Chapter 4

Prototype Library for Edit Lenses

4.1 Introduction

Having developed the theory of lenses and instantiating the framework with a syntax, we now give an exposition on preliminary efforts to instantiate the syntax as a concrete program. Our work on a prototype has two main purposes. The edit lens framework is predicated on a relatively abstract, algebraic data model, whereas long-term data storage on computers typically employs a fairly low-level model based on strings. When only the data is important, these two realms are typically connected by defining a parser that processes strings and produces a more structured representation, as well as a formatter that produces a string representation of a given structure. For edit lenses, however, not only the data is important; one also wants access to the edits made to the data. So the primary goal is to investigate what extensions are needed to describe the connection between edits to strings and edits to structured data. A secondary goal is to validate that the fundamental edit lens design is complete; producing a few example transformations gives an opportunity for any unforeseen infelicities to rear their head. In the pursuit of these goals, we discuss two artifacts: first, a core library which closely models the edit lens theory given in Chapter 3; and second, a demonstration program that synchronizes two simple, text-based databases according to a predetermined lens. The latter task involves building a text-editing GUI, connecting the lens to the GUI, and validating and extracting edits from user actions, tasks that fall outside the realm of the existing edit lens theory.

We have chosen to implement our demo in Haskell, a language which encourages high abstraction levels, supports rapid prototyping, and has good library support. Because one of our primary goals was experimentation, we wanted to retain a lightweight approach throughout; in particular we chose not to begin with a mechanization of the theory in a dependently-typed language. (In retrospect, while it still seems worthwhile to have avoided reproving all of our results in Coq from an experimentation point of view, it is not clear that avoiding dependent types entirely was beneficial. For example, the way we used Haskell's typeclass mechanism to model modules would

really have benefited from dependent types, as we often found ourselves wishing for the ability to define new types for different choices of *init* value.) We also investigated extending Boomerang [8], an existing asymmetric, state-based string lens implementation. Boomerang is very complete, and consequently would have required many tangential coding efforts; to avoid distractions, we chose to take a less feature-complete route. However, we retained Boomerang's choice of string-based data model since, as discussed above, this closely matches real-world scenarios.

Our primary challenge, which we will discuss in detail below, can be broadly described as parsing. With edit lenses, there are always two domains of discourse: the collection of repositories and the collection of edits. Repositories store ordinary data, and the problem of connecting strings with structured data is well-studied under the umbrella of parsing. (Turning structured data into a string—often called serialization—is typically a significantly simpler task.) However, standard parsing techniques—even incremental techniques purportedly designed for making it easy to maintain a correct parse tree in the presence of ongoing updates—do not adequately describe the connection between string modifications and edits in the sense described in Chapter 3. One could avoid the situation entirely by designing a structured editor. Historically, though, structured editors have failed to take—perhaps because their strictures are too confining for day-to-day editing tasks—so we chose to avoid this route. We have proposed a few heuristics that seem to behave acceptably in a number of standard cases; however, they are relatively special-purpose (tailored to the file format under consideration here) and do not adequately reflect all user actions as analogous edits. This seems like a promising area for future efforts.

4.2 Usage Example and Functionality

In order to ground the discussion, we give here a quick overview of the capabilities of the program we have built. When started, the program presents a GUI containing two text-editing panes in which the user can freely type. The two texts in the panes are connected by a lens, so that when the text in one pane has been suitably modified, the text in the other pane spontaneously changes to maintain synchrony. The particular lens we will demonstrate below is a variant of the lens in Figure 1.5, but instead of connecting teachers, salaries, and room assignments, we will connect composers, birth years, and birth countries. In one repository, we will have a list of newline-terminated records, where each record has a composer's name and birth year separated by a comma. In the other, each record has a composer's name and birth country separated by a semicolon. Figure 4.1a gives a pair of example synchronized repositories entered into our program's text panes. In the abstract notation of Chapter 3, the lens connecting the two panes might be written as $(id \otimes disconnect)^\star$. The concrete lens used here must include a bit more information—for example, instructions to change the comma separating parts of the record into a semicolon, or a check that dates consist of exactly four digits—but we will skip discussing these surface

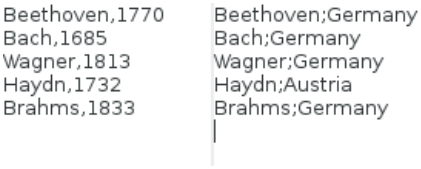
syntax issues for now. In any case, the typical chain of events begins with the user making an edit to one of the repositories. This user action is processed to produce alignment information between the old and new repositories, which is handed to the underlying edit lens' $\Rrightarrow$ or $\Lleftarrow$ function; the computed edit is then used to produce a "user" action which is automatically applied to the other repository.

The remainder of Figure 4.1 demonstrates how the text panes would evolve under a few plausible edits to the repositories. Part b shows what happens when the user adds an extra line to the text pane on the right. As the right-hand repository now has an extra record for Mozart, the lens produces an insertion that adds a record for Mozart to the repository on the left, using a default birth year. Since the insertion is inferred by watching the typing commands performed by the user, the alignment for insertions of this kind can be exact: even if the user were to duplicate a record from elsewhere in the database, no confusion would arise, and a new record would appear with default data in the correct location in the other repository. Similarly, when the user deletes a line—in this case, the record for Haydn—on the right, the program maintains synchrony by deleting the record for Haydn from the left pane, as shown in part c. The left pane may also be edited by the user, as demonstrated in part d, where Haydn has been re-inserted into the repository on the left, resulting in a computed insertion containing a default country on the right. In addition to the wholesale insertion and deletion of records, the user may modify parts of a single record, and the program will correctly maintain alignment of the edited records. Part e shows that modifications to data that appears in only one repository has no effect on the other, while in part f the user has corrected the name "Hayn" to "Haydn" in the right repository, and this is correctly reflected as an update to the left repository without losing Haydn's birth year. Because we have access to the actions performed by the user, we need not guess about whether the old "Hayn" record should be aligned with the new "Haydn" record in this case. On the other hand, there are certainly edits where the user intention is still not entirely clear; parts g and h show the user constructing a new record on the right that contains bits and pieces of several old records (by performing a deletion that crosses record boundaries). As pictured, our heuristics choose to treat this as a deletion of all the old records that contributed and the insertion of a completely fresh record, so a default birth year is used for the new record in the left repository.

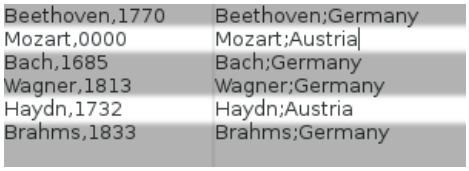
This final example begins to hint at some of the oddities that can arise when attempting to translate between edits to a serialized structure and edits to the abstract structure, which we will discuss in §4.3.

4.3 Implementation Details

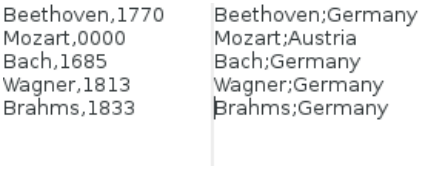
In this section, we will begin with a brief overview of the architecture of the program, with an eye toward guiding the interested reader towards the appropriate part of the full source in §A. (The explanatory material in this part will therefore be quite



(a) An initial pair of databases in two text editing panes.

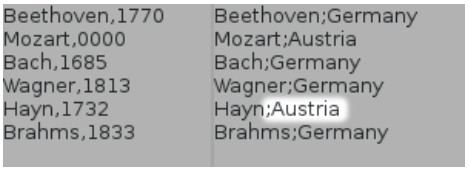


(b) Insertion on the right introduces some default data on the left.

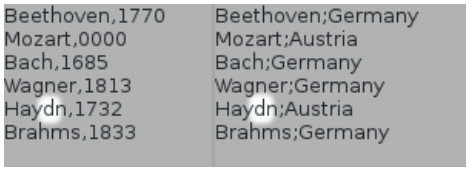


(c) Deleting a row from either side is reflected to the other automatically.

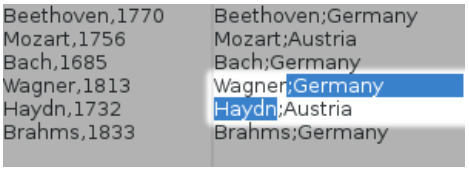
(d) A default country is used for the new row on the right.



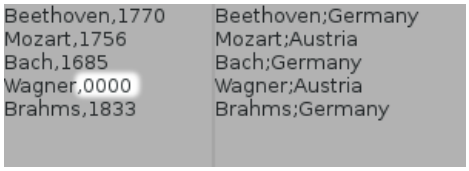
(e) Correcting the country has no effect...



(f) ...but correcting the spelling of either name corrects both.



(g) More bizarre edits, like this deletion that spans records...



(h) ...reset the alignment, but only for the affected region.

Figure 4.1: A demonstration use of the prototype, using the composers lens

brief.) We will then discuss the heuristics used to convert user actions into edits in some detail. After describing the current transformation, we will consider some advantages and disadvantages of this approach as well as possible ways forward with this challenging area of the implementation.

The program code broadly recapitulates the development of edit lenses in Chapter 3 (though behavioral equivalence does not make an appearance, since its primary use is in proofs). A typeclass for monoids already exists in the base libraries of Haskell; an extension of this typeclass to a class for modules, along with facilities for building modules in terms of free monoids, is given in `Data.Module.Class` (§A.20). We chose to implement lenses as a typeclass as well; thus, to create a new lens, one typically declares a new type with a single constructor. One can view this as giving a way to overload the names `dputr` and `dputl`—the way we write $\Rrightarrow$ and $\Lleftarrow$ in the implementation—for many lenses. (Another approach would be to write lenses as a record; using a typeclass makes associating a complement type with the lens slightly less intrusive. Yet another way would be to give an explicit type representing the abstract syntax tree of edit lenses given in this paper; one advantage of typeclasses over algebraic types is that they are open, meaning programmers can add to the collection of lenses without modifying the core library.) This typeclass, along with facilities for defining $\Rrightarrow$ and $\Lleftarrow$ by monoid presentation, is available from `Data.Lens.Edit.Stateful` (§A.17). The `Stateful` part of the name alludes to the existence of a complement; `Data.Lens.Edit.Stateless` (§A.18) contains an experimental typeclass for edit lenses which do not need a complement, and many of the lens types we define below will implement both the `Stateful` and `Stateless` versions of the `Lens` typeclass. These modules together cover the theoretical framework of edit lenses, but give no syntax.

The basic lenses (*id*, composition, $-^{op}$, and *disconnect*) are implemented in the `Data.Lens.Edit.Primitive` (§A.15) module. The *Unit* edit module is implemented in `Data.Module.Primitive` (§A.23). The modules for tensor products and sums are given in `Data.Module.Product` (§A.24) and `Data.Module.Sum` (§A.26) (and are built on the product and sum types from Haskell's base library), and the lenses are given in `Data.Lens.Edit.Product` (§A.16) and `Data.Lens.Edit.Sum` (§A.19). Similarly, the module for list editing is given in `Data.Module.List` (§A.22) (and is built on the list type from Haskell's base library), while the `map` and *partition* lenses are implemented in `Data.Lens.Edit.List` (§A.14). There is no generic container type in Haskell's base library, so this is given in `Data.Container` (§A.8), together with the module for editing list shapes from Example 3.4.6 in `Data.Module.Shape` (§A.25), a module for editing containers in `Data.Module.Container` (§A.21), and the mapping lens in `Data.Lens.Edit.Container` (§A.13).

This completes the recapitulation of edit lenses; the program itself then includes a handful of modules that go beyond the theory. `Data.Module.String` (§A.6) contains the most interesting extended functionality. It includes the `StringModule` typeclass which adds methods for parsing, serializing, and checking validity of repository

strings—all fairly routine operations—as well as a method for translating string edits to `Module` edits. This typeclass is then instantiated for a handful of types, and some utilities are given for defining base modules with a particular *init* value. These utilities are wrapped up in `Data.Lens.Edit.String` (§A.5), which offers some combinators for creating triples containing a value with a `StringModule` instance for each repository and a value with a `Lens` instance connecting the `Module`s associated with those `StringModule`s. Finally, the top-level program lies in `lens-editor.hs` (§A.3), which defines a particular string lens, constructs a GUI with two text panes, allocates a reference cell for the complement, and connects the text panes' editing events to invocations of the appropriate `StringModule` functions. This discussion is summarized in Figure 4.2.

Given this overview, let us discuss in more detail the process of turning the user's edit actions into Module-based edits. The GUI glue code observes user actions and abstracts them into the Edit type:

```
data Edit = Insert Int String | Delete Int Int
```

Here, one should think of Insert n s as being an insertion of string s before index n (with index 0 being the first character), and Delete m n as deleting the range which begins before index m and ends after index n. In the typical case, an insertion contains just a single character corresponding to the key most recently tapped by the user, and the range described by a deletion is just one character wide, containing the character near the cursor when the user tapped the backspace or delete key. Often this granularity is too fine; for example, when modifying a year from 1234 to 1357, the intermediate states 123, 12, 1, 13, and 135 are not (intended to be) valid years. To accommodate this, the GUI waits until the repository string is in a valid format, batching together edit actions.[1] The conglomerated [Edit] list, which represents an action that transforms a valid repository into another valid repository, is then handed off as one of the arguments to the string module's edit parser.

The full type of the edit parser itself is specified in the StringModule typeclass. We sketch the typeclass here, along with its superclass, Module.

```
class (Default (V dX), Monoid dX) ⇒ Module dX where
        type V dX
        apply :: dX → V dX → Maybe (V dX)

class Module (M m) ⇒ StringModule m where
        type M m
        edit :: m → V (M m) → [Edit] → M m
        -- etc.
```

Recall that a module has two types, namely the type of repository values X and the type of edits ∂X; the declaration of the Module typeclass reflects these two types as

[1] One could also consider batching together edit actions indefinitely until the user explicitly requests a run of the lens. Implementing a mode like this would be straightforward.

Data.Container a generic container type, with a typeclass for container shapes

Data.Iso a data type for isomorphisms (primarily used internally)

Data.Lens.Bidirectional a typeclass for bidirectional transformations, used to unify the source- and target-type structure of isomorphisms and the two kinds of lenses

Data.Lens.Edit convenience module: re-exports some submodules

- **.Stateful** `Lens` typeclass for standard edit lenses
- **.Stateless** `Lens` typeclass for edit lenses with no complement
- **.Primitive** base lenses
- **.Product** $\otimes$ lens combinator
- **.Sum** $\oplus$ lens combinator
- **.List** `map` and *partition* lenses
- **.Container** container-mapping lens
- **.String** lens combinators that also construct `StringModule`s

Data.Module convenience module: re-exports some submodules

- **.Class** `Module` typeclass, with instance for free monoid-based modules
- **.Primitive** *Unit* module
- **.Product** $\otimes$ module
- **.Sum** $\oplus$ module
- **.List** $-^\star$ module
- **.Shape** a module for editing the shape of lists construed as containers
- **.Container** container-based module
- **.String** `StringModule` typeclass and instances for the several modules

Main set up and execute a `StringModule`-powered GUI

Figure 4.2: Summary of the module hierarchy in the prototype implementation

V dX and dX, respectively (mnemonic: V dX are the *values* associated with edits dX). Thus the type of the edit parsing function, named edit, may be read: given some string-module specific information, a particular repository value, and a sequence of abstracted user actions, produce an edit for the associated module. This is the key method we wish to implement; and, since it is a typeclass method, we may implement it separately for each module. Note that values of type **String** together with edits of type [Edit] do *not* form a module,[2] hence it does not make sense to impose the same restrictions on this function as we do on edit lens put functions.

For our basic string modules—the ones we use with *id* and *disconnect* lenses—we have a free choice of repository value type and edit type, so there the implementation is easy: we can choose the values to be **String**s and the edits to be values of type [Edit], and let the edit function simply return the string edits it is provided. However, the tensor product and list modules do not share this simplicity. The product module is simpler than the list module, but has most of the important complications, so we focus there.

First we must choose a serialization. The natural choice is to serialize pairs via concatenation: when value a is represented by string sa and value b is represented by string sb, we will represent the pair (a,b) by the string sa ++ sb. For example, the comma-separated composer and date pairs in our example are actually the serialization of nested pairs (**String**,((),**String**)), where the extra () value is a placeholder for the comma separator. The Haskell value ("Beethoven",((),"1770")) would be serialized this way:

Beethoven									()	1770			
B	e	e	t	h	o	v	e	n	,	1	7	7	0

Notionally, there are invisible boundaries in the text before and after the comma, delimiting the portions of the text that correspond to the three parts of the structured value. A natural idea for a heuristic is to try to track these boundaries as text is inserted and deleted and partition the string edits according to where they occur in relation to the chunk boundaries. For a simple example of where this heuristic works well, consider the edit which inserts "Ludwig van " and changes 1770 to 1760:

original	Beethoven	,	1770
Delete 12 13	Beethoven	,	170
Insert 12 "6"	Beethoven	,	1760
Insert 0 "Ludwig van "	Ludwig van Beethoven	,	1760

It seems clear that the right way to split this up is to group together (and re-index) the edit [Delete 2 3, Insert 2 "6"] to the date, give an empty edit [] for the separator, and separate the final insertion [Insert 0 "Ludwig van "] to the composer.

[2]Conglomerated string edits often take the string from a parsing state to a non-parsing one and back. Representing this in our module framework would involve a prefix of the edit to have undefined edit application; but then no suffix can recover from a failed edit.

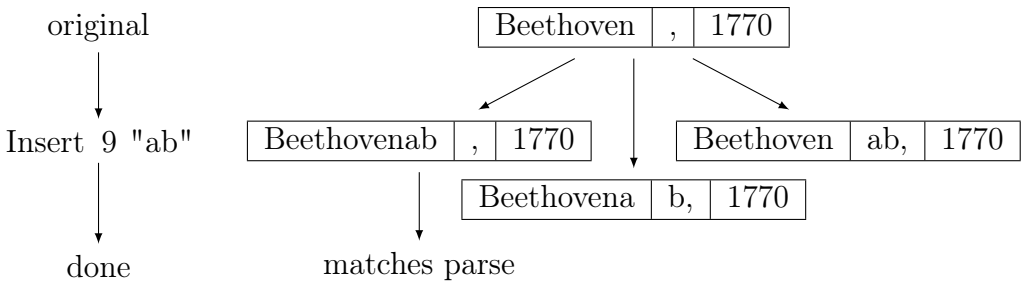
There are several tricky cases to consider, which we discuss in turn below: insertions may happen at the boundary, deletions may span a boundary, deletions may cause candidate boundaries to coincide, and the boundary may jump if an insertion or deletion significantly changes the parse tree. We will tackle the first two problems by nondeterministically guessing the right way to move the boundary at such edits; the third problem by choosing an arbitrary tie-breaking rule; and the fourth problem by having a "nuclear option" backup plan of doing a complete re-parse (losing all alignment information) if all else fails.

Consider the difference between the following two insertions:

original	Beethoven	,	1770
Insert 9 "x"	Beethovenx	,	1770

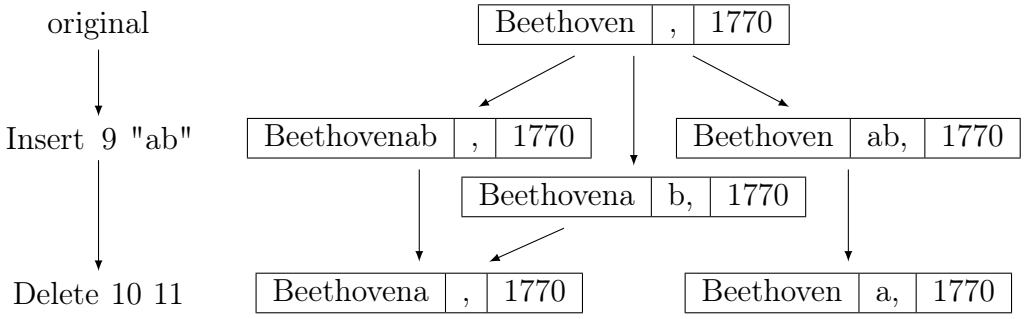
original	Beethoven	,	1770
Insert 10 "x"	Beethoven	,	x1770

Hidden in the above illustration, we have made a subtle decision in our interpretation of the two insertions. In the former insertion, index 9 is between Beethoven and the separating comma, directly on the boundary, and after the insertion we have moved the boundary to the right to accommodate the new character; in the latter insertion, index 10 is between the separating comma and 1770, again on the boundary, but this time after the insertion we do not move the boundary to accommodate the new character, preferring instead to put the new character after the boundary. In both cases, the decision is made this way because we know the middle chunk must contain exactly the string "," so that if we wish the three chunks to successfully parse we *must* put the boundaries as we did. So our parsing routine must inform our movement of boundaries during insertions; but in general the string that results from an insertion may not parse (for example, "x1770" is not a valid date). The solution is to track multiple candidate boundary locations; once we have processed all edits and the string is back in a parsing state, we then inspect where the boundary ended up. If this corresponds to any of our candidates, we then guess that the appropriate candidate faithfully tracks the motion of the boundary through all the edits. Extending the above example, this might proceed as follows:



For this insertion, we guess all reasonable new boundary positions, track them through whatever additional edits there are (in this case, none), then prune away all the can-

didate boundary positions that do not match the actual parse of the final repository. This offers a very convenient way to handle a second problem, namely, that deletions may include a chunk boundary: we choose to simply prune any candidate boundaries which would lie entirely inside a deletion. On the other hand, this nondeterminism also opens us up to another kind of problem. Consider what happens if we subsequently delete the just-inserted "b":



We can choose between different boundary locations by examining where the parser tells us chunk boundaries actually fall; but it is not clear that one path or the other from the starting boundary locations to a particular choice of final boundary locations is more canonical. (Choosing one path or the other, in this case, corresponds to choosing whether to pass the temporary insertion of a "b" to the composer lens or the separator lens to handle.) When there are many paths, our current heuristic arbitrarily chooses one and discards the rest.

A final subtlety is that insertions and deletions can make the boundary jump to a completely fresh place unrelated to the previous boundary location.[3] To see why, we will have to briefly step away from the running example used in the prototype implementation. In our new example, we will suppose that we have two fields, that the first field is terminated by the string "abc", and that the second field does not contain "c" at all. For example, "foo abc bar baz abc field2" would be a perfectly reasonable repository in this format, representing the pair ("foo abc bar baz abc", "field2"). Now a deletion like Delete 18 19 shows how boundaries can jump:

original	foo abc bar baz abc	field2
Delete 18 19	foo abc	bar baz ab field2

Our existing heuristics would only generate candidate boundaries near the deletion point. Insertions can be similarly problematic; for example, re-inserting the deleted "c" would cause the boundary to jump back, even though no boundary was near the insertion point. In case none of our boundary candidates match the actual final boundary location, we bail: instead of passing on alignment information derived from the user's actions, we pass on an edit which completely overwrites

[3] In certain pathological cases, it seems possible to cause the boundary to jump not just far from its previous location, but also far from the insertion or deletion point.

the repository. So for the above scenario, we would generate and pass on the edits [Delete 0 19, Insert 0 "foo abc"] to the edit processor for the first field and [Delete 0 6, Insert 0 "bar baz ab field2 "] for the second field.

When editing lists rather than pairs, we use essentially the same repertoire of tricks, with the exception that in case the correct boundary locations are not all among the candidates we resort to using the diff algorithm rather than emitting a complete rewrite edit. As noted in the source, there are many obvious opportunities for improvement here.

Overall, the process described above succeeded in transforming string edits into structured edits correctly reasonably often during our limited experiments, but it is unsatisfactory in many ways. It is not clear that its effectiveness would scale well with the complexity of the format being parsed. Our example was particularly simple, so perhaps there are hidden difficulties that a larger-scale example would expose. Additionally, the current implementation does not make a serious attempt to realize the performance advantage that edit lenses are intended to enable. Entire repository strings are reparsed on every key stroke, and the edit translation functions walk at least the spine of the repository and in some cases do considerably more. Thirdly, the string modules are built up compositionally, like modules and lenses are. This is convenient from the perspective of an implementor, but one of the lessons of matching lenses [6] is that a global analysis and optimization of changes can often make better decisions than a compositional one. A final consideration is that the heuristics here do not adequately reflect all user actions, and in particular copy-and-paste operations on lists do not get translated to reordering edits; however, unlike the previous considerations, it seems that the heuristics could be extended with some engineering effort to handle this transformation. A really satisfactory transformation between user actions and module edits seems to be a ripe area for serious research. Incremental parsing seems a promising area to look to for inspiration, though the problem being explored here seems to be slightly richer. Incremental parsers are very good at improving the efficiency of parsing when the string under consideration has not changed much; however, they do not make a serious attempt to track provenance or alignment. In particular, if there is editing information available, typical incremental parsers will use this to find the smallest region that needs to be reparsed but otherwise ignore all the rich semantic content available in the edit, and it is not clear how to recover this valuable alignment information just by looking at the (efficiently-produced) parse trees. It does seem possible to align the chunks of the parse tree that were not re-parsed; but an ideal algorithm would also be able to give some information about the connection between the newly parsed chunk of the tree and the old tree.

4.4 Conclusion

The full source is given in Appendix A. It includes an elegant core library, which is an indication that the theoretical foundations discussed in this dissertation can be realized as code, and an associated program which extends the foundations and shows one way to extract alignment information from observations of the actions taken to modify the repository. This observation process turned out to be surprisingly difficult; for the techniques discussed here to be practical, they will need to be refined to improve their robustness and to investigate their performance characteristics. Satisfactory progress may require a general theory for lifting parsing techniques to the domain of edits. Nevertheless, the framework built here could be used as the basis for a further studies in the usability of the syntax (by generating additional example lenses), performance, and the general practical applicability of edit lenses, including in applications such as file synchronization, text editing, database engines, client-server applications, system configuration, or software model transformations.