQUIV for } l_1 \text{ and } l_2\]
\[= (l_1 \cdot l_2).\text{get } s'\]
\[\quad \text{by definition } (l_1 \cdot l_2).\text{get}\]

and obtain the required equivalence.

- **PutEquiv**: Let \(v \in (V_1 \cdot V_2)\) and \(v' \in (V_1 \cdot V_2)\) be strings such that \(v \sim_V v'\) and similarly, let \(s \in (S_1 \cdot S_2)\) and \(s' \in (S_1 \cdot S_2)\) be strings such that \(s \sim_S s'\). As \(V_1 \cdot V_2\) there exist unique strings \(v_1 \in V_1\) and \(v_2 \in V_2\) and \(v'_1 \in V_1\) and \(v'_2 \in V_2\) such that \(v = (v_1 \cdot v_2)\) and \(v' = (v'_1 \cdot v'_2)\). Likewise, as \(S_1 \cdot S_2\) there exist unique strings \(s_1 \in S_1\) and \(s_2 \in S_2\) and \(s'_1 \in S_1\) and \(s'_2 \in S_2\) such that \(s = (s_1 \cdot s_2)\) and \(s' = (s'_1 \cdot s'_2)\). Moreover, by the definition of \(\sim_V\) we have that \(v_1 \sim_{V_1} v'_1\) and \(v_2 \sim_{V_2} v'_2\) and similarly, that \(s_1 \sim_{S_1} s'_1\) and \(s_2 \sim_{S_2} s'_2\). Using these facts and definitions, we calculate as follows

\[(l_1 \cdot l_2).\text{put } v \ s\]
\[= (l_1 \cdot l_2).\text{put } (v_1 \cdot v_2) \ (s_1 \cdot s_2) \quad \text{by definition } v_1 \text{ and } v_2 \text{ and } s_1 \text{ and } s_2\]
\[= (l_1.\text{put } v_1 \ s_1) \cdot (l_2.\text{put } v_2 \ s_2) \quad \text{by definition } (l_1 \cdot l_2).\text{put}\]
\[\sim_S (l_1.\text{put } v'_1 \ s'_1) \cdot (l_2.\text{put } v'_2 \ s'_2) \quad \text{by PUTEQUIV for } l_1 \text{ and } l_2\]
\[= (l_1 \cdot l_2).\text{put } (v'_1 \cdot v'_2) \ (s'_1 \cdot s'_2) \quad \text{by definition } (l_1 \cdot l_2).\text{put}\]
\[= (l_1 \cdot l_2).\text{put } v' \ s'\]
\[\quad \text{by definition } v_1 \text{ and } v_2 \text{ and } s_1 \text{ and } s_2\]

and obtain the required equivalence.

- **CreateEquiv**: Similar to the proof for PutEquiv.

- **GetPut**: Let \(s \in (S_1 \cdot S_2)\) be a string. 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 this fact, we calculate as follows

\[
(l_1 \cdot l_2).\text{put} ((l_1 \cdot l_2).\text{get} s) s
\]

\[
= (l_1 \cdot l_2).\text{put} ((l_1 \cdot l_2).\text{get} (s_1 \cdot s_2)) (s_1 \cdot s_2) \quad \text{by definition } s_1 \text{ and } s_2
\]

\[
= (l_1 \cdot l_2).\text{put} ((l_1.\text{get} s_1) \cdot (l_2.\text{get} s_2)) (s_1 \cdot s_2) \quad \text{by definition } (l_1 \cdot l_2).\text{get}
\]

\[
= (l_1.\text{put} (l_1.\text{get} s_1)) \cdot (l_2.\text{put} (l_2.\text{get} s_2)) \quad \text{by definition } (l_1 \cdot l_2).\text{put} \text{ with } V_1 \cdot V_2
\]

\[
\text{and cod}(l_1.\text{get}) = V_1 \text{ and cod}(l_2.\text{get}) = V_2
\]

\[
\sim S s_1 \cdot s_2 \quad \text{By GetPut for } l_1 \text{ and } l_2 \text{ and definition of } \sim S
\]

\[
eq s \quad \text{by definition } s_1 \text{ and } s_2
\]

and obtain the required equivalence.

\[\begin{array}{c}
\text{\textbf{PutGet:}} \text{ Let } v \in (V_1 \cdot V_2) \text{ and } s \in (S_1 \cdot S_2) \text{ be strings. As } V_1 \cdot V_2 \text{ there exist unique strings } v_1 \in V_1 \text{ and } v_2 \in V_2 \text{ such that } v = (v_1 \cdot v_2). \text{ Likewise, as } S_1 \cdot S_2 \text{ there exist unique strings } s_1 \in S_1 \text{ and } s_2 \in S_2 \text{ such that } s = (s_1 \cdot s_2). \text{ Using these facts, we calculate as follows}
\end{array}\]

\[
(l_1 \cdot l_2).\text{get} ((l_1 \cdot l_2).\text{put} v s)
\]

\[
= (l_1 \cdot l_2).\text{get} ((l_1 \cdot l_2).\text{put} (v_1 \cdot v_2) (s_1 \cdot s_2)) \quad \text{by definition } v_1 \text{ and } v_2 \text{ and } s_1 \text{ and } s_2
\]

\[
= (l_1 \cdot l_2).\text{get} ((l_1.\text{put} v_1 s_1) \cdot (l_2.\text{put} v_2 s_2)) \quad \text{by definition } (l_1 \cdot l_2).\text{put}
\]

\[
= (l_1.\text{get} (l_1.\text{put} v_1 s_1)) \cdot (l_2.\text{get} (l_2.\text{put} v_2 s_2)) \quad \text{by definition } (l_1 \cdot l_2).\text{get} \text{ with } S_1 \cdot S_2
\]

\[
\text{and cod}(l_1.\text{put}) = S_1 \text{ and cod}(l_2.\text{put}) = S_2
\]

\[
\sim V v_1 \cdot v_2
\]

\[
= v \quad \text{By PutGet for } l_1 \text{ and } l_2
\]

\[
\text{by definition } v_1 \text{ and } v_2
\]

and obtain the required equivalence.

\[\begin{array}{c}
\text{\textbf{CreateGet:}} \text{ Similar to the proof for PutGet.}
\end{array}\]

\[
\begin{array}{|c|c|c|}
\hline
l \in S/\sim_S & V/\sim_V & S^* \quad V^* \\
\hline
l^* \in S^*/(\sim_S^*) & V^*/(\sim_V^*) & \\
\hline
\end{array}
\]

\[\begin{array}{c}
\textbf{4.3.4 Lemma:} \text{ Let } l \in S/\sim_S \iff V/\sim_V \text{ be a quotient lens such that } S^* \text{ and } V^*. \text{ Then } l^* \text{ is a quotient lens in } S^*/\sim_s \iff V^*/\sim_v.
\end{array}\]

\[\begin{array}{c}
\text{Proof:}
\end{array}\]

196
GetEquiv: Let \( s \in S^* \) and \( s' \in S^* \) be strings \( s \sim s' \). As \( S^* \) there exist unique strings \( s_1 \in S \) to \( s_n \in S \) and \( s'_1 \in S \) to \( s'_m \in S \) such that \( s = (s_1 \cdots s_n) \) and \( s' = (s'_1 \cdots s'_m) \). Moreover, by the definition of \( \sim_S \) we have that \( (n = m) \) and \( s_i \sim_C s'_i \) for \( i \in \{1, \ldots, n\} \). Using these facts, we calculate as follows

\[
\begin{align*}
l^*\cdot \text{get } s &= l^*\cdot \text{get } (s_1 \cdots s_n) \quad \text{by definition } s_1 \text{ to } s_n \\
&= (l\cdot \text{get } s_1) \cdots (l\cdot \text{get } s_n) \quad \text{by definition } l^*\cdot \text{get} \\
\sim_{V^*} (l\cdot \text{get } s'_1) \cdots (l\cdot \text{get } s'_n) &= \text{GetEquiv for } l \text{ (n times)} \\
&= l^*\cdot \text{get } (s'_1 \cdots s'_n) \quad \text{by definition } l^*\cdot \text{get} \\
&= l^*\cdot \text{get } s' \quad \text{by definition } s'_1 \text{ to } s'_n
\end{align*}
\]

and obtain the required equivalence.

PutEquiv: Let \( v \in V^* \) and \( v' \in V^* \) such that \( v \sim v' \) and similarly, let \( S \in S^* \) and \( s' \in S^* \) such that \( s \sim s' \). As \( V^* \) there exist unique strings \( v_1 \in V \) to \( v_n \in V \) and \( v'_1 \in V \) to \( v'_m \in V \) such that \( v = (v_1 \cdots v_n) \) \( v' = (v'_1 \cdots v'_n) \). Similarly, as \( \text{uniqueiter} S \) there exist unique strings \( s_1 \in V \) to \( s_o \in V \) and \( s'_1 \in V \) to \( s'_p \in V \) such that \( s = (s_1 \cdots s_o) \) \( s' = (s'_1 \cdots s'_p) \). Moreover, by the definition of \( \sim_V \) we have \( (n = m) \) and \( v_i \sim_v v'_i \) for \( i \in \{1, \ldots, n\} \). Also, by the definition of \( \sim_S \) we have \( (o = p) \) and \( s_i \sim_S s'_i \) for \( i \in \{1, \ldots, o\} \). Using these facts and definitions, we calculate as follows

\[
\begin{align*}
l^*\cdot \text{put } v \ s &= l^*\cdot \text{put } (v_1 \cdots v_n) \ (s_1 \cdots s_o) \quad \text{by definition } v_1 \text{ to } v_n \text{ and } s_1 \text{ to } s_o \\
&= s_1'' \cdots s_n'' \quad \text{by definition } l^*\cdot \text{put} \\
\text{where } s_i'' &= \begin{cases} 
\text{l\cdot put } v_i \ s_i \text{ for } i \in \{1, \ldots, \max(n,o)\} \\
\text{l\cdot create } v_i \text{ for } i \in \{\max(n,o) + 1, \ldots, o\}
\end{cases} \\
\sim_S \ s_1'' \cdots s_n'' &= \text{PutEquiv for } l \text{ (n times)} \\
\text{where } s_i'' &= \begin{cases} 
\text{l\cdot put } v'_i \ s'_i \text{ for } i \in \{1, \ldots, \max(n,o)\} \\
\text{l\cdot create } v'_i \text{ for } i \in \{\max(n,o) + 1, \ldots, o\}
\end{cases} \\
&= l^*\cdot \text{put } (v'_1 \cdots v'_n) \ (s'_1 \cdots s'_o) \quad \text{by definition } v'_1 \text{ to } v'_n \text{ and } s'_1 \text{ to } s'_o \\
&= l^*\cdot \text{put } a' \ c' \quad \text{by definition } l^*\cdot \text{put}
\end{align*}
\]

and obtain the required equivalence.

CreateEquiv: Similar to the proof for PutEquiv.
\textbf{GetPut:} Let \( s \in S^* \). By \( S^t_1 \) there exist unique strings \( s_1 \in S \) to \( s_n \in S \) such that \( s = (s_1 \cdots s_n) \). We calculate as follows

\[
\begin{align*}
l^*.\text{put} (l^*.\text{get} \ s) & = l^*.\text{put} (l^*.\text{get} (s_1 \cdots s_n)) (s_1 \cdots s_n) \quad \text{by definition } s_1 \text{ to } s_n \\
& = l^*.\text{put} ((l^*.\text{get} s_1) \cdots (l^*.\text{get} s_n)) (s_1 \cdots s_n) \quad \text{by definition } l^*.\text{get} \\
& = (l^*.\text{put} (l^*.\text{get} s_1) s_1) \cdots (l^*.\text{put} (l^*.\text{get} s_n) s_n) \quad \text{by definition } l^*.\text{put} \text{ with } V^t_1 \\
& \sim_{S^t} s_1 \cdots s_n \quad \text{by } \text{GetPut} \text{ for } l \\
& = s \quad \text{by definition } s_1 \text{ to } s_n
\end{align*}
\]

and obtain the required equivalence.

\textbf{PutGet:} Let \( v \in V^* \) and \( s \in S^* \) be strings. As \( V^t_1 \), there exist unique strings \( v_1 \in V \) to \( v_n \in V \) such that \( v = (v_1 \cdots v_n) \). Similarly, as \( S^t_1 \) there exist unique strings \( s_1 \in S \) to \( s_n \in S \) such that \( s = (s_1 \cdots s_m) \). We calculate as follows

\[
\begin{align*}
l^*.\text{get} (l^*.\text{put} \ v \ s) & = l^*.\text{get} (l^*.\text{put} (v_1 \cdots v_n) (s_1 \cdots s_m)) \quad \text{by definition } v_1 \text{ to } v_n \text{ and } s_1 \text{ to } s_m \\
& = l^*.\text{get} (s'_1 \cdots s'_n) \quad \text{by definition } l^*.\text{put} \\
& \quad \text{where } s'_i = \begin{cases} 
l^*.\text{put} \ v_i \ s_i & \text{for } i \in \{1, \ldots, \max(m, n)\} \\
\text{l.create} \ v_i & \text{for } i \in \{\max(m, n) + 1, \ldots, n\}
\end{cases} \\
& = (l^*.\text{get} (s'_1)) \cdots (l^*.\text{get} (s'_n)) \quad \text{by definition } l^*.\text{get} \text{ and } S^t_1 \\
& \sim_{V^t} v_1 \cdots v_n \quad \text{by } \text{PutGet} \text{ and } \text{CreateGet} \text{ for } l \\
& = v \quad \text{by definition } v_1 \text{ to } v_n
\end{align*}
\]

and obtain the required equivalence.

\textbf{CreateGet:} Similar to the proof for PutGet. \qed
4.3.5 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 all \( v \) 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 \((l_1 | l_2)\) is a quotient lens in \( S_1 \cup S_2/\sim_{S_1} \iff V_1 \cup V_2/\sim_{V_1} \) where \( \sim_{S_1} = (\sim_{S_1} \cup \sim_{S_2}) \) and \( \sim_{V_1} = (\sim_{V_1} \cup \sim_{V_2}) \).

Proof:

\textbf{GetEquiv:} Let \( s \in (S_1 \cup S_2) \) and \( s' \in (S_1 \cup S_2) \) be strings such that \( s \sim_{S_1} s' \). 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.

\textbf{Case} \( s \in S_1 \text{ and } s' \in S_1 \): We calculate as follows

\[
(l_1 | l_2).\text{get } s = l_1.\text{get } s \\
\sim_{V_1} l_1.\text{get } s' \quad \text{by GetEquiv for } l_1 \\
= (l_1 | l_2).\text{get } s'
\]

\textbf{Case} \( s \in S_1 \text{ and } s' \in S_2 \): Symmetric to the previous case.

\textbf{PutEquiv:} Let \( v \in (V_1 \cup V_2) \) and \( v' \in (V_1 \cup V_2) \) be strings such that \( v \sim_{V_1} v' \) and similarly, let \( s \in (S_1 \cup S_2) \) and \( s' \in (S_1 \cup S_2) \) be strings such that \( s \sim_{V_1} s' \). By the conditions on \( \sim_{V_1} \) and \( \sim_{V_2} \) for strings in the intersection \((V_1 \cap V_2)\), we either have:

\begin{itemize}
  \item \( v \in (V_1 \cap V_2) \) and \( v' \in (V_1 \cap V_2) \) with \( v \sim_{V_1} v' \) and \( v \sim_{V_2} v' \)
  \item or \( v \in (V_1 - V_2) \) and \( v' \in (V_1 - V_2) \) with \( v \sim_{V_1} v' \)
  \item or \( v \in (V_2 - V_1) \) and \( v' \in (V_2 - V_1) \) with \( v \sim_{V_2} v' \)
\end{itemize}

Similarly, as \( (S_1 \cap S_2) = \emptyset \), we either have

\begin{itemize}
  \item \( s \in S_1 \text{ and } s' \in S_1 \text{ and } s \sim_{S_1} s' \)
  \item or \( s \in S_2 \text{ and } s' \in S_2 \text{ and } s \sim_{S_2} s' \).
\end{itemize}

We analyze several cases.
Case \( v \in V_1 \) and \( v' \in V_1 \) and \( s \in S_1 \) and \( s' \in S_1 \): We calculate as follows

\[
(\{l_1 \mid l_2\}).\text{put} \ v \ s = l_1.\text{put} \ v \ s \quad \text{by definition} \ (\{l_1 \mid l_2\}).\text{put}
\]

\[
\sim_S l_1.\text{put} \ v' \ s' \quad \text{by PUTEQUIV for} \ l_1
\]

\[
= (\{l_1 \mid l_2\}).\text{put} \ v' \ s' \quad \text{by definition} \ (\{l_1 \mid l_2\}).\text{put}
\]

and obtain the required equivalence.

Case \( v \in V_2 \) and \( v' \in V_2 \) and \( s \in S_2 \) and \( s' \in S_2 \): Symmetric to the previous case.

Case \( v \in (V_1 - V_2) \) and \( v' \in (V_1 - V_2) \) and \( s \in S_2 \) and \( s' \in S_2 \): We calculate as follows

\[
(\{l_1 \mid l_2\}).\text{put} \ v \ s = l_1.\text{create} \ v \ s \quad \text{by definition} \ (\{l_1 \mid l_2\}).\text{put}
\]

\[
\sim_S l_1.\text{create} \ v' \ s' \quad \text{by CREATEEQUIV for} \ l_1
\]

\[
= (\{l_1 \mid l_2\}).\text{put} \ v' \ s' \quad \text{by definition} \ (\{l_1 \mid l_2\}).\text{put}
\]

and obtain the required equivalence.

Case \( v \in (V_2 - V_1) \) and \( v' \in (V_2 - V_1) \) and \( s \in S_1 \) and \( s' \in S_1 \): Symmetric to the previous case.

- **CreateEquiv**: Similar to the proof for PUTEQUIV.

- **GetPut**: Let \( s \in (S_1 \cup S_2) \). We analyze several cases.

  **Case \( s \in S_1 \)**: We calculate as follows

  \[
  (\{l_1 \mid l_2\}).\text{put} \ ((\{l_1 \mid l_2\}).\text{get} \ s) \ s
  \]

  \[
  = (\{l_1 \mid l_2\}).\text{put} \ (l_1 \text{.get} \ s) \ s \quad \text{by definition} \ (\{l_1 \mid l_2\}).\text{get}
  \]

  \[
  = l_1.\text{put} \ (l_1 \text{.get} \ s) \ s \quad \text{by definition} \ (\{l_1 \mid l_2\}).\text{put} \text{ and as cod}(l_1 \text{.get}) = V_1
  \]

  \[
  \sim_S s \quad \text{by GETPUT for} \ l_1
  \]

  and obtain the required equivalence.

  **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

\[
(l_1 \mid l_2).\text{get} \ ((l_1 \mid l_2).\text{put} \ v \ s) \\
= (l_1 \mid l_2).\text{get} \ (l_1.\text{put} \ v \ s) \quad \text{by definition} \ (l_1 \mid l_2).\text{put} \\
= l_1.\text{get} \ (l_1.\text{put} \ v \ s) \quad \text{by definition} \ (l_1 \mid l_2).\text{get} \ \text{and} \ \text{cod}(l_1.\text{put}) = S_1 \\
\sim_V \ v \quad \text{by \textsc{PutGet} for} \ l_1
\]

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

\[
(l_1 \mid l_2).\text{get} \ ((l_1 \mid l_2).\text{put} \ v \ s) \\
= (l_1 \mid l_2).\text{get} \ (l_1.\text{create} \ v) \quad \text{by definition} \ (l_1 \mid l_2).\text{put} \\
= l_1.\text{get} \ (l_1.\text{create} \ v) \quad \text{by definition} \ (l_1 \mid l_2).\text{get} \ \text{and} \ \text{cod}(l_1.\text{create}) = S_1 \\
\sim_V \ v \quad \text{by \textsc{CreateGet} for} \ l_1
\]

and obtain the required equivalence.

Case \( V \in (V_2 - V_1) \) and \( s \in S_1 \): Symmetric to the previous case.

\[\blacksquare\]

**CreateGet**: Similar to the proof for \textsc{PutGet}.

\[
\begin{array}{ccc}
S_1 \cdot S_2 & V_2 \cdot V_1 \\
\sim_S = \sim_{S_1} \cdot \sim_{S_2} & \sim_V = \sim_{V_1} \cdot \sim_{V_2} \\
(l_1 \sim l_2) \in (S_1 \cdot S_2) \sim_S & \iff (V_1 \cdot V_2) \sim_V
\end{array}
\]

**4.3.6 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 \((S_1 \cdot S_2)\) and \((V_2 \cdot V_1)\). Then \((l_1 \sim l_2)\) is a quotient lens in \((S_1 \cdot S_2) \sim_S \iff (V_2 \cdot V_1)/\sim_V\) where \(\sim_S \triangleq (\sim_{S_1} \cdot \sim_{S_2})\) and \(\sim_V \triangleq (\sim_{V_1} \cdot \sim_{V_2})\).

**Proof**: 

\[\blacksquare\]

**GetEquiv**: Let \( s \in (S_1 \cdot S_2) \) and \( s' \in (S_1 \cdot S_2) \) be strings such that \( s \sim_S s' \). As \( S_1 \cdot S_2 \) there exist unique strings \( s_1 \in S_1 \) and \( s_2 \in S_2 \) and \( s'_1 \in S_1 \) and \( s'_2 \in S_2 \) such that \( s = (s_1 \cdot s_2) \) and \( s' = (s'_1 \cdot s'_2) \).
\( s' = (s'_1 \cdot s'_2) \). Also, by the definition of \( \sim_S \) we have that \( s_1 \sim_S s'_1 \) and \( s_2 \sim_S s'_2 \). Using these facts and definitions, we calculate as follows

\[
(l_1 \sim l_2).\text{get } s \\
= (l_2.\text{get } s_2) \cdot (l_1.\text{get } s_1) \quad \text{by definition } (l_1 \sim l_2).\text{get} \\
\sim_V (l_2.\text{get } s'_2) \cdot (l_1.\text{get } s'_1) \quad \text{by GETEQUIV for } l_2 \text{ and } l_1 \\
= (l_1 \sim l_2).\text{get } s' \quad \text{by definition } (l_1 \sim l_2).\text{get}
\]

and obtain the required equivalence.

\textbf{PutEquiv:} Let \( v \in (V_2 \cdot V_1) \) and \( v' \in (V_2 \cdot V_1) \) be strings such that \( v \sim_V v' \) and similarly, let \( s \in (S_1 \cdot S_2) \) and \( s' \in (S_1 \cdot S_2) \) be strings such that \( s \sim_S s' \). As \( V_2 \cdot V_1 \) there exist unique strings \( v_2 \in V_2 \) and \( v_1 \in V_1 \) and \( v'_2 \in V_2 \) and \( v'_1 \in V_1 \) such that \( v = (v_2 \cdot v_1) \) and \( v' = (v'_2 \cdot v'_1) \). Likewise, as \( S_1 \cdot S_2 \) there exist unique strings \( s_1 \in S_1 \) and \( s_2 \in S_2 \) and \( s'_1 \in S_1 \) and \( s'_2 \in S_2 \) such that \( s = (s_1 \cdot s_2) \) and \( s' = (s'_1 \cdot s'_2) \). Moreover, by the definition of \( \sim_V \) we have that \( v_2 \sim_V v'_2 \) and \( v_1 \sim_V v'_1 \) and similarly, that \( s_1 \sim_S s'_1 \) and \( s_2 \sim_S s'_2 \). Using these facts and definitions, we calculate as follows

\[
(l_1 \sim l_2).\text{put } v \ s \\
= (l_1 \sim l_2).\text{put } (v_2 \cdot v_1) \ (s_1 \cdot s_2) \quad \text{by definition } v_2 \text{ and } v_1 \text{ and } s_1 \text{ and } s_2 \\
= (l_1.\text{put } v_1 \ s_1) \cdot (l_2.\text{put } v_2 \ s_2) \quad \text{by definition } (l_1 \sim l_2).\text{put} \\
\sim_S (l_1.\text{put } v'_1 \ s'_1) \cdot (l_2.\text{put } v'_2 \ s'_2) \quad \text{by PUTEQUIV for } l_1 \text{ and } l_2 \\
= (l_1 \sim l_2).\text{put } (v'_2 \cdot v'_1) \ (s'_1 \cdot s'_2) \quad \text{by definition } (l_1 \sim l_2).\text{put} \\
= (l_1 \sim l_2).\text{put } v' \ s' \quad \text{by definition } v_2 \text{ and } v_1 \text{ and } s_1 \text{ and } s_2
\]

and obtain the required equivalence.

\textbf{CreateEquiv:} Similar to the proof for PutEquiv.

\textbf{GetPut:} Let \( s \in (S_1 \cdot S_2) \) be a string. 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 this fact, we calculate as follows

\[
(l_1 \sim l_2).\text{put} \ ((l_1 \sim l_2).\text{get} \ s) \ s
\]

\[
= (l_1 \sim l_2).\text{put} \ ((l_1 \sim l_2).\text{get} \ (s_1 \cdot s_2)) \ (s_1 \cdot s_2) \quad \text{by definition \( s_1 \) and \( s_2 \)}
\]

\[
= (l_1 \sim l_2).\text{put} \ ((l_2.\text{get} \ s_2) \cdot (l_1.\text{get} \ s_1)) \ (s_1 \cdot s_2) \quad \text{by definition \( (l_1 \sim l_2).\text{get} \)}
\]

\[
= (l_1.\text{put} \ (l_1.\text{get} \ s_1)) \cdot (l_2.\text{put} \ (l_2.\text{get} \ s_2) \ s_2) \quad \text{by definition \( (l_1 \sim l_2).\text{put} \) with \( V_2.1 V_1 \)}
\]

\[
\sim_S \ s_1 \cdot s_2 \quad \text{By \( \text{GetPut} \) for \( l_1 \) and \( l_2 \) and definition \( \sim_S \)}
\]

\[
= s \quad \text{by definition \( s_1 \) and \( s_2 \)}
\]

and obtain the required equivalence.

- **PutGet**: Let \( v \in (V_2.1 V_1) \) and \( s \in (S_1.1 S_2) \) be strings. As \( V_2.1 V_1 \) there exist unique strings \( v_2 \in V_2 \) and \( v_1 \in V_1 \) such that \( v = (v_2 \cdot v_1) \). Likewise, as \( S_1.1 S_2 \) 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

\[
(l_1 \sim l_2).\text{get} \ ((l_1 \sim l_2).\text{put} \ v \ s)
\]

\[
= (l_1 \sim l_2).\text{get} \ ((l_1 \sim l_2).\text{put} \ (v_2 \cdot v_1) \ (s_1 \cdot s_2)) \quad \text{by definition \( v_2 \) and \( v_1 \) and \( s_1 \) and \( s_2 \)}
\]

\[
= (l_1 \sim l_2).\text{get} \ ((l_1.\text{put} \ v_1 \ s_1) \cdot (l_2.\text{put} \ v_2 \ s_2)) \quad \text{by definition \( (l_1 \sim l_2).\text{put} \)}
\]

\[
= (l_1.\text{get} \ (l_1.\text{put} \ v_1 \ s_1)) \cdot (l_s.\text{get} \ (l_2.\text{put} \ v_2 \ s_2)) \quad \text{by definition \( (l_1 \sim l_2).\text{get} \) with \( S_1.1 S_2 \)}
\]

\[
\sim_V \ v_1 \cdot v_2 \quad \text{By \( \text{PutGet} \) for \( l_2 \) and \( l_1 \)}
\]

\[
= v \quad \text{by definition \( v_1 \) and \( v_2 \)}
\]

and obtain the required equivalence.

- **CreateGet**: Similar to the proof for **PutGet**. 

---

\[
\begin{align*}
q_1 \in S_1 & \iff U_1/\sim_{U_1} & q_2 \in S_2 & \iff U_2/\sim_{U_2} \\
S_1.1 S_2 & \sim_U = \text{TransClosure}(\sim_{U_1.1 \sim_{U_2}}) \\
split & \in \Pi u : (U_1.1 U_2). \{(u_1, u_2) \in (U_1 \times U_2) \mid (u_1 \cdot u_2) = u\} \\
q_1 . q_2 & \in S_1.1 S_2 \iff U_1.1 U_2/\sim_U
\end{align*}
\]

**4.3.7 Lemma**: Let \( q_1 \in S_1 \iff U_1/\sim_{U_1} \) and \( q_2 \in S_2 \iff U_2/\sim_{U_2} \) be canonizers such that \( S_1.1 S_2 \). Also let \( \text{split} \) be a function in \( \Pi u : (U_1.1 U_2). \{(u_1, u_2) \in (U_1 \times U_2) \mid (u_1 \cdot u_2) = u\} \). Then \( q_1 . q_2 \) is a canonizer in \( S_1.1 S_2 \iff U_1.1 U_2/\sim_U \) where \( \sim_U = \text{TransClosure}(\sim_{U_1.1 \sim_{U_2}}) \).

203
Proof:

**ReCanonize:** Let \( u \in U_1 \cdot U_2 \) with split \( u = (u_1, u_2) \). We calculate as follows

\[
(q_1 \cdot q_2).\text{canonize} \left( (q_1 \cdot q_2).\text{choose} \ u \right) = (q_1 \cdot q_2).\text{canonize} \left( (q_1.\text{choose} \ u_1) \cdot (q_2.\text{choose} \ u_2) \right)
\]

by definition \( u_1 \) and \( u_2 \)

\[
= (q_1.\text{canonize} \ (q_1.\text{choose} \ u_1)) \cdot (q_2.\text{canonize} \ (q_2.\text{choose} \ u_2))
\]

as \( S_1^{-1} S_2 \) and \( \text{cod}(q_1.\text{choose}) = S_1 \) and \( \text{cod}(q_2.\text{choose}) = S_2 \)

\[
\sim_U (u_1 \cdot u_2)
\]

by ReCanonize for \( q_1 \) and \( q_2 \)

\[
= u
\]

and obtain the required equivalence.

\[
\frac{q \in S \xrightarrow{\sim_U} U/\sim_U}{S^* \xrightarrow{\sim_{U}} \text{TransClosure}(\sim_{U})}
\]

\[
\text{split} \in \Pi u : \ast. \{[u_1, \ldots, u_n] \in U \text{ list} \mid (u_1 \cdot \cdots u_n) = u\}
\]

\[
q^* \in S^* \xrightarrow{\sim_{U^*}} U^*/\sim_U
\]

**4.3.9 Lemma:** Let \( q \in S \xrightarrow{\sim_U} U/\sim_U \) be a canonizer such that \( S^\ast \). Also let \( \text{split} \) be a function in \( \Pi u : \ast. \{[u_1, \ldots, u_n] \in U \text{ list} \mid (u_1 \cdot \cdots u_n) = u\} \). Then \( q^* \) is a canonizer in \( S^* \xrightarrow{\sim_{U^*}} U^*/\sim_U \) where \( \sim_{U^*} = \text{TransClosure}(\sim_{U^*}) \).

Proof:

**ReCanonize:** Let \( u \in U^\ast \) be a string and let \( \text{split} \ u = [u_1, \ldots, u_n] \) a list of strings. We calculate as follows

\[
q^* . \text{canonize} \ (q^* . \text{choose} \ u)
\]

\[
= q^* . \text{canonize} \ ((q.\text{choose} \ u_1) \cdot \cdots (q.\text{choose} \ b_n))
\]

by definition \( q^* . \text{choose} \)

\[
= (q.\text{canonize} \ (q.\text{choose} \ u_1)) \cdot \cdots
\]

by definition \( q^* . \text{canonize} \)

\[
(q.\text{canonize} \ (q.\text{choose} \ u_n))
\]

and \( S^* \) and \( \text{cod}(q.\text{choose}) = S \)

\[
\sim_U u_1 \cdot \cdots u_n
\]

by ReCanonize for \( q \) (\( n \) times)

\[
= u
\]

and obtain the required equivalence.
4.3.10 Lemma: Let \( q_1 \in S_1 \mapsto U_1/\sim_U \) and \( q_2 \in S_2 \mapsto U_2/\sim_U \) be canonizers such that \( (S_1 \cap S_2) = \emptyset \). Then \( (q_1 \, | \, q_2) \) is a canonizer in \( (U_1 \cup U_2)/\sim_U \) where \( \sim_U = \text{TransClosure}(\sim_U \cup \sim_U) \).

Proof:

\begin{itemize}
  \item \textbf{ReCanonize:} Let \( u \in (U_1 \cup U_2) \). We analyze two cases.

  \textbf{Case } u \in U_1: \text{ We calculate as follows}
  \[
  (q_1 \, | \, q_2).\text{canonize} \ (\langle q_1 \, | \, q_2 \rangle.\text{choose } u)
  = (q_1 \, | \, q_2).\text{canonize} \ (q_1.\text{choose } u)
  \quad \text{by definition } (q_1 \, | \, q_2).\text{choose}
  = q_1.\text{canonize} \ (q_1.\text{choose } u)
  \quad \text{by definition } (q_1 \, | \, q_2).\text{canonize}
  \quad \text{and } \text{cod}(q_1.\text{choose}) = S_1
  \quad \text{by ReCanonize for } q_1
  \quad \sim_U u
  \]
  and obtain the required equivalence.

  \textbf{Case } u \in U_1: \text{ Symmetric to the previous case.} \quad \square
\end{itemize}

4.4.1 Lemma: Let \( l \in S/\sim_S \iff V_1/\sim_{V_1} \) be a basic lens and \( f \in S \to V_2 \) a function such that \( V_1.\,!V_2 \sim_V = \sim_{V_1.\,\text{Tot}(V_2)} \).

\[
\text{dup}_1 \, l \, f \in S/\sim_S \iff (V_1.\,!V_2)/\sim_V
\]

\begin{itemize}
  \item \textbf{Proof:}
\end{itemize}
GetEquiv: Let $s \in S$ and $s' \in S$ be strings such that $s \sim_S s'$. We calculate as follows

$$(\text{dup}_1 \text{l } \text{f} \text{).get } s)$$

$$= (\text{l.get } s) \cdot (\text{f } s) \quad \text{by definition (}\text{dup}_1 \text{l } \text{f}.\text{get})$$

$\sim_V (\text{l.get } s') \cdot (\text{f } s') \quad \text{by GETEQUIV for } \text{l} \text{ and definition of } \sim_V$

$$= (\text{dup}_1 \text{l } \text{f} \text{).get } s'$$

and obtain the required equivalence.

PutEquiv: Let $v \in (V_1 \cdot V_2)$ and $v' \in (V_1 \cdot V_2)$ be strings such that $v \sim_V v'$ and similarly let $s \in S$ and $s' \in S$ be strings such that $s \sim_S s'$. As $V_1 \cdot V_2$ there exist unique strings $v_1 \in V_1$ and $v_2 \in V_2$ such that $v = (v_1 \cdot v_2)$ and $v_1' \in V_1$ and $v_2' \in V_2$ such that $v' = (v_1' \cdot v_2')$. Moreover, by the definition of $\sim_V$ we have $v_1 \sim_{V_1} v_1'$. Using these facts and definitions, we calculate as follows

$$(\text{dup}_1 \text{l } \text{f} \text{.put } v \text{ s})$$

$$= (\text{dup}_1 \text{l } \text{f} \text{.put } (v_1 \cdot v_2) \text{ s}) \quad \text{by definition } v_1 \text{ and } v_2$$

$$= \text{l.put } v_1 \text{ s} \quad \text{by definition (}\text{dup}_1 \text{l } \text{f} \text{.put})$$

$\sim_S \text{l.put } v_1' \text{ s'} \quad \text{by PUTEQUIV for } \text{l}$

$$= (\text{dup}_1 \text{l } \text{f} \text{.put } (v_1' \cdot v_2') \text{ s'}) \quad \text{by definition (}\text{dup}_1 \text{l } \text{f} \text{.put})$$

$$= (\text{dup}_1 \text{l } \text{f} \text{.put } v' \text{ s'}) \quad \text{by definition } v_1' \text{ and } v_2'$$

and obtain the required equivalence.

CreateEquiv: Similar to the proof for PUTEQUIV.

GetPut: Let $s \in S$ be a string. We calculate as follows

$$(\text{dup}_1 \text{l } \text{f}.\text{put } ((\text{dup}_1 \text{l } \text{f}.\text{get } s) \text{ s})$$

$$= (\text{dup}_1 \text{l } \text{f}.\text{put } ((\text{l.get } s) \cdot (\text{f } s)) \text{ s}) \quad \text{by definition (}\text{dup}_1 \text{l } \text{f}.\text{get})$$

$$= \text{l.put } (\text{l.get } s) \quad \text{by definition (}\text{dup}_1 \text{l } \text{f}.\text{put } \text{ and } V_1 \cdot V_2$$

and $\text{cod}(\text{l.get}) = V_1$ and $\text{cod}(\text{f}) = V_2$

$\sim_S s \quad \text{by GETPUT for } \text{l}$

and obtain the required equivalence.

PutGet: Let $v \in V$ and $s \in S$ be strings. As $V_1 \cdot V_2$, there exist unique strings $v_1 \in V_1$ and
\[ v_2 \in V_2 \text{ such that } v = (v_1 \cdot v_2). \] Using these facts, we calculate as follows

\[
(dup_1 l f).get ((dup_1 l f).put v s) \\
= (dup_1 l f).get ((dup_1 l f).put (v_1 \cdot v_2) s) \quad \text{by definition } v_1 \text{ and } v_2 \\
= (dup_1 l f).get (l.put v_1 s) \quad \text{by definition } (dup_1 l f).put \\
= (l.get (l.put v_1 s)) \cdot (f (l.put v_1 s)) \quad \text{by definition } (dup_1 l f).get \\
\sim_V v \quad \text{by } \text{PUTGET for } l \text{ and definition of } \sim_V
\]

and obtain the required equivalence.

\textbf{CreateGet:} Similar to the proof for \text{PUTGET}. \hfill \Box

\begin{align*}
&f \in S \Rightarrow S_0 \quad S_0 \subseteq S \quad \forall s \in S_0. \ f \ s = s \\
&\text{normalize } f \in S \mapsto S_0/=
\end{align*}

\textbf{4.4.2 Lemma:} Let \( S \) and \( S_0 \) be sets such that \( S_0 \subseteq S \). Also let \( f \in S \rightarrow S_0 \) be a function from \( S \) to \( S_0 \). Then \( (\text{normalize } f) \) is a canonizer in \( S \leftrightarrow S_0/\).  

Proof:

\textbf{ReCanonize:} Let \( s \in S_0 \). We calculate as follows

\[
(normalize f).\text{canonize } ((normalize f).\choose s) \\
= (normalize f).\text{canonize } s \quad \text{by definition } (normalize f).\choose \\
= f \ s \quad \text{by definition } (normalize f).\text{canonize} \\
= s \quad \text{as } s \in S_0
\]

and obtain the required equality. \hfill \Box

\begin{align*}
&n \in \mathbb{N} \quad sp \in \Sigma^* \quad nl \in \Sigma^* \\
&(\Sigma^* \cdot nl \cdot \Sigma^*) \cap S_0 = \emptyset \\
&S = [(sp \cup nl)/sp]_{S_0} \\
&\text{columnize } n \ S_0 \ sp \ nl \in S \mapsto S_0/=
\end{align*}

\textbf{4.4.3 Lemma:} Let \( n \) be a number, \( S_0 \subseteq \Sigma^* \) a language, and \( sp \in \Sigma^* \) and \( nl \in \Sigma^* \) strings such that \( nl \) does not occur in any string in \( S_0 \). Then \( (\text{columnize } n \ S_0 \ sp \ nl) \) is a canonizer in \( S \leftrightarrow S_0/\) where \( S = [(sp \cup nl)/sp]_{S_0} \).
Proof:

▷ **ReCanonize:** Let $s$ be a string in $S_0$. The required equality,

$$(\text{columnize } n \ S_0 \ sp \ nl) \ . \ \text{canonize} \ (\text{columnize } n \ S_0 \ sp \ nl) \ . \ \text{choose} \ s = s$$

is immediate as $nl$ does not occur in $s$, $(\text{columnize } n \ S_0 \ sp \ nl) \ . \ \text{choose}$ replaces some occurrences of $s$ with $nl$, and $(\text{columnize } n \ S_0 \ sp \ nl) \ . \ \text{canonize}$ replaces every occurrence of $nl$ with $s$. □

### Resourceful Lenses Proofs

#### 5.2.2 Lemma [PutChunks]:
For every resourceful lens $l \in S \xleftrightarrow{G\ E\ M} \ V$, string $v \in [V]$, skeleton $c \in C$, and resource $r \in \{\| \mathbb{N} \mapsto k. S\}$ we have $\text{locs}(l. \text{put} \ v \ (c, r)) = \text{locs}(v)$.

**Proof:** Let $l \in S \xleftrightarrow{G\ E\ M} \ V$ be a resourceful lens, $v \in [V]$ a string, $c \in C$ a skeleton, and $r \in \{\| \mathbb{N} \mapsto k. S\}$ a resource. We calculate as follows

\[
\begin{align*}
\text{locs}(l. \text{put} \ v \ (c, r)) &= \text{locs}(l. \text{get} \ (l. \text{put} \ v \ (c, r))) \quad \text{by GET CHUNKS for } l \\
&= \text{locs}(v) \quad \text{by PUT GET for } l
\end{align*}
\]

and obtain the required equality. □

#### 5.2.3 Lemma [CreateChunks]:
For every resourceful lens $l \in S \xleftrightarrow{G\ E\ M} \ V$, string $v \in [V]$, and resource $r \in \{\| \mathbb{N} \mapsto k. S\}$ we have $\text{locs}(l. \text{create} \ v \ r) = \text{locs}(v)$.

**Proof:** Similar to the proof of Lemma 5.2.2. □

#### 5.2.4 Lemma [ReorderPut]:
For every resourceful lens $l \in S \xleftrightarrow{G\ E\ M} \ V$, string $v \in [V]$, rigid complement $c \in C$, resource $r \in \{\| \mathbb{N} \mapsto k. S\}$, and permutation $q \in \text{Perms}(v)$ we have: $\oplus (l. \text{put} \ v \ (c, r)) = l. \text{put} \ (\oplus \ v) \ (c, r \circ q^{-1})$.

**Proof:** Let $l \in S \xleftrightarrow{G\ E\ M} \ V$ be a resourceful lens, $v \in [V]$ a string, $c \in C$ a rigid complement, $r \in \{\| \mathbb{N} \mapsto k. S\}$ a resource, and $q \in \text{Perms}(v)$ a permutation on the chunks in $v$ such that $\text{dom}(r) = (\text{locs}(v))$. To shorten the proof, define strings $s_1$ and $s_2$ as follows:

\[
\begin{align*}
\quad s_1 &\triangleq l. \text{put} \ (\oplus \ v) \ (c, r \circ q^{-1}) \\
\quad s_2 &\triangleq \oplus (l. \text{put} \ v \ (c, r))
\end{align*}
\]
First, we demonstrate that the sets of locations in $s_1$ and $s_2$ are identical, by calculating as follows:

$$\text{locs}(s_1) = \text{locs}(l.\text{put} (\oplus \ v) (c, r \circ q^{-1}))$$  
by definition $s_1$

$$= \text{locs}(\oplus \ v)$$  
by Lemma 5.2.2 for $l$

$$= \text{locs}(v)$$  
by definition $\text{locs}$ and $\oplus$

$$= \text{locs}(l.\text{put} \ v (c, r))$$  
by Lemma 5.2.2 for $l$

$$= \text{locs}(\oplus (l.\text{put} \ v (c, r)))$$  
by definition $\text{locs}$ and $\oplus$

$$= \text{locs}(s_2)$$  
by definition of $s_2$

Next, we show that for every location $x \in \text{locs}(s_1)$ the chunk at $x$ in $s_1$ is identical to the chunk at $x$ in $s_2$. Let $x \in \text{locs}(s_1)$ be a location. We analyze two cases.

**Case $q^{-1}(x) \in \text{dom}(r)$:** We calculate as follows

$$s_1[x] = l.\text{put} (\oplus \ v) (c, r \circ q^{-1})[x]$$  
by definition $s$

$$= k.\text{put} ((\oplus \ v)[x]) ((r \circ q^{-1})(x))$$  
by CHUNPuta l for $l$

$$= k.\text{put} (v[q^{-1}(x)]) (r(q^{-1}(x)))$$  
by definition $\oplus$ and $[]$

$$= (l.\text{put} \ v (c, r))[q^{-1}(x)]$$  
by CHUNPuta l for $l$

$$= (\oplus (l.\text{put} \ v (c, r)))[x]$$  
by definition $\oplus$ and $[]$

$$= s_2[x]$$  
by definition $s_2$

and obtain the required equality.

**Case $q^{-1}(x) \notin \text{dom}(r)$:** Similar to the previous case.

Finally, we prove that $\text{skeleton}(s_1) = \text{skeleton}(s_2)$. Observe that $\text{skeleton}(v) = \text{skeleton}(\oplus \ v)$. Using this fact, we calculate as follows:

$$\text{skeleton}(s_1) = \text{skeleton}(k.\text{put} (\oplus \ v) (c, r \circ q^{-1}))$$  
by definition $s_1$

$$= \text{skeleton}(l.\text{put} \ v (c, r))$$  
by SKELPuta l for $l$

$$= \text{skeleton}(\oplus (l.\text{put} \ v (c, r)))$$  
by definition $\text{skeleton}$ and $\oplus$

$$= \text{skeleton}(s_2)$$  
by definition $s_2$

Putting all these facts together we have $s_1 = s_2$, which completes the proof. □

**A.1.29 Lemma [ReorderCreate]:** For every resourceful lens $l \in S \overset{C.t.\emptyset}{\leftrightarrow} V$, string $v \in [V]$, rigid complement $c \in C$, resource $r \in \{\emptyset \rightarrow k.S\}$, and permutation $q \in \text{Perms}(v)$ we have:

$$\oplus (l.\text{create} \ v \ r) = l.\text{create} (\oplus \ v) (r \circ q^{-1})$$.
Proof: Similar to the proof of Lemma 5.2.4.

\[ l \in S \overset{C_t \bowtie}{\implies} V \]

\[ [l] \in [S] \iff [V] \]

5.2.5 Lemma: Let \( l \in S \overset{C_t \bowtie}{\implies} V \) be a resourceful lens. Then \([l]\) is a basic lens in \([S] \iff [V]\).

Proof:

▶ GetPut: Let \( s \in [S] \). We calculate as follows

\[
[l].\text{put} ([l].\text{get} s) ([l].\text{res} s)
\]

\[
= [l].\text{put} (l.\text{get} s) s \quad \text{by definition } [l].\text{get} \text{ and } [l].\text{res}
\]

\[
= l.\text{put} (l.\text{get} s) (c, r \circ g) \quad \text{by definition } [l].\text{put}
\]

where \( c, r = l.\text{res} s \)

and \( g = \text{align}(l.\text{get} s, l.\text{get} s) \)

\[
= l.\text{put} (l.\text{get} s) (c, r) \quad \text{By ALIGN, GETCHUNKS, and RESCHUNKS}
\]

\[
= l.\text{put} (l.\text{get} s) (l.\text{res} s) \quad \text{by definition } (c, r)
\]

\[
= s \quad \text{by GetPut for } l
\]

and obtain the required equality.

▶ PutGet: Let \( v \in [V] \) and \( s \in [S] \). We calculate as follows

\[
[l].\text{get} ([l].\text{put} v s)
\]

\[
= [l].\text{get} (l.\text{put} v (c, r \circ g)) \quad \text{by definition } [l].\text{put}
\]

where \( c, r = l.\text{res} s \)

and \( g = \text{align}(v, l.\text{get} s) \)

\[
= l.\text{get} (l.\text{put} v (c, r \circ g)) \quad \text{by definition } [l].\text{get}
\]

\[
= v \quad \text{by PutGet for } l
\]

and obtain the required equality.

▶ CreateGet: Let \( v \in [V] \). We calculate as follows

\[
[l].\text{get} ([l].\text{create} v)
\]

\[
= [l].\text{get} (l.\text{create} v []) \quad \text{by definition } [l].\text{create}
\]

\[
= l.\text{get} (l.\text{create} v []) \quad \text{by definition } [l].\text{put}
\]

\[
= v \quad \text{by CreateGet for } l
\]
and obtain the required equality, which completes the proof. \[\]

\[
\begin{align*}
  k \in S' & \iff V' \quad l \in S \iff V \\
  \hat{l} \in S & \iff \hat{V}
\end{align*}
\]

5.3.1 Lemma: Let \( l \in S \iff V \) and \( k \in S' \iff V' \) be basic lenses. Then \( \hat{l} \) is a resourceful lens in \( S \iff V \).

Proof:

\[\]

- **GetPut**: Let \( s \in \lfloor S \rfloor \) be a string. As \( S \) is an ordinary regular expression, we have that \( s \in S \). Using this fact, we calculate as follows

\[
\begin{align*}
  \hat{l}.put (\hat{l}.get s) (\hat{l}.res s) \\
  = \hat{l}.put (l.get s) (l.res s, \emptyset) & \text{ by definition of } \hat{l}.get \text{ and } \hat{l}.res \\
  = l.put (l.get s) (l.res s) & \text{ by definition of } \hat{l}.put \\
  = s & \text{ by } \text{GetPut} \text{ for } l
\end{align*}
\]

and obtain the required equality.

- **PutGet**: Let \( v \in \lfloor V \rfloor \) be a string, \( c \in C \) a rigid complement, \( r \in \{\left\{ N \to k.S \right\} \} \) a resource. As \( V \) is an ordinary regular expression, we have that \( v \in V \). Using this fact, we calculate as follows

\[
\begin{align*}
  \hat{l}.get (\hat{l}.put v (c, r)) \\
  = \hat{l}.get (l.put v c) & \text{ by definition } \hat{l}.put \\
  = l.get (l.put v c) & \text{ by definition } \hat{l}.get \\
  = v & \text{ by } \text{PutGet} \text{ for } l
\end{align*}
\]

and obtain the required equality.

- **CreateGet**: Let \( v \in \lfloor V \rfloor \) be a string and \( r \in \{\left\{ N \to k.S \right\} \} \) a resource. As \( V \) is an ordinary regular expression, we have that \( v \in V \). Using this fact, we calculate as follows

\[
\begin{align*}
  \hat{l}.get (\hat{l}.create v r) \\
  = \hat{l}.get (l.create v) & \text{ by definition } \hat{l}.create \\
  = l.get (l.create v) & \text{ by definition } \hat{l}.get \\
  = v & \text{ by } \text{CreateGet} \text{ for } l
\end{align*}
\]

211
and obtain the required equality.

**GetChunks:** Let $s \in \lfloor S \rfloor$. We calculate as follows

$$\text{locs}(s) = \emptyset \quad \text{as } S \text{ is an ordinary regular expression}$$

$$= \text{locs}(\hat{l}.\text{get } s) \quad \text{as } V \text{ is an ordinary regular expression}$$

and obtain the required equality.

**ResChunks:** Let $s \in \lfloor S \rfloor$ be a string, $c \in C$ a rigid complement, and $r \in \{[N \mapsto k.S]\}$ a resource such that $(c, r) = \hat{l}.\text{res } s$. By the definition of $\hat{l}.\text{res}$ we have that $r = \{[]\}$. Using this fact, we calculate as follows

$$\text{locs}(s) = \emptyset \quad \text{as } S \text{ is an ordinary regular expression}$$

$$= \text{dom}(r) \quad \text{as } r = \{[]\}$$

and obtain the required equality.

**ChunkPut:** Vacuously holds. Suppose, for a contradiction, that there exists a string $v \in \lfloor V \rfloor$, a resource $r \in \{[N \mapsto k.S]\}$, and a location $x \in (\text{locs}(v) \cap \text{dom}(r))$. As $V$ is an ordinary regular expression, we have that $\text{locs}(v) = \emptyset$, which contradicts $x \in \text{locs}(v)$.

**ChunkCreate:** Vacuously holds by the same argument as the proof for ChunkPut.

**NoChunkPut:** Vacuously holds by the same argument as the proof for ChunkPut.

**NoChunkCreate:** Vacuously holds by the same argument as the proof for ChunkPut.

**SkelPut:** Let $v \in \lfloor V \rfloor$ and $v' \in \lfloor V \rfloor$ be strings, $c \in C$ a rigid complement, and $r \in \{[N \mapsto k.S]\}$ and $r' \in \{[N \mapsto k.S]\}$ resources such that $\text{skel}(v) = \text{skel}(v')$. As $V$ is an ordinary regular expression, we have that $v = v'$. Using this fact, we calculate as follows:

$$\text{skel}(\hat{l}.\text{put } v (c, r))$$

$$= \text{skel}(\hat{l}.\text{put } v c) \quad \text{by definition } \hat{l}.\text{put}$$

$$= \text{skel}(\hat{l}.\text{put } v' c) \quad \text{as } v = v'$$

$$= \text{skel}(\hat{l}.\text{put } v' (c, r')) \quad \text{by definition } \hat{l}.\text{put}$$

and obtain the required equality.

**SkelCreate:** Similar to the proof for SkelPut.
5.3.2 Lemma: Let \( t \in T \) be a tag and \( k \in S \) be a basic lens. Then \( \langle t : k \rangle \) is a resourceful lens in \( \langle S \rangle \overset{t}{\overset{\vdash}{\overset{k}{\equiv}}} \langle V \rangle \).

Proof:

\( \blacktriangleright \text{GetPut:} \) Let \( s \in \lfloor \langle S \rangle \rfloor \). By the semantics of chunk-annotated regular expressions we also have \( s \in S \). Using this fact, we calculate as follows

\[
\langle t : k \rangle \text{.} \text{put} ((\langle t : k \rangle \text{.} \text{get} \ s) \text{)} \text{.} \text{res} \ s)
\]

\[
= \langle t : k \rangle \text{.} \text{put} (k \text{.} \text{get} \ s) (\{1 \mapsto s\}) \quad \text{by definition } \langle t : k \rangle \text{.} \text{get} \text{ and } \langle t : k \rangle \text{.} \text{res}
\]

\[
= k \text{.} \text{put} (k \text{.} \text{get} \ s) (\{1 \mapsto s\}(1)) \quad \text{by definition } \langle t : k \rangle \text{.} \text{put} \text{ with } 1 \in \text{dom}(\{1 \mapsto s\})
\]

\[
= k \text{.} \text{put} (k \text{.} \text{get} \ s) \ s \quad \text{by definition application}
\]

\[
= s \quad \text{by GetPut for } k
\]

and obtain the required equality.

\( \blacktriangleright \text{PutGet:} \) Let \( v \in \lfloor \langle V \rangle \rfloor \) be a string, \( \square \in \{\square\} \) a rigid complement, and \( r \in \{\{N \mapsto k.S\}\} \) a resource. By the semantics of chunk-annotated regular expressions we also have \( v \in V \). Using this fact, we calculate as follows

\[
\langle t : k \rangle \text{.} \text{get} ((\langle t : k \rangle \text{.} \text{put} \ v \ (\square, r))
\]

\[
= \begin{cases}
  k \text{.} \text{get} (k \text{.} \text{put} \ v \ r(1)) & \text{if } 1 \in \text{dom}(r) \\
  k \text{.} \text{get} (k \text{.} \text{create} \ v) & \text{otherwise}
\end{cases}
\]

\[
= v \quad \text{by the definition of } \langle t : k \rangle \text{.} \text{get} \text{ and } \langle t : k \rangle \text{.} \text{put}
\]

by PutGet or CreateGet for \( l \)

and obtain the required equality.

\( \blacktriangleright \text{CreateGet:} \) Similar to the proof for PutGet.

\( \blacktriangleright \text{GetChunks:} \) Let \( s \in \lfloor \langle S \rangle \rfloor \). We calculate as follows

\[
\text{locs}(s) = \{1\} \quad \text{by definition } \text{locs}
\]

\[
= \text{locs}(\langle t : k \rangle \text{.} \text{get} \ s) \quad \text{by definition } \text{locs}
\]

and obtain the required equality.
ResChunks: Let $s \in \langle S \rangle$ be a string, $\Box \in \{\Box\}$ a rigid complement, and $r \in \{\mathbb{N} \mapsto k.S\}$ a resource such that $(\Box, r) = \langle t : k \rangle \cdot res \ s$. We calculate as follows

$$\text{dom}(r) = \text{dom}(\{1 \mapsto s\}) \quad \text{by definition } \langle t : k \rangle \cdot res \text{ with } r = \langle t : k \rangle \cdot res \ s$$

$$= \{1\} \quad \text{by definition dom}$$

$$= \text{locs}(s) \quad \text{by definition locs}$$

and obtain the required equality.

ChunkPut: Let $v \in \langle V \rangle$ be a string, $\Box \in \{\Box\}$ a rigid complement, $r \in \{\mathbb{N} \mapsto k.S\}$ a resource, and $x \in (\text{locs}(v) \cap \text{dom}(r))$ a location. As $\text{locs}(v) = \{1\}$ we must have that $x = 1$ and $1 \in \text{dom}(r)$. Using these facts and definitions, we calculate as follows

$$\left(\langle t : k \rangle \cdot \text{put} \ v \ (\Box, r)\right)[x]$$

$$= \langle t : k \rangle \cdot \text{put} \ v \ (\Box, r) \quad \text{by definition } [\cdot] \text{ and } x = 1$$

$$= k \cdot \text{put} \ v \ (r(1)) \quad \text{by definition } \langle t : k \rangle \cdot \text{put} \text{ and as } 1 \in \text{dom}(r)$$

$$= k \cdot \text{put} \ (v[x]) \ (r(x)) \quad \text{by definition } [\cdot] \text{ and } x = 1$$

and obtain the required equality.

ChunkCreate: Similar to the proof of ChunkPut.

NoChunkPut: Let $v \in \langle V \rangle$ be a string, $\Box \in \{\Box\}$ a rigid complement, $r \in \{\mathbb{N} \mapsto k.S\}$ a resource, and $x \in (\text{locs}(v) - \text{dom}(r))$ a location. As $\text{locs}(v) = \{1\}$ we must have that $x = 1$ and $1 \not\in \text{dom}(r)$. Using these facts and definitions, we calculate as follows

$$\left(\langle t : k \rangle \cdot \text{put} \ v \ (\Box, r)\right)[x]$$

$$= \langle t : k \rangle \cdot \text{put} \ v \ (\Box, r) \quad \text{by definition } [\cdot] \text{ and } x = 1$$

$$= k \cdot \text{create} \ v \quad \text{by definition } \langle t : k \rangle \cdot \text{put} \text{ and as } 1 \not\in \text{dom}(r)$$

$$= k \cdot \text{create} \ (v[x]) \quad \text{by definition } [\cdot] \text{ and } x = 1$$

and obtain the required equality.

NoChunkCreate: Similar to the proof for NoChunkPut.

SkelPut: Let $v \in \langle V \rangle$ and $v' \in \langle V \rangle$ be strings, $\Box \in \{\Box\}$ a rigid complement, and $r \in \{\mathbb{N} \mapsto k.S\}$ and $r' \in \{\mathbb{N} \mapsto k.S\}$ resources such that $\text{skel}(v) = \text{skel}(v')$. We calculate as follows

$$\text{skel}(\langle t : k \rangle \cdot \text{put} \ v \ (\Box, r)) = \Box \quad \text{by definition } \text{skel}$$

$$= \text{skel}(\langle t : k \rangle \cdot \text{put} \ v' \ (\Box, r')) \quad \text{by definition } \text{skel}$$
and obtain the required equality.

\[ \text{Lemma:} \quad \text{Let } l_1 \in S_1 \xrightarrow{C_1 \cdot t \cdot k} V_1 \quad \text{and } l_2 \in S_2 \xrightarrow{C_2 \cdot t \cdot k} V_2 \Rightarrow (l_1 \cdot l_2) \in (S_1 \cdot S_2) \xrightarrow{(C_1 \times C_2) \cdot t \cdot k} (V_1 \cdot V_2). \]

\[ \begin{array}{|c|c|}
\hline
| S_1 | \rightarrow | S_2 | & | V_1 | \rightarrow | V_2 | \\
\hline
l_1 \in S_1 \xrightarrow{C_1 \cdot t \cdot k} V_1 & l_2 \in S_2 \xrightarrow{C_2 \cdot t \cdot k} V_2 \\
\hline
(l_1 \cdot l_2) \in (S_1 \cdot S_2) \xrightarrow{(C_1 \times C_2) \cdot t \cdot k} (V_1 \cdot V_2) \\
\hline
\end{array} \]

\[ \text{5.3.3 Lemma:} \quad \text{Let } l_1 \in S_1 \xrightarrow{C_1 \cdot t \cdot k} V_1 \text{ and } l_2 \in S_2 \xrightarrow{C_2 \cdot t \cdot k} V_2 \text{ be resourceful lenses such that } [S_1] \cdot [S_2] \text{ and } [V_1] \cdot [V_2]. \text{ Then } (l_1 \cdot l_2) \text{ is a resourceful lens in } (S_1 \cdot S_2) \xrightarrow{(C_1 \times C_2) \cdot t \cdot k} (V_1 \cdot V_2). \]

**Proof:**

\[ \quad \text{\textbf{GetPut:}} \quad \text{Let } s \in [S_1 \cdot S_2]. \text{ As } [S_1] \cdot [S_2] \text{ there exist unique strings } s_1 \in [S_1] \text{ and } s_2 \in [S_2] \text{ such that } s = (s_1 \cdot s_2). \text{ Using this fact, we calculate as follows} \]

\[ \quad (l_1 \cdot l_2).\text{put} ((l_1 \cdot l_2).\text{get} s) ((l_1 \cdot l_2).\text{res} s) \]
\[ = (l_1 \cdot l_2).\text{put} ((l_1 \cdot l_2).\text{get} (s_1 \cdot s_2)) ((l_1 \cdot l_2).\text{res} (s_1 \cdot s_2)) \quad \text{by definition } s_1 \text{ and } s_2 \]
\[ = (l_1 \cdot l_2).\text{put} ((l_1.\text{get} s_1) \cdot (l_2.\text{get} s_2) \cdot (c_1, c_2), r_1 ++ r_2) \quad \text{by definition } (l_1 \cdot l_2).\text{get} \]
\[ \quad \text{where } c_1, r_1 = l_1.\text{res} s_1 \]
\[ \quad \text{and } c_2, r_2 = l_2.\text{res} s_2 \]
\[ = (l_1.\text{put} (l_1.\text{get} s_1) (c_1, r'_1)) \cdot \text{by definition } (l_1 \cdot l_2).\text{put} \text{ with } [V_1] \cdot [V_2] \]
\[ \quad (l_2.\text{put} (l_2.\text{get} s_2) (c_2, r'_2)) \quad \text{and cod}(l_1.\text{get}) = [V_1] \]
\[ \quad \text{where } r'_1, r'_2 = \text{split}(l.\text{get} s_1, r_1 ++ r_2) \quad \text{and cod}(l_2.\text{get}) = [V_2] \]
\[ = (l_1.\text{put} (l_1.\text{get} s_1) (c_1, r_1)) \quad \text{by GetChunks and ResChunks for } l_1 \]
\[ \quad (l_2.\text{put} (l_2.\text{get} s_2) (c_2, r_2)) \quad \text{and definition } \text{split} \]
\[ = (s_1 \cdot s_2) \quad \text{by GetPut for } l_1 \text{ and } l_2 \]
\[ = s \quad \text{by definition } s_1 \text{ and } s_2 \]

and obtain the required equality.

\[ \quad \text{\textbf{PutGet:}} \quad \text{Let } v \in [V_1 \cdot V_2] \text{ and } (c_1, c_2) \in (C_1 \times C_2) \text{ and } r \in \{\|N \mapsto k.\theta''\} \cdot [V_1] \cdot [V_2]. \text{ As } [V_1] \cdot [V_2] \text{ there exist unique strings } v_1 \in [V_1] \text{ and } v_2 \in [V_2] \text{ such that } v = v_1 \cdot v_2. \text{ Using this fact, we calculate} \]

\[ \quad V_1 \cdot V_2 \]
as follows

\[(l_1 \cdot l_2).get ((l_1 \cdot l_2).put v ((c_1, c_2), r))\]
\[= (l_1 \cdot l_2).get ((l_1 \cdot l_2).put (v_1 \cdot v_2) ((c_1, c_2), r))\]  \hspace{1cm} \text{by definition } v_1 \text{ and } v_2
\[= (l_1 \cdot l_2).get ((l_1.put v_1 (c_1, r_1)) \cdot (l_2.put v_2 (c_2, r_2)))\]  \hspace{1cm} \text{by definition } (l_1 \cdot l_2).put
\[\quad \text{where } r_1, r_2 = \text{split}(|v_1|, r)\]
\[= (l_1 \cdot l_2).get (l_1.put v_1 (c_1, r_1)) \cdot (l_2.put v_2 (c_2, r_2))\]  \hspace{1cm} \text{by definition } (l_1 \cdot l_2).get \text{ with } [S_1] : \downarrow [S_2]
\[= (v_1 \cdot v_2)\]  \hspace{1cm} \text{by } \text{PUTGet for } l_1 \text{ and } l_2
\[= v\]  \hspace{1cm} \text{by definition } v_1 \text{ and } v_2

and obtain the required equality.

- **CreateGet**: Similar to the proof for PUTGet.

- **GetChunks**: Let \( s \in [S_1 \cdot S_2] \). As \([S_1] : \downarrow [S_2]\) there exist unique strings \( s_1 \in [S_1] \) and \( s_2 \in [S_2] \) such that \( s = s_1 \cdot s_2 \). Using this fact, we calculate as follows

\[
\begin{align*}
\text{locs}(s) &= \text{locs}(s_1 \cdot s_2) \quad \text{by definition } s_1 \text{ and } s_2 \\
&= \{1, \ldots, (|s_1| + |s_2|)\} \quad \text{by definition locs} \\
&= \{1, \ldots, (|l_1.get s_1| + |l_2.get s_2|)\} \quad \text{by GETChunks for } l_1 \text{ and } l_2 \\
&= \text{locs}((l_1 \cdot l_2).get s_1) \cdot (l_2.get s_2)) \quad \text{by definition locs} \\
&= \text{locs}((l_1 \cdot l_2).get (s_1 \cdot s_2)) \quad \text{by definition } (l_1 \cdot l_2).get \\
&= \text{locs}((l_1 \cdot l_2).get s) \quad \text{by definition } s_1 \text{ and } s_2
\end{align*}
\]

and obtain the required equality.

- **ResChunks**: Let \( s \in [S_1 \cdot S_2] \) be a string, \((c_1, c_2) \in (C_1 \times C_2)\) a rigid complement, and \( r \in \{[N \mapsto k.S]\} \) a resource with \((c_1, c_2), r) = (l_1 \cdot l_2).res s\). As \([S_1] : \downarrow [S_2]\) there exist unique
strings \( s_1 \in [S_1] \) and \( s_2 \in [S_2] \) such that \( s = (s_1 \cdot s_2) \). Using this fact, we calculate as follows

\[
dom(r) = \text{dom}(r_1 ++ r_2) \tag{by definition \( r \) and \((l_1 \cdot l_2)\).res}
\]

where \( r_1, c_1 = l_1.res \ s_1 \)
and \( r_2, c_2 = l_2.res \ s_2 \)

\[
dom(r_1) \cup \{i + \max(\text{dom}(r_1)) \mid i \in \text{dom}(r_2)\} \tag{by definition \((++\) and \dom}
\]

\[
(\text{locs}(s_1)) \cup \{i + \max(\text{locs}(s_1)) \mid i \in (\text{locs}(s_2))\} \tag{by \text{ResChunks} for \( l_1 \) and \( l_2 \)}
\]

\[
\{1, \ldots, (|s_1| + |s_2|)\} \tag{by definition \(|\cdot|\)}
\]

\[
\text{locs}(s_1 \cdot s_2) \tag{by definition \text{locs}}
\]

\[
\text{locs}(s) \tag{by definition \( s_1 \) and \( s_2 \)}
\]

and obtain the required equality.

\textbf{ChunkPut:} Let \( v \in [V_1 \cdot V_2] \) and \((c_1, c_2) \in (C_1 \times C_2)\) and \( r \in \{\mathbb{N} \mapsto k.S\} \) and \( x \in (\text{locs}(v) \cap \text{dom}(r)) \). As \([V_1] \cdot [V_2]\) there exist unique strings \( v_1 \in [V_1] \) and \( v_2 \in [V_2] \) such that \( v = (v_1 \cdot v_2) \). We analyze two cases.

\textbf{Case} \( x \in \text{locs}(v_1) \): We calculate as follows

\[
((l_1 \cdot l_2).\text{put} \ v ((c_1, c_2), r))[x] \tag{by definition \( v_1 \) and \( v_2 \)}
\]

\[
= ((l_1 \cdot l_2).\text{put} \ (v_1 \cdot v_2) ((c_1, c_2), r))[x] \tag{by definition \((l_1 \cdot l_2).\text{put}\)}
\]

\[
= ((l_1.\text{put} \ v_1 (c_1, r_1)) (l_2.\text{put} \ v_2 (c_2, r_2)))[x] \tag{by definition \((l_1 \cdot l_2).\text{put}\)}
\]

where \( r_1, r_2 = \text{split}(|v|, r) \)

\[
= (l_1.\text{put} \ v_1 (c_1, r_1))[x] \tag{by Lemma 5.2.2 and definition \([-\]}\]

\[
= k.\text{put} \ (v_1[x]) (r_1(x)) \tag{by \text{ChunkPut} for \( l_1 \)}
\]

\[
= k.\text{put} \ ((v_1 \cdot v_2)[x]) (r_1 ++ r_2(x)) \tag{by definition \([-\}] \text{ and } (++)\}
\]

\[
= k.\text{put} \ (v[x]) (r(x)) \tag{by definition \text{split} \text{ and } v_1, v_2, r_1 \text{ and } r_2\}
\]

and obtain the required equality.

\textbf{Case} \( x \notin \text{locs}(v_1) \): Similar to the previous case.

\textbf{ChunkCreate:} Similar to the proof for \text{ChunkPut}.

\textbf{NoChunkPut:} Similar to the proof for \text{ChunkPut}.

\textbf{NoChunkCreate:} Similar to the proof for \text{ChunkPut}.
Lemma:

\[ \text{Let } v \in [V_1 \cdot V_2] \text{ and } v' \in [V_1 \cdot V_2] \text{ be strings, } (c_1, c_2) \in (C_1 \times C_2) \text{ a rigid complement, and } r \in \{ \mathbb{N} \mapsto k.S \} \text{ and } r' \in \{ \mathbb{N} \mapsto k.S \} \text{ resources such that } \text{ske}(v) = \text{ske}(v'). \text{ As } [V_1 \cdot V_2] \text{ there exist unique strings } v_1 \in [V_1] \text{ and } v_2 \in [V_2] \text{ and } v'_1 \in [V_1] \text{ and } v'_2 \in [V_2] \text{ such that } v = (v_1, v_2) \text{ and } v' = (v'_1, v'_2). \text{ Moreover, using the definition of } \text{ske} \text{ we have that } \text{ske}(v_1) = \text{ske}(v'_1) \text{ and } \text{ske}(v_2) = \text{ske}(v'_2). \text{ Using these facts and definitions, we calculate as follows}
\]

\[
\begin{align*}
\text{ske}((l_1 \cdot l_2).\text{put } v (c_1, c_2, r)) &= \text{ske}((l_1 \cdot l_2).\text{put } (v_1 \cdot v_2) ((c_1, c_2), r)) & \text{by definition } v_1 \text{ and } v_2 \\
&= \text{ske}(l_1.\text{put } v_1 (c_1, r_1))(l_2.\text{put } v_2 (c_2, r_2)) & \text{by definition } (l_1 \cdot l_2).\text{put} \\
&\quad \text{where } r_1, r_2 = \text{split}([v_1], r) \\
&= \text{ske}(l_1.\text{put } v_1 (c_1, r_1)) \cdot \text{ske}(l_2.\text{put } v_2 (c_2, r_2)) & \text{by definition } \text{ske} \\
&= \text{ske}(l_1.\text{put } v'_1 (c_1, r'_1)) \cdot \text{ske}(l_2.\text{put } v'_2 (c_2, r'_2)) & \text{by SKELPUT for } l_1 \text{ and } l_2 \\
&\quad \text{where } r'_1, r'_2 = \text{split}([v'_1], r') \\
&= \text{ske}(l_1.\text{put } v'_1 (c_1, r'_1)) \cdot (l_2.\text{put } v'_2 (c_2, r'_2)) & \text{by definition } \text{ske} \\
&= \text{ske}((l_1 \cdot l_2).\text{put } (v'_1 \cdot v'_2) ((c_1, c_2), r')) & \text{by definition } (l_1 \cdot l_2).\text{put} \text{ and } r'_1 \text{ and } r'_2 \\
&= \text{ske}((l_1 \cdot l_2).\text{put } v' ((c_1, c_2), r)) & \text{by definition } v'_1 \text{ and } v'_2
\end{align*}
\]

and obtain the required equality.

- **SKELPUT:** Similar to the proof for SKELPUT, which completes the proof. □

| l \in S \overset{C.t.k}{\leftrightarrow} V | |S|^* | |V|^* |
|-----------------|-------------------|-------------------|
| l^* \in S^* \overset{(C.\text{list}, t.k)}{\leftrightarrow} V^* |

5.3.4 Lemma: Let \( l \in S \overset{C.t.k}{\leftrightarrow} V \) be a resourceful lens such that \([S]^*\) and \([V]^*\). Then \( l^* \) is a resourceful lens in \( S^* \overset{(C.\text{list}, t.k)}{\leftrightarrow} V^* \).

Proof:

- **GetPut:** Let \( s \in [S^*] \). As \([S]^*\) there exist unique strings \( s_1 \in [S] \) to \( s_n \in [S] \) such that \( s = (s_1 \ldots s_n) \). To shorten the proof, let \((c_i, r_i) = l.\text{res } s_i \) for \( i \in \{1, \ldots, n\} \) and \( r = (r_1 ++ \ldots ++ r_n) \). Also let \( r''_0 = r \) and \((r'_i, r''_i) = \text{split}([l.\text{get } s_i], r''_0) \) for \( i \in \{1, \ldots, n\} \).
Using these facts and definitions, we calculate as follows

\[
l^{*}.\text{put} \ (l^{*}.\text{gets}) \ (l^{*}.\text{res} \ s)
\]

\[
= l^{*}.\text{put} \ (l^{*}.\text{get}(s_1 \cdots s_n)) \ (l^{*}.\text{res} \ (s_1 \cdots s_n)) \quad \text{by definition } s_1 \text{ to } s_n
\]

\[
= l^{*}.\text{put} \ ((l.\text{get} s_1) \cdots (l.\text{get} s_n)) \ ([c_1, \ldots, c_n], r) \quad \text{by definition } l^{*}.\text{get} \text{ and } l^{*}.\text{res}
\]

\[
= (l.\text{put} \ (l.\text{get} s_1) \ (c_1, r'_1)) \cdots
\]

\[
(l.\text{put} \ (l.\text{get} s_n) \ (c_n, r'_n)) \quad \text{by definition } l^{*}.\text{put} \text{ with } [V]^{1*}
\]

\[
= (l.\text{put} \ (l.\text{get} s_1) \ (c_1, r_1)) \cdots
\]

\[
(l.\text{put} \ (l.\text{get} s_n) \ (c_n, r_n)) \quad \text{and definition } split
\]

\[
= (s_1 \cdots s_n) \quad \text{by GetPut for } l
\]

\[
= s \quad \text{by definition } s_1 \text{ to } s_n
\]

and obtain the required equality.

- **PutGet**: Let \( v \in [V^{*}] \) be a string, \([c_1, \ldots, c_m] \in (C \text{ list}) \) a rigid complement, and \( r \in \{[N \mapsto k.\text{res}] \} \) a resource. As \([V]^{1*}\) there exist unique strings \( v_1 \in [V] \) to \( v_n \in [V] \) such that \( v = (v_1 \cdots v_n) \). Let \( r'_0 = r \) and \( (r_i, r'_i) = \text{split}(|v_i|, r'_{i-1}) \) for \( i \in \{1, \ldots, n\} \). Using these facts and definitions, we calculate as follows

\[
l^{*}.\text{get} \ (l^{*}.\text{put} \ v \ ([c_1, \ldots, c_m], r))
\]

\[
= l^{*}.\text{get} \ (l^{*}.\text{put} \ (v_1 \cdots v_n) \ ([c_1, \ldots, c_m], r)) \quad \text{by definition of } v_1 \text{ to } v_n
\]

\[
= l^{*}.\text{get} \ (s'_1 \cdots s'_n) \quad \text{by the definition of } l^{*}.\text{put}
\]

where \( s'_i = \begin{cases} 
(\ l.\text{put} \ v_i \ (c_i, r_i) \ i \in \{1, \ldots, \min(n, m)\} \\
(\ l.\text{create} \ v_i \ r_i \ i \in \{m + 1, \ldots, n\}
\end{cases} \)

\[
= (l.\text{get} s'_1) \cdots (l.\text{get} s'_n) \quad \text{by the definition of } l^{*}.\text{get} \text{ with } [V]^{1*}
\]

\[
= (v_1 \cdots v_n) \quad \text{and cod}(l.\text{put}) = \text{cod}(l.\text{create}) = [V]
\]

\[
= v \quad \text{by PutGet and CreateGet for } l
\]

and obtain the required equality.

- **CreateGet**: Similar to the proof for PutGet.

- **GetChunks**: Let \( s \in [S^{*}] \). As \([S]^{1*}\) there exist unique strings \( s_1 \in [S] \) to \( s_n \in [S] \) such
that \( s = (s_1 \cdots s_n) \). Using these facts, we calculate as follows

\[
\text{locs}(s) \\
= \text{locs}(s_1 \cdots s_n) \\
= \{1, \ldots, \sum_{i=1}^{n} |s_i|\} \\
= \{1, \ldots, \sum_{i=1}^{n} |l.get \ s_i|\} \\
\]

by definition \( \text{locs} \)

by GETCHUNKS for \( l \)

\[
= \text{locs}(l.get(s_1) \cdots (l.get s_n)) \\
= \text{locs}(l^*.get(s_1 \cdots s_n)) \\
= \text{locs}(l^*.get \ s) \\
\]

by definition \( s_1 \) to \( s_n \)

and obtain the required equality.

**ResChunks:** Let \( s \in [S^*] \) be a string, \( c \in (C \ list) \) a rigid complement, and \( r \in \{[N \mapsto k.S]\} \) a resource such that \( (c, r) = l^*.res \ s \). As \( S^* \) there exist unique strings \( s_1 \in [S] \) to \( s_n \in [S] \) such that \( s = (s_1 \cdots s_n) \). To shorten the proof, let \( (c_i, r_i) = l^*.res \ s_i \) for \( i \in \{1, \ldots, n\} \). Using these facts and definitions, we calculate as follows

\[
\text{dom}(r) \\
= \text{dom}(r_1 ++ \cdots ++ r_n) \\
= \bigcup_{i=1}^{n} \{j + \sum_{k=1}^{(i-1)} \max(\text{dom}(r_k)) \mid j \in \text{dom}(r_i)\} \\
= \bigcup_{i=1}^{n} \{j + \sum_{k=1}^{(i-1)} \max(\text{locs}(l.get \ x_k)) \mid j \in \text{locs}(l.get s_i)\} \\
\]

by definition \( \text{dom} \)

by definition \( \text{locs} \)

\[
= \{1, \ldots, \sum_{i=1}^{n} |l.get \ s_i|\} \\
= \text{locs}(l.get(s_1) \cdots (l.get s_n)) \\
= \text{locs}(l^*.get(s_1 \cdots s_n)) \\
= \text{locs}(l^*.get \ s) \\
\]

by definition \( | \cdot | \)

by definition \( l^*.get \)

by definition \( s_1 \) to \( s_n \)

and obtain the required equality.

**ChunkPut:** Let \( v \in [V^*] \) be a string, \( [c_1, \ldots, c_m] \in (C \ list) \) a rigid complement, \( r \in \{[N \mapsto k.S]\} \) a resource, and \( x \in (\text{locs}(v) \cap \text{dom}(r)) \) a location. As \( [V]^* \) there exist unique strings \( v_1 \in [V] \) to \( v_n \in [V] \) such that \( v = (v_1 \cdots v_n) \). To shorten the proof, let \( r_0' = r \) and \( (r_i, r_i') = \text{split}((v_i, r_i') \mid V) \) for \( i \in \{1, \ldots, n\} \). We analyze several cases.
Case \( x \in \text{locs}(v_1) \): We calculate as follows

\[
(l^\ast .put \ v ([c_1, \ldots, c_m], r)) [x] \\
= (l^\ast .put \ (v_1 \cdots v_n) ([c_1, \ldots, c_m], r)) [x] \quad \text{by definition \( v_1 \) to \( v_n \)} \\
= (s'_1 \cdots s'_n) [x] \quad \text{by definition \( l^\ast .put \)} \\
\text{where } s'_i = \begin{cases} \\
  l.put \ v_i (c_i, r_i) & i \in \{1, \ldots, \min(n, m)\} \\
  l.create \ v_i r_i & i \in \{m + 1, \ldots, n\} \\
\end{cases} \\
= s'_1 [x] \quad \text{by Lemmas 5.2.2-5.2.3 and definition} [:] \\
= k.put \ v_1 (r_1 (x)) \quad \text{by CHUNKPUT and CHUNKCREATE for} \ l \\
= k.put \ (v_1 \cdots v_n) [x] \ ((r_1 ++ \ldots ++ r_n)(x)) \quad \text{by definition} [:] \text{and} ++ \\
= k.put \ v[x] (r(x)) \quad \text{by definition} \ split, \ v_1 \text{ to} \ v_n, \text{ and} \ r_1 \text{ to} \ r_n
\]

and obtain the required equality.

Case \( x \notin \text{locs}(v_1) \): Similar to the previous case.

\begin{itemize}
  \item **ChunkCreate**: Similar to the proof for CHUNKPUT.
  \item **NoChunkPut**: Similar to the proof for CHUNKPUT.
  \item **NoChunkCreate**: Similar to the proof for CHUNKPUT.
  \item **SkelPut**: Let \( v \in [V^\ast] \) and \( v' \in [V^\ast] \) be strings, \( [c_1, \ldots, c_m] \in (C \ list) \) a rigid complement, and \( r \in \{\|N \mapsto k.S\|\} \) and \( r' \in \{\|N \mapsto k.S\|\} \) resources such that \( \text{skel}(v) = \text{skel}(v') \). As \([V^\ast]\) there exist unique strings \( v_1 \in [V] \) to \( v_n \in [V] \) and \( v'_1 \in [V] \) to \( v'_n \in [V] \) such that \( v = (v_1 \cdots v_n) \) and \( v' = (v_1 \cdots v'_o) \). Moreover, by the definition of \( \text{skel} \) we have that \( n = o \) and \( \text{skel}(v_i) = \text{skel}(v'_i) \) for \( i \in \{1, \ldots, n\} \). To shorten the proof, let

\[
\begin{align*}
  r'_0 &= r \\
  (r'_i, r''_i) &= \text{split}(|v_i|, r'_{(i-1)}) \quad \text{for} \ i \in \{1, \ldots, n\} \\
  r''_0 &= r' \\
  (r''_i, r'''_i) &= \text{split}(|v'_i|, r'''_{(i-1)}) \quad \text{for} \ i \in \{1, \ldots, o\}
\end{align*}
\]
\end{itemize}
Using these facts and definitions, we calculate as follows

\[
\text{ske}(l^*\cdot \text{put} \ v ([c_1, \ldots, c_m], r)) = \text{ske}(l^*\cdot \text{put} (v_1 \cdots v_n) ([c_1, \ldots, c_m], r)) \quad \text{by definition } v_1 \text{ to } v_n
\]

\[
= \text{ske}(s'_1 \cdots s'_n) \quad \text{by the definition of } l^*\cdot \text{put}
\]

where

\[
s'_i = \begin{cases} 
\text{l.put } v_i (c_i, r_i) & i \in \{1, \ldots, \text{min}(n, m)\} \\
\text{l.create } v_i & i \in \{m + 1, \ldots, n\}
\end{cases}
\]

\[
= (\text{ske}(s'_1)) \cdots (\text{ske}(s'_n)) \quad \text{by the definition of } \text{ske}
\]

\[
= (\text{ske}(s''_1)) \cdots (\text{ske}(s''_n)) \quad \text{by } \text{SKELPUT and SKELCREATE for } l
\]

where

\[
s''_i = \begin{cases} 
\text{l.put } v'_i' (c_i, r''_i) & i \in \{1, \ldots, \text{min}(n, m)\} \\
\text{l.create } v'_i & i \in \{m + 1, \ldots, n\}
\end{cases}
\]

\[
= \text{ske}(s''_1 \cdots s''_n) \quad \text{by the definition of } \text{ske}
\]

\[
= \text{ske}(l^*\cdot \text{put} (v'_1 \cdots v'_n) ([c_1, \ldots, c_m], r'')) \quad \text{by definition } l^*\cdot \text{put} \text{ and } r''_1 \text{ to } r''_n
\]

\[
= \text{ske}(l^*\cdot \text{put} v' ([c_1, \ldots, c_m], r')) \quad \text{by definition } v'_1 \text{ to } v'_n
\]

and obtain the required equality.

\textbf{SkelCreate:} Similar to the proof for SKELPUT. \hfill \square

\[
\begin{array}{c}
|S_1| \cap |S_2| = \emptyset & |V_1| \cap |V_2| \subseteq |V_1 \cap V_2| \\

l_1 \in S_1 \xleftrightarrow{C_1+t+k} V_1 & l_2 \in S_2 \xleftrightarrow{C_2+t+k} V_2 \\

(l_1 \mid l_2) \in (S_1 \cup S_2) \xleftrightarrow{(C_1+C_2)+t+k} (V_1 \cup V_2)
\end{array}
\]

\textbf{5.3.5 Lemma:} Let \( l_1 \in S_1 \xleftrightarrow{C_1+t+k} V_1 \) and \( l_2 \in S_2 \xleftrightarrow{C_2+t+k} V_2 \) be resourceful lenses such that \( |S_1| \cap |S_2| = \emptyset \) and \( |V_1| \cap |V_2| \subseteq |V_1 \cap V_2| \). Then \((l_1 \mid l_2)\) is a resourceful lens in \((S_1 \cup S_2) \xleftrightarrow{(C_1+C_2)+t+k} (V_1 \cup V_2)\).

\textbf{Proof:}

\textbf{GetPut:} Let \( s \in |S_1 \cup S_2| \). We analyze two cases.
Case $s \in [S_1]$: We calculate as follows

\[
(l_1 | l_2).\text{put } ((l_1 | l_2).\text{get } s) ((l_1 | l_2).\text{res } s)
\]

\[
= (l_1 | l_2).\text{put } (l_1, \text{get } s) (\text{Inl}(c_1), r)
\]

by definition $(l_1 | l_2).\text{get}$ and $(l_1 | l_2).\text{res}$

where $c_1, r = l.\text{res } s$

\[
= l_1.\text{put } (l_1, \text{get } s) (c_1, r)
\]

by the definition of $(l_1 | l_2).\text{put}$

with $\text{cod}(l, \text{get}) = [V_1]$

\[
= l_1.\text{put } (l_1, \text{get } s) (l_1.\text{res } s)
\]

by the definition of $(c_1, r)$

\[
= s
\]

by PUTGET for $l_1$

and obtain the required equality, which finishes the case.

Case $s \in S_2$: Symmetric to the previous case.

\textbf{PutGet:} Let $v \in [V_1 \cup V_2]$ and $c \in (C_1 + C_2)$ and $r \in \{|N \rightarrow k.S\}$. We analyze several cases.

Case $v \in [V_1]$ and $c = \text{Inl}(c_1)$: We calculate as follows

\[
(l_1 | l_2).\text{get } ((l_1 | l_2).\text{put } v (c, r))
\]

\[
= (l_1 | l_2).\text{get } (l_1.\text{put } v (c_1, r))
\]

by definition $(l_1 | l_2).\text{put}$

\[
= l_1.\text{get } (l_1.\text{put } v (c_1, r))
\]

by definition $(l_1 | l_2).\text{get}$ with $\text{cod}(l_1.\text{put}) = [S_1]$

\[
= v
\]

by PUTGET for $l$

and obtain the required equality.

Case $v \in [V_2]$ and $c = \text{Inr}(c_2)$: Symmetric to the previous case.

Case $v \notin [V_2]$ and $c = \text{Inr}(c_2)$: We calculate as follows

\[
(l_1 | l_2).\text{get } ((l_1 | l_2).\text{put } v (c, r))
\]

\[
= (l_1 | l_2).\text{get } (l_1.\text{create } v r)
\]

by definition $(l_1 | l_2).\text{put}$

\[
= l_1.\text{get } (l_1.\text{create } v r)
\]

by definition $(l_1 | l_2).\text{get}$ with $\text{cod}(l_1.\text{create}) = [S_1]$

\[
= v
\]

by CREATEGET for $l$

and obtain the required equality.

Case $v \notin [V_1]$ and $c = \text{Inl}(c_1)$: Symmetric to the previous case.
CreateGet: Similar to the proof for PutGet.

GetChunks: Let \( s \in [S_1 \cup S_2] \). As \( ([V_1] \cap [V_2]) \subseteq [V_1 \cap V_2] \), for every \( v \in [V_1 \cap V_2] \) we have that the set of chunks of \( v \) specified by \( V_1 \) and by \( V_2 \) are identical. We analyze two cases.

Case \( s \in [S_1] \): We calculate as follows

\[
\text{locs}(s) = \text{locs}(l_1 \cdot \text{get } s) \quad \text{by GetChunks for } l_1
\]

\[
= \text{locs}((l_1 \mid l_2) \cdot \text{get } s) \quad \text{by definition } (l_1 \mid l_2) \cdot \text{get} \text{ with } s \in [S_1]
\]

and obtain the required equality.

Case \( s \in [S_2] \): Symmetric to the previous case.

ResChunks: Let \( s \in [S_1 \cup S_2] \) be a string, \( c \in (C_1 + C_2) \) a rigid complement, and \( r \in \{ [N \mapsto k.S] \} \) a resource such that \( (c, r) = (l_1 \mid l_2) \cdot \text{res } s \). We analyze two cases.

Case \( s \in [S_1] \): By the assumption of the case and the definition of \( (l_1 \mid l_2) \cdot \text{res} \) we have that \( c = \text{Inl}(c_1) \) where \( c_1, r = l_1 \cdot \text{res } s \). The required equality, \( \text{locs}(s) = \text{dom}(r) \), is immediate by ResChunks for \( l_1 \).

Case \( s \in [S_1] \): Symmetric to the previous case.

ChunkPut: Let \( v \in [V_1 \cup V_2] \) be a string, \( k \in (C_1 + C_2) \) a rigid complement, \( r \in \{ [N \mapsto k.S] \} \) a resource, and \( x \in (\text{locs}(v) \cap \text{dom}(r)) \) a location. We analyze several cases.

Case \( v \in [V_1] \) and \( c = \text{Inl}(c_1) \): As \( ([V_1] \cap [V_2]) \subseteq [V_1 \cap V_2] \), we have that \( x \) is also a location of a chunk as specified by \( V_1 \). Using this fact, we calculate as follows

\[
(l_1 \mid l_2) \cdot \text{put } v \cdot (c, r)[x]
\]

\[
= l_1 \cdot \text{put } v \cdot (c_1, r)[x] \quad \text{by definition } (l_1 \mid l_2) \cdot \text{put} \text{ with } v \in [V_1] \text{ and } c = \text{Inl}(c_1)
\]

\[
= k \cdot \text{put } (v[x]) \cdot (r(x)) \quad \text{by ChunkPut for } l_1
\]

and obtain the required equality.

Case \( v \in [V_2] \) and \( c = \text{Inr}(c_2) \): Symmetric to the previous case.
Case $v \not\in [V_2]$ and $c = \text{Inr}(c_2)$: We calculate as follows.

\[
(l_1 \mid l_2).\text{put} \ (c, r)[x] \\
= l_1.\text{create} \ r[x] \quad \text{by definition} \ (l_1 \mid l_2).\text{put} \ with \ v \not\in [V_2] \ and \ c = \text{Inr}(c_2) \\
= k.\text{put} \ (v[x]) \ (r(x)) \quad \text{by CHUNKCREATE for} \ l_1
\]

and obtain the required equality.

Case $v \not\in [V_1]$ and $c = \text{Inl}(c_1)$: Symmetric to the previous case.

- **ChunkCreate**: Similar to the proof for CHUNKPUT.
- **NoChunkPut**: Similar to the proof for CHUNKPUT.
- **NoChunkCreate**: Similar to the proof for CHUNKPUT.
- **SkelPut**: Let $v \in [V_1 \cup V_2]$ and $v' \in [V_1 \cup V_2]$ be strings, $c \in (C_1 + C_2)$ a rigid complement, and $r \in \{N \mapsto k.S\}$ and $r' \in \{N \mapsto k.S\}$ resources such that $\text{skel}(v) = \text{skel}(v')$. We analyze several cases.

Case $v \in [V_1]$ and $v' \in [V_1]$ and $c = \text{Inl}(c_1)$: Using the assumptions of the case, we calculate as follows

\[
\text{skel}((l_1 \mid l_2).\text{put} \ (c, r)) \\
= \text{skel}(l_1.\text{put} \ v \ (c_1, r)) \quad \text{by definition} \ (l_1 \mid l_2).\text{put} \ with \ v \in [V_1] \ and \ c = \text{Inl}(c_1) \\
= \text{skel}(l_1.\text{put} \ v' \ (c_1, r')) \quad \text{by SKELPUT for} \ l_1 \\
= \text{skel}((l_1 \mid l_2).\text{put} \ (c, r')) \quad \text{by definition} \ (l_1 \mid l_2).\text{put} \ with \ v' \in [V_1] \ and \ c = \text{Inl}(c_1)
\]

and obtain the required equality.

Case $v \in [V_2]$ and $v' \in [V_2]$ and $c = \text{Inr}(c_2)$: Symmetric to the previous case.

Case $v \in [V_1]$ and $v' \in [V_1]$ and $c = \text{Inr}(c_2)$: Similar to the first case.

Case $v \in [V_2]$ and $v' \in [V_2]$ and $c = \text{Inl}(c_1)$: Similar to the first case.

Case $v \in [V_1]$ and $v' \not\in [V_1]$: Can’t happen. As $\text{skel}(v) = \text{skel}(v')$, we have the sets of locations $\text{locs}(v)$ and $\text{locs}(v')$ are identical. Let $v''$ be the string obtained from $v$ by setting the chunk at every location in $\text{locs}(v)$ 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 $v' \not\in [V_1]$, which is a contradiction.
Case $v \in \lfloor V_2 \rfloor$ and $v' \notin \lfloor V_2 \rfloor$: Symmetric to the previous case.

**SkelCreate**: Similar to the proof for SkelPut, which completes the proof.

5.3.6 Lemma: Let $l_1 \in S \xleftarrow{\pi_{1,k_1}} U$ and $l_2 \in U \xrightarrow{\pi_{2,k_2}} V$ be resourceful lenses. Then $(l_1;l_2)$ is a resourceful lens in $S \xleftarrow{\pi_{(k_1,k_2)}} V$.

Proof:

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

$$(l_1;l_2).\text{put} ((l_1;l_2).\text{get} s) ((l_1;l_2).\text{res} s)$$

where $c_1, r_1 = l_1.\text{res} s$

and $c_2, r_2 = l_2.\text{res} (l_1.\text{get} s)$

$$= l_1.\text{put} (l_2.\text{put} (l_2.\text{get} (l_1.\text{get} s)) (c_1, c_2)) (c_1, r_1')$$

with $r_1', r_2' = \text{unzip} (zip r_1 r_2)$

$$= l_1.\text{put} (l_2.\text{put} (l_2.\text{get} (l_1.\text{get} s)) (c_2, r_2)) (c_1, r_1)$$

as $\text{unzip}(zip r_1 r_2) = r_1, r_2$

$$= l_1.\text{put} (l_2.\text{put} (l_2.\text{get} (l_1.\text{get} s)) (l_2.\text{res} (l_1.\text{get} s))) (l_1.\text{res} s)$$

by definition $(c_1, r_1)$ and $(c_2, r_2)$

$$= l_1.\text{put} (l_1.\text{get} s) (l_1.\text{res} s)$$

by GetPut for $l_2$

$$= s$$

by GetPut for $l_1$

and obtain the required equality.

**PutGet**: Let $v \in V$ and $(c_1, c_2) \in (C_1 \otimes C_2)$ and $r \in \{|\mathbb{N} \mapsto (k_1,k_2).C|\}$. We calculate as
follows

\[(l_1;l_2).get ((l_1;l_2).put v ((c_1, c_2), r)) = (l_1;l_2).get (l_1.put (l_2.put v (c_2, r_2)) (c_1, r_1))\]

by definition \((l_1;l_2).put\)

where \(r_1, r_2 = unzip r\)

\[= l_2.get (l_1.get (l_1.put (l_2.put v (c_2, r_2)) (c_1, r_1)))\]

by definition \((l_1;l_2).get\)

\[= l_2.get (l_2.put v (c_2, r_2))\]

by \(\text{PutGet for } l_1\)

\[= v\]

by \(\text{PutGet for } l_2\)

and obtain the required equality.

\(\blacktriangleright \text{CreateGet: Similar to the proof for PutGet.}\)

\(\blacktriangleright \text{GetChunks: Let } s \in \lfloor S \rfloor. \text{ We calculate as follows}\)

\[\text{locs}(s) = \text{locs}(l_1.get s)\]

by \(\text{GetChunks for } l_1\)

\[= \text{locs}(l_2.get (l_1.get s))\]

by \(\text{GetChunks for } l_2\)

\[= \text{locs}((l_1;l_2).get s)\]

by definition \((l_1;l_2).get\)

and obtain the required equality.

\(\blacktriangleright \text{ResChunks: Let } s \in \lfloor S \rfloor \text{ be a string, } (c_1, c_2) \in (C_1 \otimes C_2), \text{ a rigid complement, and } r \in \{\|N \rightarrow (k_1;k_2).C\}\} \text{ a resource with } (c, r) = (l_1;l_2).res s. \text{ The proof goes in three steps.}\)

First, we show that the set of locations in \(s\) is equal to the domain of the resource computed from \(s\) using \(l_1.res\).

\[\text{locs}(s) = \text{dom}(r_1)\]

by \(\text{ResChunks for } l_1\)

where \(c_1, r_2 = l_1.res s\)

Next, we show that the set of locations in \(s\) is equal to the domain of the resource computed from \((l_1.get s)\) using \(l_2.res\).

\[\text{locs}(s) = \text{locs}(l_1.get s)\]

by \(\text{GetChunks for } l_1\)

\[\text{dom}(r_2)\]

by \(\text{ResChunks for } l_2\)

where \(c_2, r_2 = l_2.res (l_1.get s)\)

Finally, using these facts, we calculate as follows

\[\text{locs}(s)\]

by definition \(zip\) with \(\text{dom}(r_1) = \text{locs}(s) = \text{dom}(r_2)\)

\[= \text{dom}(r)\]

by definition \((l_1;l_2).res\)
and obtain the required equality.

**ChunkPut**: Let \( v \in [V] \) be a string, \( \langle c_1, c_2 \rangle \in (C_1 \otimes C_2) \) a rigid complement, \( r \in \{ \mathbb{N} \mapsto (k_1, k_2).C \} \) a resource, and \( x \in (\text{loess}(v) \cap \text{dom}(r)) \) a location. We calculate as follows

\[
(l_1; l_2).\text{put} \ v \ ((c_1, c_2), \ r))[x] = (l_1.\text{put} \ (l_2.\text{put} \ (c_2, r_2)) \ (c_1, r_1))[x] \quad \text{by definition} \ (l_1; l_2).\text{put}
\]

where \( r_1, r_2 = \text{unzip} \ r \)

\[
= k_1.\text{put} \ ((l_2.\text{put} \ (c_2, r_2))[x]) \ (r_1(x)) \quad \text{by ChunkPut for} \ l_1
\]

\[
= k_1.\text{put} \ (k_2.\text{put} \ (v[x]) \ (r_2(x))) \ (r_1(x)) \quad \text{by ChunkPut for} \ l_2
\]

\[
= (k_1; k_2).\text{put} \ (v[x]) \ (r_1(x), r_2(x)) \quad \text{by definition of} \ (k_1; k_2).\text{put}
\]

\[
= (k_1; k_2).\text{put} \ (v[x]) \ (r(x)) \quad \text{by definition} \ (r_1, r_2) \text{ and unzip}
\]

and obtain the required equality.

**ChunkCreate**: Similar to the proof for **ChunkPut**.

**NoChunkPut**: Similar to the proof for **ChunkPut**.

**NoChunkCreate**: Similar to the proof for **ChunkPut**.

**SkelPut**: Let \( v \in [V] \) and \( v' \in [V] \) be strings \( \langle c_1, c_2 \rangle \in C_1 \otimes C_2 \) a rigid complement, and \( r \in \{ \mathbb{N} \mapsto k.S \} \) and \( r' \in \{ \mathbb{N} \mapsto k.S \} \) resources such that \( \text{skel}(v) = \text{skel}(v') \). To shorten the proof, let \( r_1 \) and \( r_2 \) and \( r_1' \) and \( r_2' \) be resources, and \( u \) and \( u' \) be strings defined as follows:

\[
r_1, r_2 = \text{unzip} \ r
\]

\[
r_1', r_2' = \text{unzip} \ r'
\]

\[
u = l_2.\text{put} \ v \ (c_2, r_2)
\]

\[
u' = l_2.\text{put} \ v' \ (c_2, r_2')
\]

Observe that \( \text{skel}(u) = \text{skel}(u') \) by **SkelPut** for \( l_2 \). Using these facts and definitions, we calculate as follows

\[
\text{skel}((l_1; l_2).\text{put} \ v \ ((c_1, c_2), \ r))
\]

\[
= \text{skel}(l_1.\text{put} \ (l_2.\text{put} \ (c_2, r_2)) \ (c_1, r_1)) \quad \text{by definition} \ (l_1; l_2).\text{put}
\]

\[
= \text{skel}(l_1.\text{put} \ u \ (c_1, r_1)) \quad \text{by definition} \ u
\]

\[
= \text{skel}(l_1.\text{put} \ u' \ (c_1, r_1')) \quad \text{by **SkelPut** for} \ l_1
\]

\[
= \text{skel}(l_1.\text{put} \ (l_2.\text{put} \ v' \ (c_2, r_2')) \ (c_1, r_1')) \quad \text{by definition} \ u'
\]

\[
= \text{skel}((l_1; l_2).\text{put} \ v' \ ((c_2, c_2), \ r')) \quad \text{by definition} \ (l_1; l_2).\text{put}
\]
and obtain the required equality.

**Lemma: Let $l_1 \in S_1 \xrightleftharpoons{(C_1 \times t)^k} V_1$ and $l_2 \in S_2 \xrightleftharpoons{(C_2 \times t)^k} V_2$. Then $(l_1 \sim l_2)$ is a resourceful lens in $(S_1 \cdot S_2) \xrightleftharpoons{(C_2 \times C_1)^k} (V_2 \cdot V_1)$.**

**Proof:**

**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 this fact, we calculate as follows:

\[
(l_1 \sim l_2).\text{put} ((l_1 \sim l_2).\text{get} s) ((l_1 \sim l_2).\text{res} s)
\]

\[
= (l_1 \sim l_2).\text{put} ((l_1 \sim l_2).\text{get} (s_1 \cdot s_2)) ((l_1 \sim l_2).\text{res} (s_1 \cdot s_2))
\]

by definition $s_1$ and $s_2$

\[
= (l_1 \sim l_2).\text{put} ((l_2.\text{get} s_2) \cdot (l_1.\text{get} s_1)) ((c_2, c_1), r_2 ++ r_1)
\]

where $c_1, r_1 = l_1.\text{res} s_1$

and $c_2, r_2 = l_2.\text{res} s_2$

\[
= (l_1.\text{put} (l_1.\text{get} s_1) (c_1, r_1')) \cdot (l_2.\text{put} (l_2.\text{get} s_2) (c_2, r_2'))
\]

by definition $(l_1 \sim l_2).\text{put}$ with $[V_2].1[V_1]$ and $\text{cod}(l_2.\text{get}) = [V_2]$ and $\text{cod}(l_1.\text{get}) = [V_1]$

and $\text{GetChunks}$ for $l_1$

and $\text{ResChunks}$ for $l_1$

and definition $\text{split}$

\[
= (l_1.\text{put} (l_1.\text{get} s_1) (l_1.\text{res} s_1)) \cdot (l_2.\text{put} (l_2.\text{get} s_2) (l_2.\text{res} s_2))
\]

by definition $(c_1, r_1)$ and $(c_2, r_2)$

and $\text{GetPut}$ for $l_1$ and $l_2$

by definition $s_1$ and $s_2$

\[
= (s_1 \cdot s_2)
\]

\[
= s
\]

and obtain the required equality.
**PutGet:** Let \( v \in [V_1 \cdot V_2] \) and \((c_2, c_1) \in (C_2 \times C_1)\) and \( r \in \{\mathbb{N} \mapsto k. S\}\). As \( [V_2]^{-1} [V_1] \) there exist unique strings \( v_2 \in [V_2] \) and \( v_1 \in [V_1] \) such that \( v = (v_2 \cdot v_1) \). Using this fact, we calculate as follows

\[
(l_1 \sim l_2).get ((l_1 \sim l_2).put v ((c_2, c_1), r))
\]

\[
= (l_1 \sim l_2).get ((l_1 \sim l_2).put (v_2 \cdot v_1) ((c_2, c_1), r)) \quad \text{by definition} \ v_2 \ \text{and} \ v_1
\]

\[
= (l_1 \sim l_2).get ((l_1. put v_1 (c_1, r_1)) \cdot (l_2. put v_2 (c_2, r_2))) \quad \text{by definition} \ (l_1 \sim l_2).put
\]

where \( r_2, r_1 = \text{split}(v_2, r) \)

\[
= (l_2. get (l_2. put v_2 (c_2, r_2))).
\]

\[
(l_1. get (l_1. put v_1 (c_1, r_1)))
\]

\[
= (v_2 \cdot v_1) \quad \text{by definition} \ (l_1 \sim l_2).get
\]

\[
= v \quad \text{by PutGet for} \ l_2 \ \text{and} \ l_1
\]

and obtain the required equality.

**CreateGet:** Similar to the proof for PutGet.

**GetChunks:** Let \( s \in [S_1 \cdot S_2] \). As \( [S_1]^{-1} [S_2] \) there exist unique strings \( s_1 \in [S_1] \) and \( s_2 \in [S_2] \) such that \( s = s_1 \cdot s_2 \). Using this fact, we calculate as follows

\[
\text{locs}(s)
\]

\[
= \text{locs}(s_1 \cdot s_2) \quad \text{by definition} \ s_1 \ \text{and} \ s_2
\]

\[
= \{1, \ldots, (|s_1| + |s_2|)\} \quad \text{by definition} \ \text{locs}
\]

\[
= \{1, \ldots, (|l_1. \text{get } s_1| + |l_2. \text{get } s_2|)\} \quad \text{by GetChunks for} \ l_1 \ \text{and} \ l_2
\]

\[
= \text{locs}(l_2. \text{get } s_2) \cdot (l_1. \text{get } s_1) \quad \text{by definition} \ \text{locs}
\]

\[
= \text{locs}(l_1 \sim l_2).get (s_1 \cdot s_2) \quad \text{by definition} \ (l_1 \sim l_2).get
\]

\[
= \text{locs}(l_1 \sim l_2).get s \quad \text{by definition} \ s_1 \ \text{and} \ s_2
\]

and obtain the required equality.

**ResChunks:** Let \( s \in [S_1 \cdot S_2] \) be a string, \((c_1, c_2) \in (C_1 \times C_2)\) a rigid complement, and \( r \in \{\mathbb{N} \mapsto k. S\}\) a resource with \( ((c_1, c_2), r) = (l_1 \sim l_2).\text{res } s \). As \( [S_1]^{-1} [S_2] \) there exist unique
strings \( s_1 \in [S_1] \) and \( s_2 \in [S_2] \) such that \( s = (s_1 \cdot s_2) \). Using this fact, we calculate as follows:

\[
\begin{align*}
\text{dom}(r) &= \text{dom}(r_2 ++ r_1) & \text{by definition } r \text{ and } (l_1 \sim l_2).res \\
\text{where } r_1, c_1 &= l_1.res \ s_1 \\
\text{and } r_2, c_2 &= l_2.res \ s_2 \\
\text{by definition } s_1 \text{ and } s_2 \\
\end{align*}
\]

Let \( v = (v_1, r_2, r_1) \) such that \( l_1 \sim l_2 \) and \( (v_1, r_1, r_2) \) is a location satisfying \( v(y) = x \) and the definition of \( (\cdot) \). Using these facts, we calculate as follows:

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } v_2 \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } (\cdot) \text{ and } (\cdot) \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } (\cdot) \text{ and } (\cdot) \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } (\cdot) \text{ and } (\cdot) \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } (\cdot) \text{ and } (\cdot) \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } (\cdot) \text{ and } (\cdot) \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } (\cdot) \text{ and } (\cdot) \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } (\cdot) \text{ and } (\cdot) \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]

\[
\begin{align*}
(l_1 \sim l_2).put \ ((v_2, c_2, r_2)) &= (l_1 \sim l_2).put \ ((v_1, c_1, r_1))(s) & \text{by definition } (\cdot) \text{ and } (\cdot) \\
\text{where } r_2, r_2 &= \text{split}(v_2, r) \\
\text{by definition } \cdot \\
\end{align*}
\]
and obtain the required equality.

**Case** $x \not\in \text{locs}(v_2)$: Similar to the previous case.

- **ChunkCreate**: Similar to the proof for **CHUNKPUT**.

- **SkelPut**: Let $v \in [V_2 \leftrightarrow V_1]$ and $v' \in [V_2 \leftrightarrow V_1]$ be strings, $(c_1, c_2) \in (C_1 \times C_2)$ a rigid complement, and $r \in \{N \leftrightarrow k.S\}$ and $r' \in \{N \leftrightarrow k.S\}$ resources such that $\text{skel}(v) = \text{skel}(v')$. As $[V_2] \downarrow [V_1]$ there exist unique strings $v_2 \in [V_2]$ and $v_1 \in [V_1]$ and $v'_2 \in [V_2]$ and $v'_1 \in [V_1]$ such that $v = (v_2 \cdot v_1)$ and $v' = (v'_2 \cdot v'_1)$. Moreover, from the definition of **skel** we have that $\text{skel}(v_2) = \text{skel}(v'_2)$ and $\text{skel}(v_1) = \text{skel}(v'_1)$. Using these facts and definitions, we calculate as follows

$$\begin{align*}
\text{skel}((l_1 \sim l_2).\text{put}(c_1, c_2, r)) &= \text{skel}((l_1 \sim l_2).\text{put}(v_2 \cdot v_1) ((c_1, c_2), r)) \quad \text{by definition } v_2 \text{ and } v_1 \\
&= \text{skel}(l_1.\text{put} v_1 (c_1, r_1)) \cdot (l_2.\text{put} v_2 (c_2, r_2)) \quad \text{by definition } (l_1 \sim l_2).\text{put} \\
& \quad \text{where } r_1, r_2 = \text{split}(|v_2|, r) \\
&= \text{skel}(l_1.\text{put} v_1 (c_1, r_1)) \cdot \text{skel}(l_2.\text{put} v_2 (c_2, r_2)) \quad \text{by definition } \text{skel} \\
&= \text{skel}(l_1.\text{put} v'_1 (c_1, r'_1)) \cdot \text{skel}(l_2.\text{put} v'_2 (c_2, r'_2)) \quad \text{by SKELPUT for } l_1 \text{ and } l_2 \\
& \quad \text{where } r'_1, r'_2 = \text{split}(|v'_1|, r') \\
&= \text{skel}(l_1.\text{put} v'_1 (c_1, r'_1)) \cdot (l_2.\text{put} v'_2 (c_2, r'_2)) \quad \text{by definition } \text{skel} \\
&= \text{skel}((l_1 \sim l_2).\text{put}(v'_2 \cdot v'_1) ((c_1, c_2), r')) \quad \text{by definition } (l_1 \sim l_2).\text{put} \text{ and } r'_1 \text{ and } r'_2 \\
&= \text{skel}((l_1 \sim l_2).\text{put}(v' ((c_1, c_2), r)) \quad \text{by definition } v'_2 \text{ and } v'_1 \\
\end{align*}$$

and obtain the required equality.

- **SkelCreate**: Similar to the proof for **SKELPUT**.

**Secure Lens Proofs**

**6.1.1 Lemma**: The **redact** lens is a secure lens at the following type:

$$((\text{SPACE-TIME-DESC-LOCATION-NEWLINE}):T \mid (\text{ASTERISK-TIME-DESC-LOCATION-NEWLINE}):E)^*$$

$$\leftrightarrow$$

$$((\text{TIME-DESC-NEWLINE}):T \mid (\text{TIME-BUSY-NEWLINE}):E)^*$$

**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:T ) .
copy ( ( TIME . DESC ):T ) .
del ( LOCATION:T ) .
copy ( NEWLINE:T )

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 E data since every regular expression R is equivalent
to R:E in the two-point integrity lattice.

By the typing rules for del, copy, <->, and concatenation we have:

\[ \text{public} \in (\text{SPACE-TIME-DESC-LOCATION-NEWLINE}):T \quad \leftrightarrow \quad (\text{TIME-DESC-NEWLINE}):T \]
\[ \text{private} \in (\text{ASTERISK-TIME-DESC-LOCATION-NEWLINE}) \leftrightarrow (\text{TIME-BUSY-NEWLINE}) \]

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—e.g.,
between \((R:T)-(S:T)):T\) and \((R-S):T\).

By the typing rule for iteration, using the equivalence between \(R:T)^*:T\) and \((R:T)^*:T\), we have:

\[ \text{public*} \in ((\text{SPACE-TIME-DESC-LOCATION-NEWLINE}):T)^* \leftrightarrow ((\text{TIME-DESC-NEWLINE})^*:T) \]

Next, by the typing rule for concatenation, and as E observes the unambiguous concatenability
of the types in the view, we have:

\[ \text{private . public*} \in ((\text{ASTERISK-TIME-DESC-LOCATION-NEWLINE}) \cdot ((\text{SPACE-TIME-DESC-LOCATION-NEWLINE}):T)^*) \leftrightarrow ((\text{TIME-BUSY-NEWLINE}) \cdot (\text{TIME-DESC-NEWLINE}):T)^* \]

Then, by the typing rule for iteration, as E observes the unambiguous iterability of the types in
the view, we have:

\[
\begin{align*}
\text{(private . public\*)*} \in & \quad ((\text{ASTERISK\cdot TIME\cdot DESC\cdot LOCATION\cdot NEWLINE}) \\
& \cdot ((\text{SPACE\cdot TIME\cdot DESC\cdot LOCATION\cdot NEWLINE}) : \text{T})^*)^* \\
\iff & \quad ((\text{TIME\cdot BUSY\cdot NEWLINE}) \\
& \cdot ((\text{TIME\cdot DESC\cdot NEWLINE}) : \text{T})^*)^*
\end{align*}
\]

Finally, by the typing rule for concatenation, and again as \( E \) observes the unambiguous concatenability of the types in the view, we have:

\[
\begin{align*}
\text{public\* . (private . public\*)*} \in & \quad (((\text{SPACE\cdot TIME\cdot DESC\cdot LOCATION\cdot NEWLINE})^*) : \text{T})^* \\
& \cdot (\text{ASTERISK\cdot TIME\cdot DESC\cdot LOCATION\cdot NEWLINE}) \\
& \cdot (((\text{SPACE\cdot TIME\cdot DESC\cdot LOCATION\cdot NEWLINE})^*) : \text{T})^* \\
\iff & \quad (((\text{TIME\cdot DESC\cdot NEWLINE})^*) : \text{T})^* \\
& \cdot (\text{TIME\cdot BUSY\cdot NEWLINE}) \\
& \cdot (((\text{TIME\cdot DESC\cdot NEWLINE})^*) : \text{T})^*
\end{align*}
\]

The equivalent type stated in the lemma can be obtained using the equivalence between \((R : T)^* \cdot (S \cdot (R : T)^*)^*\) and \(((R : T) \mid S)^*\) which holds when \( R \) and \( S \) are disjoint and unambiguously iterable.

6.2.2 Lemma: Secure lenses admit the following inference rule:

\[
\begin{align*}
\text{PutPutEndorsed} & \\
\frac{v' \approx_k l.\text{get } s \approx_k v}{l.\text{put } v' (l.\text{put } v \ s) \approx_k l.\text{put } v' \ s}
\end{align*}
\]

Proof: Let \( k \in \mathcal{Q} \) be a label, and let \( v, v' \in V \) and \( s \in S \) be strings such that \( v \approx_k (l.\text{get } s) \) and \( v' \approx_k (l.\text{get } s) \). By PutGet for \( l \) we have:

\[
l.\text{get } (l.\text{put } v \ s) = v
\]

By the reflexivity of \( \approx_k \) it follows that:

\[
l.\text{get } (l.\text{put } v \ s) \approx_k v
\]
Using this fact, we calculate as follows:

\[
\begin{align*}
  l_.\text{put } v' (l_.\text{put } v\ s) \\
  \approx_k l_.\text{put } v\ s & \text{ by GetPut for } l \\
  \approx_k s & \text{ by GetPut for } l
\end{align*}
\]

Also by GetPut for \(l\) we have

\[
l_.\text{put } v'\ s \approx_k s
\]

The required equivalence follows from the transitivity of \(\approx_k\). \(\square\)

\[
\begin{array}{c}
E \text{ well-formed} \\
\text{copy } E \in E \iff E
\end{array}
\]

6.4.1 Lemma: Let \(E \in \mathcal{R}\) be a security-annotated regular expression. Then \((\text{copy } E)\) is a secure lens in \(E \iff E\).

Proof:

\begin{itemize}
  \item \textbf{GetPut:} Let \(k \in \mathbb{Q}\) and \(e, e' \in E\) with \(e' \approx_k (\text{copy } E).\text{get } e\). We calculate as follows

  \[
  (\text{copy } E).\text{put } e'\ e \\
  = e' & \text{ by definition of } (\text{copy } E).\text{put} \\
  \approx_k (\text{copy } E).\text{get } e & \text{ by assumption} \\
  = e & \text{ by definition of } (\text{copy } E).\text{get}
  \]

  and obtain the required equivalence.

  \item \textbf{PutGet:} Let \(e, e' \in E\). We calculate as follows

  \[
  (\text{copy } E).\text{get } ((\text{copy } E).\text{put } e'\ e) \\
  = (\text{copy } E).\text{get } e' & \text{ by definition of } (\text{copy } E).\text{put} \\
  = e' & \text{ by definition of } (\text{copy } E).\text{get}
  \]

  and obtain the required equality.
\end{itemize}

235
GetNoLeak: Let \( j \in \mathcal{P} \) and \( e, e' \in E \) with \( e \sim j e' \). We calculate as follows

\[
\begin{align*}
\text{(copy } E\text{).get } e &= e & \text{by definition of (copy } E\text{).get} \\
\sim_j e' &= e' & \text{by assumption} \\
= (\text{copy } E\text{).get } e' & \text{by definition of (copy } E\text{).get}
\end{align*}
\]

and obtain the required equivalence, which completes the proof.

\[
\begin{array}{|c|c|}
\hline
\text{E, F well-formed} & |\mathcal{L}(F)| = 1 \\
\hline
\text{const } E F \in E \leftrightarrow F \\
\hline
\end{array}
\]

6.4.2 Lemma: Let \( E \) and \( F \) be well-formed security-annotated regular expressions such that \( |\mathcal{L}(F)| = 1 \). Then \((\text{const } E F)\) is a secure lens in \( E \leftrightarrow F \).

Proof:

GetPut: Let \( q \in \mathcal{Q} \) and \( e \in E \) and \( u \in F \) with \( u \approx_q (\text{const } E F).\text{get } e \). By the definition of \((\text{const } E F).\text{put}\) we have \((\text{const } E F).\text{put } u e = e\). The required equivalence follows as \( \approx_k \) is reflexive.

PutGet: Let \( u \in F \) and \( e \in E \). We calculate as follows

\[
\begin{align*}
(\text{const } E F).\text{get } (\text{(const } E F).\text{put } u e) &= (\text{const } E F).\text{get } e & \text{by definition of (const } E F).\text{put} \\
&= \text{representative}(F) & \text{by definition of (const } E F).\text{get} \\
&= u & \text{as } |\mathcal{L}(F)| = 1
\end{align*}
\]

and obtain the required equality.

CreateGet: Similar to the proof of PutGet.

GetNoLeak: Let \( j \in \mathcal{P} \) and \( e, e' \in E \) with \( e \sim_j e' \). Let \( u \) be the unique element of \( F \). By the definition of \((\text{const } E F).\text{get}\) we have \((\text{const } E F).\text{get } e = u = (\text{const } E F).\text{get } e'\). The required equivalence follows as \( \sim_j \) is reflexive, which completes the proof.
6.4.3 Lemma: Let \( l_1 \in S_1 \leftrightarrow V_1 \) and \( l_2 \in S_2 \leftrightarrow V_2 \) be secure lenses such that \((S_1 \cap S_2) = \emptyset\). Then \((l_1 \mid l_2)\) is a secure lens in \((S_1 \mid S_2): q \leftrightarrow (V_1 \mid V_2): p\) where \( q = \bigvee \{ k \mid k \text{ min obs. } V_1 \neq V_2 \wedge V_1 \& V_2 \text{ agree} \} \) and \( p = \bigvee \{ k \mid k \text{ min obs. } (S_1 \cap S_2) = \emptyset \} \).

Proof:

\( \blacktriangleright \) GetPut: Let \( k \in Q \) and \( s \in (S_1 \mid S_2): q \) and \( v \in (V_1 \mid V_2): p \) with \( v \approx_k \text{get } s \). We analyze two cases.

Case \( k \not\supseteq q \): Then \( \approx_k \) is the total relation on \((S_1 \mid S_2): q\) and \((l_1 \mid l_2).\text{put } v \approx_k \text{get } s \) trivially.

Case \( k \supseteq q \) and \( v \in V_1 \) and \( s \in S_1 \): From

\[
v \in V_1, \quad v \approx_k^{(V_1 \mid V_2): p} (l_1 \mid l_2).\text{get } s \quad k \text{ observes } V_1 \& V_2 \text{ agree}
\]

we have \( v \approx_k^{V_1} l_1.\text{get } s \). Using this fact, we calculate as follows

\[
(l_1 \mid l_2).\text{put } v \approx_k \text{get } s
\]

by definition of \((l_1 \mid l_2).\text{get}\)

and obtain the required equivalence.

Case \( k \supseteq q \) and \( v \in V_2 \) and \( s \in S_2 \): Symmetric to the previous case.

Case \( k \supseteq q \) and \( v \in V_2 - V_1 \) and \( s \in S_1 \): Can’t happen. The assumptions \( v \approx_k (l_1 \mid l_2).\text{get } s \) and \( k \) observes \( V_1 \neq V_2 \) lead to a contradiction.

Case \( k \supseteq q \) and \( v \in V_1 - V_2 \) and \( s \in S_2 \): Symmetric to the previous case.

\( \blacktriangleright \) PutGet: Let \( v \in (V_1 \mid V_2): p \) and \( s \in (S_1 \mid S_2): q \). We analyze several cases.
Case \( v \in V_1 \) and \( s \in S_1 \): We calculate as follows (\( \text{cod}(\cdot) \) denotes the codomain of a function)

\[
(l_1 \mid l_2).\text{get}((l_1 \mid l_2).\text{put} v s) \\
= (l_1 \mid l_2).\text{get} (l_1.\text{put} v s) \\
= l_1.\text{get} (l_1.\text{put} v s) \\
= v
\]

by definition of \( (l_1 \mid l_2).\text{put} \) as \( \text{cod}(l_1.\text{put}) = S_1 \)

where the last line follows by \text{PutGet} for \( l_1 \).

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

\[
(l_1 \mid l_2).\text{get}((l_1 \mid l_2).\text{put} v s) \\
= (l_1 \mid l_2).\text{get} (l_1.\text{create} v) \\
= l_1.\text{get} (l_1.\text{create} v) \\
= v
\]

by \text{CreateGet} for \( l_1 \)

and obtain the required equality.

Case \( v \in (V_2 - V_1) \) and \( s \in S_1 \): Symmetric to the previous case.

\[\textbf{GetNoLeak}:\] Let \( j \in \mathcal{P} \) and \( s, s' \in (S_1 \mid S_2):q \) with \( s \sim_j s' \). We analyze two cases.

Case \( j \not\sqsubseteq p \): Then \( \sim_j \) is the total relation on \( (V_1 \mid V_2):p \) so \( (l_1 \mid l_2).\text{get} s \sim_j (l_1 \mid l_2).\text{get} s' \) trivially.

Case \( j \sqsubseteq p \) and \( s \in S_1 \): From

\[
s \in S_1 \quad s \sim_j s' \quad k \text{ observes } (S_1 \cap S_2) = \emptyset
\]

we have \( s' \in S_1 \) and \( s \sim_k^{S_1} s' \). Using these facts, we calculate as follows

\[
(l_1 \mid l_2).\text{get} s \\
= l_1.\text{get} s \\
\sim_k l_1.\text{get} s' \\
= (l_1 \mid l_2).\text{get} s'
\]

by \text{GetNoLeak} for \( l_1 \)

and obtain the required equivalence.

Case \( j \sqsubseteq p \) and \( s \in S_2 \): Symmetric to the previous case. \( \square \)
6.4.4 Lemma: Let \( l_1 \in S_1 \xleftrightarrow{\ } V_1 \) and \( l_2 \in S_2 \xleftrightarrow{\ } V_2 \) such that \( S_1 \xleftarrow{\ } S_2 \) and \( V_1 \xleftarrow{\ } V_2 \). Then \((l_1 \cdot l_2)\) is a secure lens in \((S_1 \cdot S_2): q \xleftrightarrow{\ } (V_1 \cdot V_2): p\) where \( q = \bigvee \{ k \mid k \text{ min obs. } V_1 \cdot V_2 \} \) and \( p = \bigvee \{ k \mid k \text{ min obs. } S_1 \cdot S_2 \} \).

Proof:

\[ l_1 \in S_1 \xleftrightarrow{\ } V_1 \quad S_1 \xleftarrow{\ } S_2 \quad l_2 \in S_2 \xleftrightarrow{\ } V_2 \quad V_1 \xleftarrow{\ } V_2 \]

\[ q = \bigvee \{ k \mid k \text{ min obs. } V_1 \cdot V_2 \} \]

\[ p = \bigvee \{ k \mid k \text{ min obs. } S_1 \cdot S_2 \} \]

\[ l_1 \cdot l_2 \in (S_1 \cdot S_2): q \xleftrightarrow{\ } (V_1 \cdot V_2): p \]

\[ \begin{align*}
\text{\textbf{GetPut:}} & \quad \text{Let } k \in Q \text{ be a label, } (s_1 \cdot s_2) \in (S_1 \cdot S_2): q \text{ a string, and } (v_1 \cdot v_2) \in (V_1 \cdot V_2): p \text{ a string such that } (v_1 \cdot v_2) \approx_k (l_1 \cdot l_2).\text{get } (s_1 \cdot s_2). \text{ We analyze two cases.} \\
\text{Case } k \nmid q: & \quad \text{Then } \approx_k \text{ is the total relation on } (S_1 \cdot S_2): q \text{ and so } (l_1 \cdot l_2).\text{put } (v_1 \cdot v_2) (s_1 \cdot s_2) \approx_k (s_1 \cdot s_2) \text{ trivially.} \\
\text{Case } k \mid q: & \quad \text{From the assumption of the case we have that } k \text{ observes } V_1 \cdot V_2 \text{ and so we also have} \\
\quad & \quad v_1 \approx_{V_1} l_1.\text{get } s_1 \quad v_2 \approx_{V_2} l_2.\text{get } s_2 \\
\quad & \quad \text{using the definition of } (l_1 \cdot l_2).\text{get}. \text{ Using these equivalences, we calculate as follows} \\
\quad & \quad (l_1 \cdot l_2).\text{put } (v_1 \cdot v_2) (s_1 \cdot s_2) \\
\quad & \quad = (l_1.\text{put } v_1 s_1) \cdot (l_2.\text{put } v_2 s_2) \quad \text{by definition of } (l_1 \cdot l_2).\text{put} \\
\quad & \quad \approx_k s_1 \cdot s_2 \quad \text{by GetPut for } l_1 \text{ and } l_2 \\
\quad & \quad \text{and obtain the required equivalence.} \\
\text{\textbf{PutGet:}} & \quad \text{Let } (v_1 \cdot v_2) \in (V_1 \cdot V_2): p \text{ and } (s_1 \cdot s_2) \in (S_1 \cdot S_2): q. \text{ We calculate as follows} \\
\quad & \quad (l_1 \cdot l_2).\text{get } ((l_1 \cdot l_2).\text{put } (v_1 \cdot v_2) (s_1 \cdot s_2)) \\
\quad & \quad = (l_1 \cdot l_2).\text{get } ((l_1.\text{put } v_1 s_1) \cdot (l_2.\text{put } v_2 s_2)) \quad \text{by definition of } (l_1 \cdot l_2).\text{put} \\
\quad & \quad = (l_1.\text{get } (l_1.\text{put } v_1 s_1)) \cdot (l_2.\text{get } (l_2.\text{put } v_2 s_2)) \quad \text{by definition of } (l_1 \cdot l_2).\text{get with } S_1 \cdot S_2 \\
\quad & \quad = v_1 \cdot v_2 \quad \text{by PutGet for } l_1 \text{ and } l_2 \\
\end{align*} \]
and obtain the required equality.

**GetNoLeak:** Let \( j \in P \) and \((s_1 \cdot s_2), (s'_1 \cdot s'_2) \in (S_1 \cdot S_2)\); \( q \) with \((s_1 \cdot s_2) \sim_j (s'_1 \cdot s'_2)\). We analyze two cases.

Case \( j \nsubseteq p \): Then \( \sim_j \) is the total relation on \((V_1 \cdot V_2)\); \( p \) and so \((l_1 \cdot l_2).get (s_1 \cdot s_2) \sim_j (l_1 \cdot l_2).get (s'_1 \cdot s'_2)\) trivially.

Case \( j \subseteq p \): From the assumption of the case we have that \( j \) observes \( S_1 \cdot S_2\) and so:

\[
s_1 \sim_j^{S_1} s'_1 \quad s_2 \sim_j^{S_2} s'_2
\]

Using these equivalences, we calculate as follows

\[
(l_1 \cdot l_2).get (s_1 \cdot s_2) = (l_1.get s_1) \cdot (l_2.get s_2) \quad \text{by definition of (} l_1 \cdot l_2 \text{).get}
\]

\[
\sim_k (l_1.get s_1) \cdot (l_2.get s_2) \quad \text{by GetNoLeak for } l_1 \text{ and } l_2 \text{ and the definition of } \sim_k
\]

\[
= (l_1 \cdot l_2).get (s'_1 \cdot s'_2) \quad \text{by definition of (} l_1 \cdot l_2 \text{).get}
\]

and obtain the required equivalence, which completes the proof.

\[\square\]

\[\begin{array}{l}
l \in S \quad \text{V} \quad S^* \quad V^* \\
q = \bigvee \{k \mid k \text{ min obs. } V^*\} \\
p = \bigvee \{k \mid k \text{ min obs. } S^*\}
\end{array}\]

\[l^* \in (S^*): q \quad \text{V} \quad (V^*): p\]

**6.4.5 Lemma:** Let \( l \in S \leftrightarrow V \) be a secure lens such that \( S^* \) and \( V^* \). Then \( l^* \) is a secure lens in \((S^*): q \leftrightarrow (V^*): p\) where \( q = \bigvee \{k \mid k \text{ min obs. } V^*\}\) and \( p = \bigvee \{k \mid k \text{ min obs. } S^*\}\).

**Proof:**

**GetPut:** Let \( k \in Q \) be a label, \((s_1 \cdots s_m) \in (S^*): q \) a string, and \( v_1 \cdots v_n \in (V^*): p \) a string such that \( (v_1 \cdots v_n) \approx_k l^* . get (s_1 \cdots s_m)\). We analyze two cases:

Case \( k \nsubseteq q \): Then \( \approx_k \) is the total relation on \((S^*): q\) and we trivially have:

\[
l^* . put (v_1 \cdots v_n) (s_1 \cdots s_m) \approx_k (s_1 \cdots s_m)
\]

and obtain the required equivalence.
Case $k \supseteq q$: From

$$k \text{ observes } V^* : (v_1 \cdots v_n) \approx_k l^* \text{.get } (s_1 \cdots s_m)$$

we have

$$m = n \quad v_i \approx_k l \text{.get } s_i \text{ for } i \in \{1, \ldots, n\}$$

Using these facts, we calculate as follows

$$l^* \text{.put } (v_1 \cdots v_n) (s_1 \cdots s_n)$$

$$= (l \text{.put } v_1 s_1) \cdots (l \text{.put } v_n s_n) \quad \text{by the definition of } l^* \text{.put}$$

$$\approx_k (s_1 \cdots s_n) \quad \text{by } \text{GetPut for } l \text{ (} n \text{ times)}$$

and obtain the required equivalence.

**PutGet**: Let $(v_1 \cdots v_n) \in (V^*):p$ and $(s_1 \cdots s_m) \in (S^*):q$. We calculate as follows

$$l^* \text{.get } (l^* \text{.put } (v_1 \cdots v_n) (s_1 \cdots s_m))$$

$$= l^* \text{.get } (s'_1 \cdots s'_n)$$

where $s'_i = \begin{cases} l \text{.put } v_i s_i & \text{for } i \in \{1, \ldots, \min m, n\} \\ l \text{.create } v_i & \text{for } i \in \{m + 1, \ldots, n\} \end{cases}$

$$= (l \text{.get } s'_1) \cdots (l \text{.get } s'_n) \quad \text{by definition of } l^* \text{.get as } S^l*$$

$$= (v_1 \cdots v_n) \quad \text{by } \text{PutGet for } l \text{ (} m \text{ times)}$$

and createGet for $l \text{ (} n - m \text{ times)}$

and obtain the required equality.

**GetNoLeak**: Let $j \in \mathcal{P}$ be a label 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 $j \not\supseteq p$**: Then $\sim_j$ is the total relation on $(V^*):p$ and so

$$l^* \text{.get } (s_1 \cdots s_n) \sim_j l^* \text{.get } (s'_1 \cdots s'_n)$$

trivially.

**Case $j \supseteq p$**: From

$$j \text{ observes } S^l*: (s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)$$

241
we have

\[ m = n \quad s_i \sim_j s'_i \text{ for } i \in \{1, \ldots, n\} \]

Using these facts, calculate as follows

\[
\begin{align*}
l^* \text{.get} (s_1 \cdots s_n) \\
= (l \text{.get } s_1) \cdots (l \text{.get } s_n) & \quad \text{by definition of } l^* \text{.get} \\
\sim_j (l \text{.get } s'_1) \cdots (l \text{.get } s'_n) & \quad \text{by } \text{GetNoLeak for } l \text{ (n times)} \\
= l^* \text{.get} (s'_1 \cdots s'_n) & \quad \text{by definition of } l^* \text{.get}
\end{align*}
\]

and obtain the required equivalence, which completes the proof. \(\square\)

\[
\begin{array}{ccc}
l_1 \in S & \iff T & l_2 \in T \iff V \\
l_1;l_2 \in S & \iff V
\end{array}
\]

6.4.6 Lemma: Let \( l_1 \in S \iff T \) and \( l_2 \in T \iff V \) be secure lenses. Then \((l_1;l_2)\) is a secure lens in \( S \iff V \).

Proof:

\begin{itemize}
\item \textbf{GetPut:} Let \( k \in Q \) and \( s \in S \) and \( v \in V \) with \( v \approx_k (l_1;l_2)\text{.get } s \). By the definition of \((l_1;l_2)\text{.get}\) we have:

\[ v \approx_k l_2 \text{.get} (l_1 \text{.get } s) \]

By \text{GetPut} for \( l_2 \) we also have

\[ l_2 \text{.put } v (l_1 \text{.get } s) \approx_k (l_1 \text{.get } s) \]

Using this fact and \text{GetPut} for \( l_1 \) we obtain

\[ (l_1;l_2)\text{.put } v \ s = l_1 \text{.put} (l_2 \text{.put } v (l_1 \text{.get } s)) \ s \approx_k s \]

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

$$(l_1;l_2).get ((l_1;l_2).put v s)$$

$$= (l_1;l_2).get (l_1.put (l_2.put v (l_1.get s)) s) \quad \text{by definition of } (l_1;l_2).put$$

$$= l_2.get (l_1.get (l_1.put (l_2.put v (l_1.get s)) s)) \quad \text{by definition of } (l_1;l_2).get$$

$$= l_2.get (l_2.put v (l_1.get s)) \quad \text{by PutGet for } l_1$$

$$= v \quad \text{by PutGet for } l_2$$

and obtain the required equality.

**GetNoLeak:** Let $j \in \mathcal{P}$ be a label, and let $s$ and $s'$ strings in $S$ such that $s \sim_j s'$. We calculate as follows:

$$(l_1;l_2).get s$$

$$= l_2.get (l_1.get s) \quad \text{by definition of } (l_1;l_2).get$$

$$\sim_j l_2.get (l_1.get s') \quad \text{by GetNoLeak for } l_1 \text{ and } l_2$$

$$= (l_1;l_2).get s' \quad \text{by definition of } (l_1;l_2).get$$

and obtain the required equivalence.

\[\square\]

\begin{tabular}{lcl}
\hline
$E, F$ well-formed & $(\mathcal{L}(E) \cap \mathcal{L}(F)) = \emptyset$ & $(\mathcal{L}(E)|\mathcal{L}(F))^\dagger$ \\
$q = \sqrt{\{k \mid k \text{ min obs. } E^\dagger\}}$ & $p \supseteq \sqrt{\{k \mid k \text{ observes } E^\dagger F \text{ and } F^\dagger E\}}$ & \\
$\text{filter } E F \in (E:q \mid F:p)^* \iff E^*$ \\
\hline
\end{tabular}

6.4.7 Lemma: Let $E$ and $F$ be well-formed security-annotated regular expressions such that $(\mathcal{L}(E) \cap \mathcal{L}(F)) = \emptyset$ and $(\mathcal{L}(E)|\mathcal{L}(F))^\dagger$. Then (filter $E F$) is a secure lens in $(E:q \mid F:p)^* \iff E^*$ where $q = \sqrt{\{k \mid k \text{ min obs. } E^\dagger\}}$ and $p \supseteq \sqrt{\{k \mid k \text{ observes } E^\dagger F \text{ and } F^\dagger E\}}$.

Proof:

**GetPut:** Let $k \in Q$ and $(s_1 \cdots s_m) \in (E:q \mid F:p)^*$ and $(v_1 \cdots v_n) \in E^*$ with

$$(v_1 \cdots v_n) \approx_k ((\text{filter } E F).get (s_1 \cdots s_m))$$

We analyze two cases.
Case $k \not\models q$: From the assumption of the case we have that $\text{hide}_k(\cdot)$ maps elements of $E$ to $\epsilon$. Let $(f_1, \ldots, f_o)$ be the elements of $F$ in $(s_1 \cdots s_m)$. By the definition of unfilter, the elements of $F$ in unfilter $F$ $(v_1 \cdots v_n)$ $(s_1 \cdots s_m)$ are also $(f_1, \ldots, f_o)$. Using these facts, we calculate as follows

\[
\text{hide}_k((\text{filter } E F).\text{put} \ (v_1 \cdots v_n) \ (s_1 \cdots s_m))
\]

\[
= \text{hide}_k(\text{unfilter } F \ (v_1 \cdots v_n) \ (s_1 \cdots s_m)) \quad \text{by definition of } (\text{filter } E F).\text{put}
\]

\[
= \text{hide}_k(f_1 \cdots f_o) \quad \text{as } \text{hide}_k(\cdot) \text{ maps } E \text{ to } \epsilon
\]

\[
= \text{hide}_k(s_1, \ldots, s_m) \quad \text{also as } \text{hide}_k(\cdot) \text{ maps } E \text{ to } \epsilon
\]

and obtain the required equivalence.

Case $k \models q$: Let $(t_1, \ldots, t_o)$ be the elements of $(E : q)$ in $(s_1 \cdots s_m)$. From

\[
k \text{ observes } E^* \quad (v_1 \cdots v_n) \approx_k (\text{filter } E F).\text{get} \ (s_1 \cdots s_m)
\]

we have:

\[
n = o \quad v_i \approx_k \ E^* \ t_i \text{ for } i \in \{1, \ldots, n\}
\]

Using these facts, we calculate as follows:

\[
\text{hide}_k((\text{filter } E F).\text{put} \ (v_1 \cdots v_n) \ (s_1 \cdots s_m))
\]

\[
= \text{hide}_k(\text{unfilter } F \ (v_1 \cdots v_n) \ (s_1 \cdots s_m)) \quad \text{by definition of } (\text{filter } E F).\text{put}
\]

\[
= \text{hide}_k(s'_1 \cdots s'_m) \quad \text{by definition of unfilter}
\]

where $s'_i = \begin{cases} v_j & \text{if } s_i \text{ is the } j^{th} \text{ element of } E \text{ in } (s_1 \cdots s_m) \\ s_i & \text{if } s_i \in F \end{cases}$

\[
= \text{hide}_k(s_1 \cdots s_m) \quad \text{as } \text{hide}_k(v_i) = \text{hide}_k(t_i) \quad \text{for } i \in \{1, \ldots, o\}
\]

and obtain the required equivalence.

**PutGet:** Let $(v_1 \cdots v_n) \in E^*$ and $(s_1 \cdots s_m) \in (E : q | F : p)^*$. Observe that string_filter preserves the elements of $E$ in its argument and that string_unfilter propagates all of the elements
in its first argument, which all belong to $E$. Using these facts, we calculate as follows

\[
(f \text{ilter } E \ F).\text{get } ((f \text{ilter } E \ F).\text{put } (v_1 \cdots v_n) \ (s_1 \cdots s_m))
\]

\[
= (f \text{ilter } E \ F).\text{get } (f \text{ilter } F \ (v_1 \cdots v_n) \ (s_1 \cdots s_m)) \quad \text{by definition of } (f \text{ilter } E \ F).\text{put}
\]

\[
= \text{string}_{f \text{ilter }} E \ (f \text{ilter } \text{unfilter } F \ (v_1 \cdots v_n) \ (s_1 \cdots s_m)) \quad \text{by definition of } (f \text{ilter } E \ F).\text{get}
\]

\[
= (v_1 \cdots v_n) \quad \text{by definition string}_{f \text{ilter}} \text{ and } \text{string}_{\text{unfilter}}
\]

and obtain the required equality.

**GetNoLeak**: Let $j \in \mathcal{P}$ and \((s_1 \cdots s_m), (s'_1 \cdots s'_n) \in (E \ q | F \ p)^*\) with \((s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)\).

We analyze two cases

**Case** $j \not\supset p$: Then \(\text{hide}_j(\cdot)\) maps elements of $F$ to $\epsilon$. Let \((e_1, \ldots, e_i)\) and \((e'_1, \ldots, e'_j)\) be the elements of \((s_1 \cdots s_m)\) and \((s'_1 \cdots s'_n)\) that belong to $E$. From \((s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)\) we calculate as follows

\[
\text{hide}_j(s_1 \cdots s_m) = \text{hide}_j(s'_1 \cdots s'_n)
\]

i.e., \(\text{hide}_j(e'_1 \cdots e'_j) = \text{hide}_j(e'_1 \cdots e'_j)\)

as \(\text{hide}_j(\cdot)\) maps elements of $F$ to $\epsilon$

i.e., \(\text{hide}_j((f \text{ilter } E \ F).\text{get } (s_1 \cdots s_m)) = \text{hide}_j((f \text{ilter } E \ F).\text{get } (s'_1 \cdots s'_n))\)

by definition of \((f \text{ilter } E \ F).\text{get}\)

i.e., \((f \text{ilter } E \ F).\text{get } (s_1 \cdots s_m) \sim_j (f \text{ilter } E \ F).\text{get } (s'_1 \cdots s'_n)\)

by definition of \(\sim_j\)

and obtain the required equivalence.

**Case** $j \supset p$: Let \((e_{11} \cdots e_{1i}), \ldots, (e_{o1} \cdots e_{oj})\) and \((e'_{11} \cdots e'_{1q}), \ldots, (e_{p1} \cdots e_{pr})\) be the contiguous—i.e, those not separated by an $F$—elements of $E$ in \((s_1 \cdots s_m)\) and \((s'_1 \cdots s'_n)\). From \((s_1 \cdots s_m) \sim_j (s'_1 \cdots s'_n)\) and $j$ observes $E \cdot \! F$ and $F \cdot \! E$ we have $o = p$ and

\[
\text{hide}_j(e_{11} \cdots e_{1i}) = \text{hide}_j(e'_{11} \cdots e'_{1q}) \quad \ldots \quad \text{hide}_j(e_{o1} \cdots e_{oj}) = \text{hide}_j(e'_{p1} \cdots e'_{pr})
\]

245
Using these facts, we calculate as follows

\[ \text{hide}_j(e_{11} \cdots e_{1i}) \cdots \text{hide}_j(e_{o1} \cdots e_{oj}) = \text{hide}_j(e'_{11} \cdots e'_{1q}) \cdots \text{hide}_j(e'_{p1} \cdots e'_{pr}) \]

i.e., \[ \text{hide}_j((e_{11} \cdots e_{1i}) \cdots (e_{o1} \cdots e_{oj})) = \text{hide}_j((e'_{11} \cdots e'_{1q}) \cdots (e'_{p1} \cdots e'_{pr})) \]

by definition \( \text{hide}_j(\cdot) \)

i.e., \[ \text{hide}_j(\text{string\_filter \( E\) \( (s_1 \cdots s_m)\)}) = \text{hide}_j(\text{string\_filter \( E\) \( (s'_1 \cdots s'_n)\)}) \]

by definition of filter

i.e., \[ \text{hide}_j((\text{filter \( E\) \( F\)\.get} \( (s_1 \cdots s_m)\)) = \text{hide}_j((\text{filter \( E\) \( F\)\.get} \( (s'_1 \cdots s'_n)\)) \]

by definition of \( (\text{filter \( E\) \( F\)\.get} \( (s_1 \cdots s_m)\)) \sim_j (\text{filter \( E\) \( F\)\.get} \( (s'_1 \cdots s'_n)\)) \]

by definition of \( \sim_j \)

and obtain the required equivalence, which completes the proof.

\[ \begin{array}{c}
  l \in S \leftrightarrow V \quad q \in Q \quad p \in P \\
  l \in (S:q) \leftrightarrow (V:p)
\end{array} \]

6.4.8 Lemma: Let \( l \in S \leftrightarrow V \) be a secure lens and let \( q \in Q \) and \( p \in P \) be labels. Then \( l \) is also a secure lens in \( (S:q) \leftrightarrow (V:p) \).

Proof:

\begin{itemize}
  \item GetPut: Let \( k \in Q \) and \( s \in (C:q) \) and \( s \in (V:p) \) with \( v \approx_k l\.get \ s \). We analyze two cases:

    \begin{itemize}
      \item Case \( k \not\sqsubseteq q \): Then \( \approx_k \) is the total relation on \( (C:q) \) and so and so \( l\.put \ v \ s \approx_k s \) trivially.
      \item Case \( k \sqsubseteq q \): By the definition of the semantics of \( V:p \) we have \( v \approx_k l\.get \ s \). By GetPut for \( l \) we have \( l\.put \ v \ s \approx_k \hat{s} \). Finally, by the definition of the semantics of \( S:q \) we have \( l\.put \ v \ s \approx_k^{(S:q)} \hat{s} \), as required.
    \end{itemize}

  \item PutGet: Follows directly from PutGet for \( l \).

  \item GetNoLeak: Let \( j \in P \) and \( s, s' \in (S:q) \) with \( s \sim_j s' \). We analyze two cases:

    \begin{itemize}
      \item Case \( j \not\sqsubseteq p \): Then \( \sim_j \) is the total relation on \( V:p \) and so and so \( l\.get \ s \approx_j l\.get \ s' \) trivially.
    \end{itemize}
\end{itemize}
Case \( j \equiv p \): By the definition of the semantics of \( S: q \) we have \( s \sim_{S} s' \). Then, by GetNoLeak for \( l \) we have \( l.\text{get} s \sim_{k}^* l.\text{get} s' \). By the definition of the semantics of \( V: p \) we also have \( l.\text{get} s \sim_{j} l.\text{get} s' \) as required, which completes the proof. \( \Box \)

6.5.1 Lemma: Let \( E \) be a well-formed security-annotated regular expression. Then \( (\text{copy } E) \) is a secure lens in \( E \leftarrow E \).

Proof:

\( \triangleright \text{GetPut} \): Let \((j, k) \in C \) and \( s \in E \) and \( v \in E \) with \( (\text{copy } E).\text{safe} (j, k) v s \). By the definition of \( (\text{copy } E).\text{safe} \) we have \( v \equiv_{k} s \). Using this fact, we calculate as follows

\[
(\text{copy } E).\text{put} v s
= v
\]

by the definition of \( (\text{copy } E).\text{put} \)

\[
\approx_{k} s
\]

and obtain the required equivalence.

\( \triangleright \text{PutNoLeak} \): Let \((j, k) \in C \) and \( s, s' \in E \) and \( v, v' \in E \) with \( s \sim_{j} s' \) and \( v \sim_{j} v' \) and \( (\text{copy } E).\text{safe} (j, k) v s \) and \( (\text{copy } E).\text{safe} (j, k) v' s' \). We calculate as follows

\[
(\text{copy } E).\text{put} v s
= v
\]

by definition of \( (\text{copy } E).\text{put} \)

\[
\sim_{j} v'
\]

by assumption

\[
= (\text{copy } E).\text{put} v' s'
\]

by definition of \( (\text{copy } E).\text{put} \)

and obtain the required equivalence.

\( \triangleright \text{SafeNoLeak} \): Let \((j, k) \in C \) and \( s, s' \in E \) and \( v, v' \in E \) with \( v \sim_{j} v' \) and \( s \sim_{j} s' \). Then as \( \sim_{j} \subseteq \equiv_{k} \) we have \( s \equiv_{k} s' \) and \( v \equiv_{k} v' \). Using these equivalences, we calculate as follows

\[
(\text{copy } E).\text{safe} (j, k) v s
\]

\[
\iff v \equiv_{k} s
\]

by definition of \( (\text{copy } E).\text{safe} \)

\[
\iff v' \equiv_{k} s'
\]

by symmetry and transitivity of \( \equiv_{k} \)

\[
\iff (\text{copy } E).\text{safe} (j, k) v' s'
\]

by definition of \( (\text{copy } E).\text{safe} \)

and obtain the required equality, which completes the proof. \( \Box \)
6.5.2 Lemma: Let $E$ and $F$ be well-formed security-annotated regular expressions such that $|\mathcal{L}(F)| = 1$. Then $(\text{const } E \ F)$ is a secure lens in $E \iff F$.

Proof:

- **GetPut**: Identical to previous proof of \textsc{GetPut} for the secure lens version of \textit{const}.

- **PutNoLeak**: Let $(j, k) \in C$ and $s, s' \in E$ and $v, v' \in F$ with $s \sim_j s'$ and $v \sim_j v'$ and $(\text{const } E \ F).\text{safe} (j, k) \ v \ s$ and $(\text{const } E \ F).\text{safe} (j, k) \ v' \ s'$. We calculate as follows

\[
(\text{const } E \ F).\text{put} \ v \ s
\]

\[
= s \quad \text{by definition of } (\text{const } E \ F).\text{put}
\]

\[
\sim_j s'
\]

\[
= (\text{const } E \ F).\text{put} \ v' \ s' \quad \text{by definition of } (\text{const } E \ F).\text{put}
\]

and obtain the required equivalence.

- **SafeNoLeak**: Let $s, s' \in E$ and $v, v' \in E$ with $s \sim_j s'$ and $v \sim_j v'$. We have

\[
(\text{const } E \ F).\text{safe} (j, k) \ v \ s = \top = (\text{const } E \ F).\text{safe} (j, k) \ v' \ s'
\]

immediately, which completes the proof. \qed

6.5.3 Lemma: Let $l_1 \in S_1 \iff V_1$ and $l_2 \in S_2 \iff V_2$ such that $S_1:1 S_2$ and $V_1:1 V_2$. Then $(l_1 \cdot l_2)$ is a secure lens in $(S_1:1 S_2) \iff (V_1:1 V_2);p$ where $p = \bigvee \{ k \mid k \text{ min obs. } S_1:1 S_2 \}$.

Proof:
**GetPut**: Let \((j, k) \in C\) be a clearance, \((v_1, v_2) \in (V_1, V_2): p\) a string, and \((s_1, s_2) \in (S_1, S_2)\) a string such that \((l_1, l_2).\text{safe} (j, k) (v_1, v_2) (s_1, s_2)\). By the definition of \((l_1, l_2).\text{safe}\) we have:

\[
l_1.\text{safe} (j, k) v_1 s_1 \quad l_2.\text{safe} (j, k) v_2 s_2
\]

Using these facts, we calculate as follows

\[
(l_1, l_2).\text{put} (v_1, v_2) (s_1, s_2)
= (l_1.\text{put} v_1 s_1) \cdot (l_2.\text{put} v_2 s_2)
\]

by definition of \((l_1, l_2).\text{put}\)

\[
\approx_k s_1 s_2
\]

by GetPut for \(l_1\) and \(l_2\)

and obtain the required equivalence.

**PutNoLeak**: Let \((j, k) \in C\) and \((s_1, s_2), (s'_1, s'_2) \in (S_1, S_2)\) and \((v_1, v_2), (v'_1, v'_2) \in (V_1, V_2): p\) with:

\[
(s_1, s_2) \sim_j (s'_1, s'_2) \quad (v_1, v_2) \sim_j (v'_1, v'_2)
(l_1, l_2).\text{safe} (j, k) (v_1, v_2) (s_1, s_2)
(l_1, l_2).\text{safe} (j, k) (v'_1, v'_2) (s'_1, s'_2)
\]

From the definition of \((l_1, l_2).\text{safe}\), we have that \(j\) observes \(S_1, S_2\) and \(V_1, V_2\) and hence:

\[
s_1 \sim_j^{S_1} s'_1 \\ v_1 \sim_j^{V_1} v'_1
s_2 \sim_j^{S_2} s'_2 \\ v_2 \sim_j^{V_2} v'_2
\]

Also by the definition of \((l_1, l_2).\text{safe}\) we have

\[
l_1.\text{safe} (j, k) v_1 s_1 \quad l_2.\text{safe} (j, k) v_2 s_2
l_1.\text{safe} (j, k) v_1 s'_1 \quad l_2.\text{safe} (j, k) v_2 s'_2
\]

Using all these facts, we calculate as follows

\[
(l_1, l_2).\text{put} (v_1, v_2) (s_1, s_2)
= (l_1.\text{put} v_1 s_1) \cdot (l_2.\text{put} v_2 s'_2)
\]

by definition of \((l_1, l_2).\text{put}\)

\[
\sim_j (l_1.\text{put} v'_1 s'_1) \cdot (l_2.\text{put} v'_2 s'_2)
\]

by PutNoLeak for \(l_1\) and \(l_2\)

\[
= (l_1, l_2).\text{put} (v'_1, v'_2) (s'_1, s'_2)
\]

by definition of \((l_1, l_2).\text{put}\)

and obtain the required equivalence.

**SafeNoLeak**: Let \((j, k) \in C\) and \((s_1, s_2), (s'_1, s'_2) \in (S_1, S_2)\) and \((v_1, v_2), (v'_1, v'_2) \in (V_1, V_2): p\) with \((s_1, s_2) \sim_j (s'_1, s'_2)\) and \((v_1, v_2) \sim_j (v'_1, v'_2)\). 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{align*}
s_1 & \sim_j S_1 \ s'_1 \\
v_1 & \sim_j V_1 \ v'_1
\end{align*}
\[
\begin{align*}
s_2 & \sim_j S_2 \ s'_2 \\
v_2 & \sim_j V_2 \ v'_2
\end{align*}
\]

Using these facts, we calculate as follows

\[
(l_1 \cdot l_2).safe (j, k) (v_1 \cdot v_2) (s_1 \cdot s_2)
\]

\[
\iff (l_1.safe (j, k) v_1 s_1) \land (l_2.safe (j, k) v_2 s_2) \quad \text{by definition of } (l_1 \cdot l_2).safe
\]

\[
\iff (l_1.safe (j, k) v'_1 s'_1) \land (l_2.safe (j, k) v'_2 s'_2) \quad \text{by SafeNoLeak for } l_1 \text{ and } l_2
\]

\[
\iff (l_1 \cdot l_2).safe (j, k) (v'_1 \cdot v'_2) (s'_1 \cdot s'_2)
\]

and obtain the required equality.

Case \(j\) does not observe \( S_1 \cdot s_2 \) and \( V_1 \cdot V_2 \): Then

\[
(l_1 \cdot l_2).safe (j, k) (v_1 \cdot v_2) (s_1 \cdot s_2) = \bot = (l_1 \cdot l_2).safe (j, k) (v'_1 \cdot v'_2) (s'_1 \cdot s'_2)
\]

immediately, which completes the case and the proof. \(\square\)

\[
\begin{array}{c}
(S_1 \cap S_2) = \emptyset \\
l_1 \in S_1 \iff V_1 \\
l_2 \in S_2 \iff V_2 \\
p = \bigvee \{k \mid k \min obs. \ (S_1 \cap S_2) = \emptyset\}
\end{array}
\]

\[
\begin{array}{c}
l_1 \mid l_2 \in (S_1 \mid S_2) \iff (V_1 \mid V_2) : p
\end{array}
\]

**6.5.4 Lemma:** Let \( l_1 \in S_1 \iff V_1 \) and \( l_2 \in S_2 \iff V_2 \) be secure lenses such that \((\mathcal{L}(S_1) \cap \mathcal{L}(S_2)) = \emptyset\). Then \((l_1 \mid l_2)\) is a secure lens in \((S_1 \mid S_2) \iff (V_1 \mid V_2) : p\) where

\[
p = \bigvee \{k \mid k \min obs. \ (S_1 \cap S_2) = \emptyset\}.
\]

**Proof:**

- **GetPut:** Let \((j, k) \in \mathcal{C} \) and \( s \in (S_1 \mid S_2) \) and \( v \in (V_1 \mid V_2) : p \) with \((l_1 \mid l_2).safe (j, k) v s\). We analyze two cases.

  **Case \(s \in S_1\):** From the assumption of the case and \((l_1 \mid l_2).safe (j, k) v s\) we have:

\[
v \in V_1 \quad l_1.safe (j, k) v s
\]

250
Using these facts, we calculate as follows:

\[(l_1 | l_2).put v s\]

\[= l_1.put v s \quad \text{by definition of } (l_1 | l_2).put\]

\[\approx_k s \quad \text{by GetPut for } l_1\]

and obtain the required equivalence.

**Case** \(s \in S_2\): Symmetric to the previous case.

**\(\blacktriangleright\) PutNoLeak:** Let \((j, k) \in C\) and \(s, s' \in (S_1 | S_2)\) and \(v, v' \in (V_1 | V_2): p\) with:

\[s \sim_j s' \quad (l_1 | l_2).safe (j, k) v s\]

\[v \sim_j v' \quad (l_1 | l_2).safe (j, k) v' s'\]

We analyze two cases.

**Case** \(s \in S_1\): From the definition of \((l_1 | l_2).safe\) we have that \(j\) observes \((S_1 \cap S_2) = \emptyset\) and \(V_1 \& V_2\) agree. Using this fact and the assumption that \(s \sim_j s'\) we have that \(s' \in S_1\) and \(s \sim_{j} s'\). Next, from \((l_1 | l_2).safe (j, k) v s\) and \((l_1 | l_2).safe (j, k) v' s'\) we have:

\[v \in V_1 \quad l_1.safe (j, k) v s\]

\[v' \in V_1 \quad l_1.safe (j, k) v' s'\]

Finally, from \(j\) observes \(V_1 \& V_2\) agree we also have \(v \sim^{V_1}_j v'\). Putting all these facts together, we calculate as follows:

\[(l_1 | l_2).put v s\]

\[= l_1.put v s \quad \text{by definition of } (l_1 | l_2).put\]

\[\sim_j l_1.put v s' \quad \text{by PutNoLeak for } l_1\]

\[= (l_1 | l_2).put v s' \quad \text{by definition of } (l_1 | l_2).put\]

and obtain the required equivalence.

**Case** \(s \in S_2\): Symmetric to the previous case.

**\(\blacktriangleright\) SafeNoLeak:** Let \((j, k) \in C\) and \(v, v' \in (V_1 | V_2): p\) with \(s, s' \in (S_1 | S_2)\) and \(v \sim_j v'\) and \(s \sim_j s'\) and \((l_1 | l_2).safe v s\) and \((l_1 | l_2).safe v' s'\). We analyze several cases.
Case \( j \) observes \((S_1 \cap S_2) = \emptyset \) and \( V_1 \& V_2 \) agree and \( s \in S_1 \): From the assumptions of the case we have \( s' \in S_1 \) and \( s \sim_{S_j} s' \). We also have \( v \in V_1 \) if and only if \( v' \in V_1 \) and \( v \sim_{S_j} v' \).

Using these facts we calculate as follows

\[
(l_1 | l_2).safe (j, k) v s
\]

\[\iff v \in V_1 \land l_1.safe (j, k) v s \quad \text{by definition of} \ (l_1 | l_2).safe\]

\[\iff v' \in V_1 \land l_1.safe (j, k) v' s' \quad \text{by SafeNoLeak for} \ l_1\]

\[\iff (l_1 | l_2).safe (j, k) v' s' \quad \text{by definition of} \ (l_1 | l_2).safe\]

Case \( j \) observes \((S_1 \cap S_2) = \emptyset \) and \( V_1 \& V_2 \) agree and \( s \in S_2 \): Symmetric to the previous case.

Case \( j \) does not observe \((S_1 \cap S_2) = \emptyset \) and \( V_1 \& V_2 \) agree: Then

\[
(l_1 | l_2).safe (j, k) v s = \bot = (l_1 | l_2).safe (j, k) v' s'
\]

immediately, which completes the case and the proof.

\[\square\]

6.5.5 Lemma: Let \( l \in S \iff V \) be a secure lens such that \( S^{l*} \) and \( V^{l*} \). Then \( l^* \) is a secure lens in \((S^*) \iff (V^*):p\) where \( p = \bigvee \{ k \mid k \min \text{obs.} S^{l*} \} \).

Proof:

\(\blacktriangleright\) GetPut: Let \( (j, k) \in C \) be a clearance, \((v_1 \cdots v_n) \in (V^*):p\) a string, and \((s_1 \cdots s_m) \in C^* \) a string with \( l^*.safe (j, k) (v_1 \cdots v_n) (s_1 \cdots s_m) \). By the definition of \( l^*.safe \) we have \( n = m \) and \( l.safe (j, k) v_i s_i \) for every \( i \) from 1 to \( n \). Using these facts, we calculate as follows

\[
l^*.put (v_1 \cdots v_n) (s_1 \cdots s_m)
\]

\[
= (l.put v_1 s_1) \cdots (l.put v_n s_n) \quad \text{by definition of} \ l^*.put\]

\[
\approx_k s_1 \cdots s_m \quad \text{by GetPut for} \ l \ (n \text{ times})\]

and obtain the required equivalence.
**PutNoLeak:** Let \((j, k) \in \mathcal{C}\) and \((v_1 \cdots v_m), (v'_1 \cdots v'_n) \in (V^*)\): \(p\) and \((s_1 \cdots s_o), (s'_1 \cdots s'_p) \in S^*\) with
\[
(v_1 \cdots v_m) \sim_j (v'_1 \cdots v'_n) \quad l^*.safe\ (j, k)\ (v_1 \cdots v_m)\ (s_1 \cdots s_o)
\]
\[
(s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p) \quad l^*.safe\ (j, k)\ (v'_1 \cdots v'_n)\ (s_1 \cdots s'_p)
\]
By the definition of \(l^*.safe\) we have that \(j\) observes \(S^*\) and \(V^*\) and so \(m = n = o = p\) and
\[
\text{l.safe}\ (j, k)\ v_i s_i \quad \text{l.safe}\ (j, k)\ v'_i s'_i \quad s_i \sim_j s'_i \quad v_i \sim_j v'_i \text{ for } i \in \{1, \ldots, n\}
\]
Using these facts, we calculate as follows
\[
l^*.put\ (v_1 \cdots v_n)\ (s_1 \cdots s_o)
\]
\[
= (l.put\ v_1\ s_1) \cdots (l.put\ v_n\ s_n) \quad \text{by definition of } l^*.put
\]
\[
\sim_j (l.put\ v'_1\ s'_1) \cdots (l.put\ v'_n\ s'_n) \quad \text{by PutNoLeak for } l\ (n\ times)
\]
\[
= l^*.put\ (v'_1 \cdots v'_n)\ (s'_1 \cdots s'_p) \quad \text{by definition of } l^*.put
\]
and obtain the required equivalence.

**SafeNoLeak:** Let \((j, k) \in \mathcal{C}\) and \((v_1 \cdots v_m), (v'_1 \cdots v'_n) \in (V^*)\): \(p\) and \((s_1 \cdots s_o), (s'_1 \cdots s'_p) \in S^*\) with:
\[
(v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_n) \quad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)
\]
We analyze several cases.

**Case \(j\) observes \(S^*\) and \(V^*\) and \(n = o\):** From the assumptions and the definition of \(l^*.safe\) we have that
\[
m = n = o = p \quad v_i \sim_j v'_i \quad s_i \sim_j s'_i \text{ for } i \in \{1 \ldots n\}
\]
Using these facts we calculate as follows
\[
l^*.safe\ (j, k)\ (v_1 \cdots v_n)\ (s_1 \cdots s_o)
\]
\[
\iff l.safe\ (j, k)\ v_1\ s_1 \land \cdots \land l.safe\ (j, k)\ v_n\ s_n \quad \text{by definition } l^*.safe
\]
\[
\iff l.safe\ (j, k)\ v'_1\ s'_1 \land \cdots \land l.safe\ (j, k)\ v'_n\ s'_n \quad \text{by SafeNoLeak for } l\ (n\ times)
\]
\[
\iff l^*.safe\ (j, k)\ (v'_1 \cdots v'_n)\ (s'_1 \cdots s'_p) \quad \text{by definition } l^*.safe
\]
and obtain the required equality.

**Case \(j\) observes \(S^*\) and \(V^*\) and \(n \neq o\):** From the assumptions of the case and
\[
(v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_n) \quad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)
\]
\[
\rightarrow (v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_n) \quad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)
\]
253
we have $n = m$ and $o = p$. It follows that
\[
\text{l}.\text{safe}(j, k) \left(v_1 \cdots v_n\right) (s_1 \cdots s_o) = \bot = \text{l}.\text{safe}(v'_1 \cdots v'_m) (s'_1 \cdots s'_p)
\]

Case $j$ does not observe $S^{l,*}$ and $V^{l,*}$: Then
\[
\text{l}.\text{safe}(j, k) \left(v_1 \cdots v_n\right) (s_1 \cdots s_o) = \bot = \text{l}.\text{safe}(v'_1 \cdots v'_m) (s'_1 \cdots s'_p)
\]

immediately, which completes the case and the proof.

\[\begin{array}{c}
\text{l}_1 \in S \leftrightarrow T \\
\text{l}_2 \in T \leftrightarrow V \\
\text{l}_1; \text{l}_2 \in S \leftrightarrow V
\end{array}\]

6.5.6 Lemma: Let $l_1 \in S \leftrightarrow T$ and $l_2 \in T \leftrightarrow V$ be secure lenses. Then $(l_1;l_2)$ is a secure lens in $S \leftrightarrow V$.

Proof:

- **GetPut**: Let $(j, k) \in C$ and $s \in S$ and $v \in V$ with $(l_1;l_2).\text{safe}(j, k) v s$. By the definition of $(l_1;l_2).\text{safe}$ we have
  \[
l_1.\text{safe}(l_2.\text{put}(l_1.\text{get}) s)
  \]
  Using this fact, we calculate as follows
  \[
  (l_1;l_2).\text{put} v s
  = l_1.\text{put}(l_2.\text{put}(l_1.\text{get}) s) \quad \text{by definition of } (l_1;l_2).\text{put}
  \approx_k s \quad \text{by } \text{GetPut for } l_1
  \]
  and obtain the required equivalence.

- **PutNoLeak**: Let $(j, k) \in C$ and $v, v' \in V$ and $s, s' \in S$ with $s \sim_j s'$ and $v \sim_j v'$ and $(l_1;l_2).\text{safe} v s$ and $(l_1;l_2).\text{safe} v' s'$. By the definition of $(l_1;l_2).\text{safe}$ we have
  \[
l_1.\text{safe}(l_2.\text{put}(l_1.\text{get}) s) \quad l_1.\text{safe}(l_2.\text{put}(l_1.\text{get}) s') s'
  \]
Using these facts, we calculate as follows

\[(l_1;l_2).\text{put } v \ s = l_1.\text{put } (l_2.\text{put } v \ (l_1.\text{get } s)) \ s \quad \text{by definition of } (l_1;l_2).\text{put}\]

\[\simeq_j l_1.\text{put } (l_2.\text{put } v' \ (l_1.\text{get } s')) \ s' \quad \text{by PutNoLeak for } l_1\]

\[= (l_1;l_2).\text{put } v' \ s' \quad \text{by definition of } (l_1;l_2).\text{put}\]

and obtain the required equivalence.

\[\blacktriangledown \text{SafeNoLeak: Let } (j, k) \in C \text{ and } s, s' \in S \text{ and } v, v' \in V \text{ with } s \sim_j s' \text{ and } v \sim_j v'. \text{ By GetNoLeak for } l_1 \text{ we have that:} \]

\[l_1.\text{get } s \sim_j (l_1.\text{get } s')\]

By PutNoLeak for \(l_2\) we have

\[l_2.\text{put } v \ (l_1.\text{get } s) \sim_j l_2.\text{put } v' \ (l_1.\text{get } s')\]

Using SafeNoLeak for \(l_1\) we obtain the required equality

\[l_1.\text{safe } (l_2.\text{put } v \ (l_1.\text{get } s)) \ s = l_1.\text{safe } (l_2.\text{put } v' \ (l_1.\text{get } s')) \ s'\]

which completes the proof. \(\square\)

\[
\begin{array}{c}
\text{E, F well-formed} \\
\mathcal{L}(E) \cap \mathcal{L}(F) = \emptyset \\
p \equiv \bigvee \{ k \mid k \text{ observes } E \cdot F \text{ and } F \cdot E \} \quad \forall (j, k) \in C. \sim_j L_k \\
\hline \\
\text{filter } E \ F \in (E \mid F;p)^* \iff E^* \\
\end{array}
\]

**6.5.7 Lemma:** Let \(E\) and \(F\) be well-formed security-annotated regular expressions such that \(\mathcal{L}(E) \cap \mathcal{L}(F) = \emptyset\) and \((\mathcal{L}(E) \mid \mathcal{L}(F))^*\). Then \((\text{filter } E \ F)\) is a secure lens in \((E \mid F;p)^* \iff E^*\) where \(p \equiv \bigvee \{ k \mid k \text{ observes } E \cdot F \text{ and } F \cdot E \}\).

**Proof:**

\[\blacktriangledown \text{GetPut: Let } (j, k) \in C \text{ and } (v_1 \cdots v_n) \in E^* \text{ and } (s_1 \cdots s_m) \in (E \mid F;p)^* \text{ with:} \]

\[(\text{filter } E \ F).\text{safe } (j, k) \ (v_1 \cdots v_n) \ (s_1 \cdots s_m)\]

255
Let \((t_1, \ldots, t_o)\) be the elements of \(E\) in \((s_1 \cdots s_m)\). By the definition of \((\text{filter } E \cdot E).safe\) we have that
\[
k \text{ observes } E^* \quad (v_1 \cdots v_n) \approx_k \text{ string}_E (s_1 \cdots s_m)
\]
and hence:
\[
n = o \quad v_i \approx_k t_i \text{ for } i \in \{1 \ldots n\}
\]

Using these facts, we calculate as follows
\[
hide_k((\text{filter } E \cdot E).put (v_1 \cdots v_n) (s_1 \cdots s_m))
\]
\[
= hide_k(\text{unfilter } F (v_1 \cdots v_n) (s_1 \cdots s_m)) \quad \text{by definition of (filter } E \cdot E).put
\]
\[
= hide_k(s'_1 \cdots s'_m) \quad \text{by definition of unfilter}
\]
where \(s'_i =\)
\[
\begin{cases} v_j & \text{if } s_i \text{ is the } j^{th} \text{ element of } \in E \text{ in } (s_1 \cdots s_m) \\ s_i & \text{if } s_i \in F \end{cases}
\]
\[
= hide_k(s_1 \cdots s_m) \quad \text{as } v_i \approx_k t_i \text{ for } i \in \{1, \ldots, o\}
\]
i.e. \((\text{filter } E \cdot F).put (v_1 \cdots v_n) (s_1 \cdots s_m) \approx_k (s_1 \cdots s_m)
and obtain the required equivalence.

\textbf{PutNoLeak: } Let \((j, k) \in C\) and \((v_1 \cdots v_m), (v'_1 \cdots v'_n) \in E^*\) and \((s_1 \cdots s_o), (s'_1 \cdots s'_p) \in (E \mid F : p) : *\) with:
\[
(v_1 \cdots v_m) \sim_j (v'_1 \cdots v'_n) \quad (\text{filter } E \cdot F).safe (j, k) (v_1 \cdots v_m) (s_1 \cdots s_o)
\]
\[
(s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p) \quad (\text{filter } E \cdot F).safe (j, k) (v'_1 \cdots v'_n) (s'_1 \cdots s'_p)
\]

Let \((t_1, \ldots, t_w)\) and \((t'_1, \ldots, t'_x)\) be the elements of \(E\) in \((s_1 \cdots s_o)\) and \((s'_1 \cdots s'_p)\) respectively. As \(j\) observes \(E^*\) we have that
\[
m = n \text{ and } v_i \sim_j v'_i \text{ for } i \in \{1, \ldots, m\}
\]

Let \((f_{11} \cdots f_{1i}), \ldots, (f_{y1} \cdots f_{yj}), \ldots, (f'_{11} \cdots f'_{1k}), \ldots, (f'_{z1} \cdots f'_{zt})\) be the contiguous elements of \(F\) in \((s_1 \cdots s_o)\) and \((s'_1 \cdots s'_p)\). As \((s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)\) and \(j\) observes \(E \cdot F\) and \(F \cdot E\) we have that \(y = z\) and
\[
(f_{11} \cdots f_{1i}) \sim_j (f'_{11} \cdots f'_{1k}) \land \ldots \land (f_{y1} \cdots f_{yj}) \sim_j (f'_{z1} \cdots f'_{zt})
\]

256
Using these facts, we calculate as follows

\[
\text{hide}_j((\text{filter } E \cdot F).\text{put} \ (v_1 \cdots v_m) \ (s_1 \cdots s_o)) = \text{hide}_j(\text{unfilter} \ (v_1 \cdots v_m) \ (s_1 \cdots s_o))
\]

by definition of \((\text{filter } E \cdot F).\text{put}\)

\[
= \text{hide}_j(s''_1 \cdots s''_{o+(m-w)})
\]

by definition of unfilter

where \(s'_i = \begin{cases} 
  v_{i-w} & \text{if } i > o \\
  v_j & \text{if } i \leq o \text{ and } s_i \text{ is the } j\text{th element of } \in E \text{ in } (s_1 \cdots s_o) \\
  s_i & \text{if } s_i \in F
\end{cases}
\]

\[
= \text{hide}_j(s''_1 \cdots s''_{p+(n-x)})
\]

by above facts

where \(s''_i = \begin{cases} 
  v'_{i-x} & \text{if } i > p \\
  v'_{i} & \text{if } i \leq p \text{ and } s'_i \text{ is the } j\text{th element of } \in E \text{ in } (s'_1 \cdots s'_p) \\
  s'_i & \text{if } s'_i \in F
\end{cases}
\]

\[
= \text{hide}_j(\text{unfilter} \ (v'_1 \cdots v'_n) \ (s'_1 \cdots s'_p))
\]

by definition of unfilter

\[
= \text{hide}_j((\text{filter } E \cdot F).\text{put} \ (v'_1 \cdots v'_n) \ (s'_1 \cdots s'_p))
\]

by definition of \((\text{filter } E \cdot F).\text{put}\)

\[
i.e., \ (\text{filter } E \cdot F).\text{put} \ (v_1 \cdots v_n) \ (s_1 \cdots s_o) \sim_j (\text{filter } E \cdot F).\text{put} \ (v'_1 \cdots v'_n) \ (s'_1 \cdots s'_p)
\]

and obtain the required equivalence.

\textbf{SafeNoLeak:} Let \((j, k) \in C\) and \((v_1 \cdots v_n), (v'_1 \cdots v'_m) \in E^*\) and \((s_1 \cdots s_o), (s'_1 \cdots s'_p) \in (E \mid F:p)^*\) with:

\[
(v_1 \cdots v_n) \sim_j (v'_1 \cdots v'_m) \quad (s_1 \cdots s_o) \sim_j (s'_1 \cdots s'_p)
\]

We analyze two cases.

\textbf{Case } j \text{ observes } E^1 \cdot F \text{ and } F^1 \cdot E \text{ and } j \text{ and } k \text{ observe } E^*:\ Let \((t_1, \ldots, t_w)\) and \((t'_1, \ldots, t'_x)\) be the elements of \(E\) in \((s_1 \cdots s_o)\) and \((s'_1 \cdots s'_p)\). From assumptions of the case we have that \(w = x\) and also

\[
\text{hide}_j(t_i) = \text{hide}_j(t'_i) \text{ for } i \in \{1, \ldots, w\}
\]

257
As $\sim_j \subseteq \approx_k$ we also have:

$$\text{hide}_k(t_i) = \text{hide}_k(t'_i) \text{ for } i \in \{1, \ldots, w\}$$

Using this fact, we calculate as follows

$$(\text{filter } E \ F).\text{safe} (j, k) (v_1 \ldots v_m) (s_1 \cdots s_o)$$

$$\iff (v_1 \ldots v_n) \approx_k \text{string}_k E (s_1 \cdots s_o) \quad \text{by definition of } (\text{filter } E \ F).\text{safe}$$

$$\iff \text{hide}_k(v_1 \cdots v_m) = \text{hide}_k(t_1 \cdots t_w) \quad \text{by definition of string}_k$$

$$\iff \text{hide}_k(v'_1 \cdots v'_n) = \text{hide}_k(t'_1 \cdots t'_w) \quad \text{by symmetry and transitivity of } =$$

$$\iff (v_1 \ldots v'_n) \approx_k \text{string}_k E (s'_1 \cdots s'_{p'}) \quad \text{by definition of filter}$$

$$\iff (\text{filter } E \ F).\text{safe} (j, k) (v'_1 \ldots v'_n) (s'_1 \cdots s'_{p'}) \quad \text{by definition of } (\text{filter } E \ F).\text{safe}$$

and obtain the required equivalence.

**Case $j$ does not observe $E \cdot F$ and $F \cdot E$ or $j$ or $k$ do not observe $E^{\dagger}$:** Then

$$(\text{filter } E \ F).\text{safe} (j, k) (v_1 \ldots v_m) (s_1 \cdots s_o)$$

$$= \bot$$

$$= (\text{filter } E \ F).\text{safe} (j, k) (v'_1 \ldots v'_n) (s'_1 \cdots s'_{p'})$$

immediately, which completes the proof. \qed