Bidirectional Data Transformation by Calculation

MAP-i Thesis Planning Course

Hugo Pacheco

July 5, 2008
## Contents

1 Introduction 5

2 Refinements 11
   2.1 The 2LT Framework 13
   2.2 Do stateful refinements make sense? 16

3 Lenses 19
   3.1 Lenses for trees 20
   3.2 Lenses for relational databases 22
   3.3 Lenses for strings 23
   3.4 Schema-aware lenses 25
   3.5 Quotient lenses 26
   3.6 “Lenses” for structured-documents editors 27
   3.7 Lenses with view complement 28
   3.8 Mathematical properties of lenses 29
   3.9 A lens library 30
      3.9.1 Lenses for relational tables 31
      3.9.2 Checking for well-behavedness 33
      3.9.3 A synchronization example 35
      3.9.4 A rewrite system for lenses 38

4 Towards mixing refinements and abstractions 41
   4.0.5 Type extensions and dependent types 42
   4.0.6 Considering type invariants 43
   4.0.7 Considering value equivalences 43
   4.1 A lens library with type invariants 44
      4.1.1 Well-behaved lenses modulo invariants 45
      4.1.2 Conditional lenses 46

5 Generical Model Transformations 51
Chapter 1

Introduction

Data transformations frequently occur in software engineering, whenever interoperability between different data models is pursued, even though the programmer is not always aware of the techniques they comprise.

With the ever growing list of programming languages, transforming a data format into a different format is essential to “bridge the gap” between technology layers and ensure sharing of information among software applications. Moreover, users generally expect transformations to be bidirectional, in the sense that changes made to one of the models can be safely propagated to its connected pair (imagine the synchronization of a laptop’s and a cellphone’s contacts lists).

The naive way to create a bidirectional transformation is to engineer two unidirectional transformations and manually prove that they are consistent in some particular sense, what is likely to cause a maintenance problem, besides being a notoriously expensive and error-prone task. If changes are made to any of the models, the transformations have to be redefined and it must be ensured that the new transformations maintain consistency.

The second choice is to design a semantic space in which one expression defines a transformation in both directions. Following this notion, approaches to bidirectional transformations have emerged in the most diverse computing domains, including heterogeneous data synchronization [FGM+07, BFP+08, FPP08, MHT04a, KH06], software model transformation [Ste07, XLH+07b], schema evolution [COV06, BCPV07, BMS08], constraint maintenance for graphical user interfaces [Mee98], interactive structure editing [HMT04] and relational databases [BPV06].

Designed for the ease of use, each approach generally describes a domain-specific language that imposes restrictions on the transformations targeting the particular application domain it concerns. These restrictions are embedded into language combinators that stimulate the reuse of transformations and can be easily assembled to describe more complex transformations. The uniqueness and difficulty to design a bidirectional language lies in the neat balance between the expressiveness the combinators provide and the robustness the semantic restrictions impose.

Classification of Bidirectional Transformations  

Bidirectional transformations can be classified in terms of propagation of information from the source to the target model. We will often refer to concrete and abstract models, assuming that concrete models contain more information than abstract ones.

We distinguish three non-exclusive classes of bidirectional transformations, knowing that transformations exist that do not fit into this classification, as we will later discuss.

1. Refinements. The purpose of data refinement is to transform abstract specifications into low-level concrete implementations that have equal or more information. Bidirectional semantics for calculational data refinement [MG90, Oli07] appeared from the need to reverse program refinements as applied, for instance, to the reverse engineering of legacy databases [NOFL99]; company fusions and information systems upgrades are everyday events in the world of enterprises and it is imperative to ensure errorless and consistent data migrations [BS95]. A refinement establishes a connected representation/abstraction pair of transformations, as expressed in the following diagram.
The inequation $A \leq B$ abbreviates the fact there is an injective and entire relation $To$ (the representation relation) and a surjective and possibly partial function $From$ (the abstraction relation), such that the property $From \circ To = id_A$ holds. Since the equality of two relations is a bi-inclusion, the above equation has two readings: $id_A \subseteq From \circ To$ secures that every element of $A$ has a representation in $C$ (the no loss principle); and $From \circ To \subseteq id_A$ ensures that only one element of $A$ will be transformed into some inhabitant of $B$ (the no confusion principle). Although the calculus is specified over relations [Oli07], $To$ is frequently a function. For the sake of a clearer presentation, we will reason about data refinements on their functional fragment.

The above property can be described in the next equation. The lower-case nomenclature yields a distinct notation for functions.

$$\forall a \in A. \text{from} (\text{to } a) = a$$

We can compose refinements in an increasing chain of concreteness, where the abstract model is the interface to the lower-level data representations:

2. **Lenses.** Another approach to bidirectional data transformation is known in classical relational database theory as the view-update problem [BS81], where a concrete model is abstracted into a view and changes made to the view are propagated as updates to the original format, due to knowledge on the original concrete model. Foster et al [FGM+07] have developed a data synchronization framework named Harmony, that builds on bidirectional transformations with view-update denoted lenses.

A lens complements the notion of refinement by considering the opposing transformation scenarios: a lens transformation comprises the definition of a view, some sort of projection that typically ignores details from the original format.

Since it defines a transformation from a concrete model to a more abstract model that contains less information, a lens can described as an abstraction. An abstraction is just the dual of a refinement:

$$\forall a \in A. \text{get} (\text{create } a) = a$$

The validation property for abstractions is straightforward from inverting 1.1 for refinements.
This diagram somehow resembles the concepts found in stateful functional programming with monads [Wad92]. The forward transformation get projects an abstract view from a concrete model. On the other side, the backward transformation provides two options: if the old concrete model is available through the state, then an updated concrete model is generated by applying put to the abstract view; otherwise, only the abstract view is considered and the new concrete model generated via create. Adopting the vocabulary, we call create and put the stateless and stateful backward transformation, respectively.

For the complete definition of a lens, we need to provide additional semantic properties for put. Therefore, for a lens to be well-behaved, its forward and backward transformations must at least satisfy the properties of acceptability and stability [AC07], respectively. These properties will require all transformations to be total, including get that was previously accepted to be possibly partial. We will debate the reasons for such properties later in Chapter 3.

The forward transformation get is acceptable if put and create capture all the information in the abstract model.

$$\forall a \in A. \text{get}(\text{create} a) = a$$  \hspace{1cm} (1.3)

$$\forall a \in A, c \in C. \text{get}(\text{put}(a, c)) = a$$  \hspace{1cm} (1.4)

The backward transformation put is stable if it does not drop concrete information: abstracting and immediately putting back a concrete instance shall return exactly the same instance.

$$\forall c \in C. \text{put}(\text{get} c, c) = c$$  \hspace{1cm} (1.5)

A lens is also said to be very-well behaved if it is composable such that the following equation is true.

$$\forall c \in C, a \in A, a' \in A. \text{put}(a', \text{put}(a, c)) = \text{put}(a', c)$$  \hspace{1cm} (1.6)

The compositability property states that applying an update after another update results in exactly the same as just applying the last update.

3. **Isomorphisms.** The latter case is when a bidirectional transformation is expressible either as a refinement or a lenses, meaning that both unidirectional transformations are bijective functions or, in other words, structure-preserving mappings.

Bijective functions play a fundamental role in the definition of isomorphisms, where both source and target models contain exactly the same information, but just present it differently.

---

1 These are, in fact, the getput and putget laws from [FGM+07]. createget, (1.3) in this document, can be found in [BFP+08].
∀ a ∈ A. iso(iso a) = a

∀ b ∈ B. iso(iso o b) = b

An isomorphism is also a reversible transformation because the work performed by each unidirectional transformation can be undone by applying its entwined pair. As long as these correspond to each other inverse and the inverse of an injective function always exist, one single transformation suffices to describe the overall bidirectional transformation, if it is constructed in a way so that each step can be run forward and reverse.

Bidirectional languages that preserve information are easier to reason about and provide stronger guarantees on the composition and inversion of transformations. A variety of bijective languages has been proposed, including picklers and unpicklers [KEN04], data bindings [AJ07], embedding projection pairs [BEN05, Ram03], updatable views for object sharing in object-oriented databases [OT94], structured-documents editors [HMT04] and XML-text interoperability [BMS08].

Data bindings are natural isomorphisms, since the un/serialization of datatypes across different languages encompasses inessential changes in names and structure. In [AJ07], a mechanism for inferring XML Schema-Haskell data bindings is implemented using Generic Haskell, such that the automatically derived coercions are type-preserving, i.e., isomorphisms.

Another famous example is XSugar [BMS08], a bidirectional language to convert between XML documents and text representations. The main focus of the language is the reversibility of transformations: the result of applying a transformation to produce an output must be undoable by applying the reverse transformation to the output and getting back the original input, meaning that no information is lost. However, XSugar is not purely bijective: performing a roundtrip to an XML document should yield the exact same document modulo an equivalence relation that captures the loss of information related to the normalization of XML documents, renaming of unnamed items and reordering of XML elements described by unordered representations.

Two-level Transformations Coupled software transformation involves the modification of multiple software artifacts such that they remain consistent with each other [Lam04, CV07a]. Any coupled transformation scenario comprehends techniques for the reconciliation of artifacts: how and which transformations made to one artifact can be propagated to the others. Thus, bidirectional transformation is a perfect example of coupled transformation, considering its main concern in the definition of coherent reconciling transformations capable of reestablishing, at any time, the global consistency among two artifacts.

A more particular instance of coupled transformation is two-level transformation (2lt for short), where the coupled artifacts are the types over which the transformation occurs on the one hand, and the data instances that conform to those types on the other hand [COV06]. Here, type is a placeholder for any term denoting structure, such as schema, format, meta-model, and others.

The phenomenon of two-level data transformation occurs in a variety of contexts. For example, software maintenance commonly involves enhancement of the data formats employed for storing or exporting an application’s data. Typically such enhancements are fairly conservative, such as adding new fields to the format. When the enhanced format only serves internal data storage, a one-off conversion of old data into new data may be sufficient to restore conformance. When the format concerns data exported to other applications, or shared with older versions of the same application, old-to-new as well as new-to-old data conversions may be needed on a repetitive or continuous basis.

Two-level data transformation also encompasses less conservative format changes, such as cross-paradigm data mappings. For example, the logic of an application may be programmed against an XML schema, while for efficient storage of its persistent data a relational database is employed. The required data mapping involves a format transformation from an XML schema to an SQL schema, as well as forward and backward data conversions between XML documents and an SQL database. Another popular data mapping example is the interoperability between UML specifications and Java source
code implementations. Unlike format enhancements in the maintenance context, data mappings typically involve the handling of profound paradigm discrepancies.

Another scenario where two-level data transformation can be applied is the synchronization of data of different formats. It is common to have data distributed across different platforms, that we want to unify at some point. Consider, for instance, that someone wants to synchronize the contacts lists in a cellphone with the address book in a personal computer. A possible solution is to convert one format into another, but that does not guarantee that data can be translated in both directions under the same circumstances. In the general case, both data formats can be (bidirectionally) transformed into a common representation/view, to and from which data can be pushed and fetched “equidistantly” by instances of both formats.

Other contexts in which two-level data transformations may play a role include: system integration, where data needs to be exchanged between independently developed applications; evolution of programming languages, where grammar modifications between versions spark the need for migration of source programs; and model-driven engineering where high-level meta-model transformations give rise to conversion of their instances.

Any bidirectional language that embodies a type system for defining the domains of bidirectional transformations can be viewed as a two-level transformation language.

**Structure of the Document** We follow this introduction with a more detailed description on the properties of particular bidirectional transformations and how they can be embedded into programs that tackle real-world application scenarios.

In Chapter 2, the focus is on data refinement and on how the 2LT Framework can help in incorporating schema-aware model transformations into applications with a strong premise on reliability.

At the level of abstractions, the main research outcomes are harnessed into the Harmony framework for synchronization of heterogeneous data formats, based on transformations with view-update named lenses. Chapter 3 develops on the theory behind lenses and discusses some of the proposed extensions to address special synchronization scenarios.

Although transformations can be typically classified as refinements or abstractions, often is the case when the programmer’s needs impose less restrictive scenarios. In Chapter 4, we discuss the interoperability between refinements and lenses in systems that allow fine tweaks to the lens properties, providing a better control of the expressivity of transformations without compromising their robustness.

This document ends with an open discussion on some of the existing approaches to more generical model transformations that consider both the source and target models for performing updates on either of them. Unlike refinements, that are non-stateful, and lenses, that assume notion on the original model for issuing view updates, such transformations are dually stateful (stateful in both directions) and address wider application domains. However, it is also harder to define reasonable properties and boundaries for these transformations, as presented in Chapter 5.
Chapter 2

Refinements

The development of a large, complex system may involve different phases of design, starting from the high-level formal specification, passing through middle-level language implementations (such as C, C++) and ending into low-level executable programs. The verifiable transformation of abstract to concrete programs in a staged process is the concept of stepwise program refinement. Data refinement, on its side, can be used to transform the data models used by the system in the different stages, from the interactive objects used at the interface level (trees, abstract lists, GUI components such as buttons, etc), that become explicit pointer structures (arrays, linked lists, structs and the like) at the middleware level and are stored as persistent memory records (scattered along non-sequential allocation addresses in an hard drive) at the machine level.

A simple example of a program mapping that can be modeled as a refinement is stated in Figure 2.1, for the conversion of an UML class diagram into Java source code. The transformation itself is straightforward.

Figure 2.2: Sample refinement of an XML tree into an SQL table.

In another situation, Figure 2.2 presents a staged data refinement for an XML document tree into an SQL relational table: through a simple model refinement new elements are added to the document, whose
new implicit list is mapped into a database table by indexing each element, to guarantee unique identification. You should note that the concrete table has, in fact, more information than the corresponding abstract hierarchy because, although unique, indexes do not need to be sequential: the last refined table cannot be expressed as a list.

Note that the transformation is merely driven by the values, and no knowledge on schema information is used. However, it is harder, if not ambiguous, to reason about the semantic properties of transformations without binding types to them. For instance, consider the following XML Schemas.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="xml">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="a" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

**Figure 2.3:** XML Schema for a single element `a`. In a more formal notation, we represent it with the upper-case `A`, ignoring the root element.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="xml">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="a" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

**Figure 2.4:** XML Schema for a list of `a` elements. In a more formal notation, we represent it as `[A]`, ignoring the root element.

The transformation step that we have called "model refinement", depending on the schemas that we choose for the source and target types, can either represent a format evolution scenario or a simple type-preserving transformation (just changes the values, not the type). The ambiguity here is in the semantics given to the single XML element `a`: the type may explicitly express that only one element is allowed (Figure 2.3), or permit zero or more elements (Figure 2.4) but the XML tree just happens to consider a single one.

A two-level transformation for data refinement can then be achieved by assigning schema information to the data models. The reasoning about the transformation itself also becomes easier if types are considered than by pure analysis of the value differences.

Exploiting schema information, it becomes easier to define the bidirectional two-level transformation `tolist`, that allows non-empty repetition of some type `A` and is common for XML format evolution.

```
A \xrightarrow{\{\} \leq} [A]
```

We remark that `head` is a partial function that is not defined for the case of the empty list, yet this is admitted by the theoretical definition in Chapter 1. Testing the refinement property (1.1) for all input values, we can prove that `tolist` is a valid refinement.

\[ \forall a \in A. \text{from} (\text{to} a) = \text{head} (\{\} a) = \text{head} [a] = a \]
The XML to SQL data mapping also comprises a well-known bidirectional transformation for list elimination that we name `fromlist`. The symbol \( \rightarrow \) represents finite maps, suitable for the encoding of database tables. More information, along with other transformations for hierarchical to relational data mapping, can be found in [COV06].

\[
\begin{array}{c}
\text{seq2index} \\
\leq \\
\text{list} \\
\rightarrow \\
A
\end{array}
\]

Written in Haskell, the value transformation functions look as follows (\( a \rightarrow b \) and \( \mathbb{N} \) are encoded as \( \text{Map} \ a \ b \) and \( \text{Int} \)):

\[
\begin{align*}
\text{seq2index} &: [a] \rightarrow \text{Map} \ a \ b \\
\text{seq2index} &= \text{foldr} (\lambda (i, a) m \rightarrow \text{insert} \ i \ a \ m) \, \text{empty} \\
\text{list} &: \text{Map} \ a \ b \rightarrow [a] \\
\text{list} &= \text{fold} (:) \ [ ]
\end{align*}
\]

This proof is less trivial. We will exemplify it for a single value.

\[
\begin{align*}
\text{from} \ (\text{to} \ [2,1,3]) &= \text{list} \ (\text{seq2index} \ [2,1,3]) = \text{list} \ \{0 \mapsto 2, 1 \mapsto 1, 2 \mapsto 3\} = [2,1,3] \\
\text{Is is easy to prove that the inverse transformation does not preserve table indexes and is not a valid refinement.}
\end{align*}
\]

\[
\begin{align*}
\text{seq2index} \ (\text{list} \ \{0 \mapsto 2, 2 \mapsto 1, 4 \mapsto 3\}) &= \text{seq2index} \ [2,1,3] = \{0 \mapsto 2, 1 \mapsto 1, 2 \mapsto 3\} \\
&\neq \{0 \mapsto 2, 2 \mapsto 1, 4 \mapsto 3\}
\end{align*}
\]

2.1 The 2LT Framework

Bidirectional transformations for two-level data refinement have been the focus of a series of papers from the Theory and Formal Methods group at the University of Minho. In early work, Cunha et al [COV06] have shown how data refinement theory can be employed to formalize two-level data transformation, and how an implementation in the functional programming language Haskell can capture this formalization in a type-safe manner.

The inequations of data refinement can be used as rewrite rules that replace one datatype by another. When used as a left-to-right rewrite rule, a data refinement inequation \( A \leq B \), witnessed by functions \( to \) and \( from \), can be interpreted as a two-level data transformation step that takes its input datatype \( A \) into the triple \( (B, to, from) \).

Representation of types and rules  The core of the 2LT Framework is modeled in Haskell as a system of type-changing rewrite rules, that operate on Haskell types, and consists on the following declarations:

\[
\begin{align*}
\text{type Rule} &= \forall \ a, \text{Type} \ a \rightarrow \text{Maybe} \ (\text{Rep} \ (\text{Type} \ a)) \\
\text{data Rep} \ a \ \text{where} \\
&\quad \text{Rep} \ a b \rightarrow \text{Type} \ b \rightarrow \text{Rep} \ (\text{Type} \ a) \\
\text{data Ref} \ a b = \text{Ref} \ \{ \text{to} :: a \rightarrow b, \text{from} :: a \rightarrow b \} \\
\text{data Type} \ a \ \text{where} \\
&\quad \text{Int} :: \text{Type} \ \text{Int} \\
&\quad \text{Prod} :: \text{Type} \ a \rightarrow \text{Type} \ b \rightarrow \text{Type} \ (a, b) \\
&\quad \text{Either} :: \text{Type} \ a \rightarrow \text{Type} \ b \rightarrow \text{Type} \ (\text{Either} \ a \ b) \\
&\quad \text{List} :: \text{Type} \ a \rightarrow \text{Type} \ [a] \\
&\quad \text{Map} :: \text{Type} \ a \rightarrow \text{Type} \ b \rightarrow \text{Type} \ (\text{Map} \ a \ b) \\
&\ldots
\end{align*}
\]
Here, we use a value-level representation of datatypes [HLO06], where a value of Type a is the representation of type a. For instance, the value Prod Int Int represents type (Int, Int).

Note that, in this declaration, the type a that parameterizes Type a is restricted differently in the result of each constructor. This makes the difference between a generalized algebraic data type (GADT) [PWW04] and a common Haskell 98 parameterized data type, where the parameters in the result type must always be unrestricted in all constructors. The Type a definition, the parameter a of each constructor is restricted exactly to the type that the constructor represents, although different constructors can share the same parameter a.

The representation type Rep encompasses that a type a can be represented by a type b, as witnessed by the value-level functions to :: a → b and from :: b → a bundled in the Ref constructor, that defines a refinement from a to b. Note that only the source type a escapes from the Rep constructor, while the target type b remains encapsulated and is existentially quantified.

The Rule type defines a generalization of Rep for any input type. It expresses that a two-level transformation step is a partial function taking a type into a view of that type: the Haskell Maybe type conveys such partiality (data Maybe a = Nothing | Just a).

**Two-level transformation combinators** Two-level data transformation pipelines are built in a com-positional fashion. To construct complex two-level transformations from basic ones, combinators are defined for identity, sequential composition, left-biased choice, repetition, and generic traversal:

\[
\text{nop :: Rule} \\
\text{nop x = Just (Rep (Ref id id) x)} \\
(\triangleright) :: \text{Rule} \rightarrow \text{Rule} \rightarrow \text{Rule} \\
(f \triangleright g) a = \text{do Rep (Ref t1 f1) b } \leftarrow f a \\
& \text{Rep (Ref t2 f2) c } \leftarrow g b \\
& \text{return (Rep (Ref (t2 o t1) (f1 o f2)) c)} \\
(\odot) :: \text{Rule} \rightarrow \text{Rule} \rightarrow \text{Rule} \\
\text{everywhere :: Rule} \rightarrow \text{Rule} \\
\text{many :: Rule} \rightarrow \text{Rule} \\
\text{somewhere :: Rule} \rightarrow \text{Rule}
\]

These combinators are common for typed strategic rewriting libraries [LP03]. They allow the combination of local, single-step transformations into a single global transformation. Several local, single-step transformation rules for hierarchical-relational mapping and XML format evolution are defined in [COV06]. These rules are implemented in Haskell in a straightforward way. For example, the rules tolist and fromlist are implemented as follows:

\[
\text{tolist :: Rule} \\
\text{tolist a = return (Rep (Ref (:[]) head) (List a))} \\
\text{fromlist :: Rule} \\
\text{fromlist (List a) = return (Rep (Ref seq2index list) (Map Int a))}
\]

Since we represent types with the universal representation Type, the 2LT Framework requires separate interfaces that convert specific language schemas to Type. The work in [BCPV07] describes how to equip the core rewrite system with front-ends for XML recursive trees and SQL relational databases.

**One-level transformation** Since the approach to two-level transformation is combinatorial, transformations are calculated through composition of smaller single-step transformations. At the value-level, the resulting transformations can be a source of inefficiency. Consider the simplest example for sequentiation of the nop combinator:

\[1 \text{N is encoded in Haskell as Int.}\]
Here, the resulting value transformations are \( to = from = id \circ id \), what everyone recognizes to be equivalent to \( id \). If a general calculus is available for value-level transformations, more complex functions can also be simplified following the same reasoning.

Adopting this philosophy, Cunha and Visser [CV07a] have extended the original Haskell implementation to cope with point-free function representations.

The point-free style of programming, popularized by Backus [Bac78], defines a calculational algebra free of bound variables and quantifiers, and is characterized by a rich set of algebraic laws. We can define some functional point-free combinators in the point-wise style. Their type signatures should be straightforward from their definitions. For more information about the laws ruling point-free program calculation see [Gib02, Cun05].

\[
\begin{align*}
\text{\( \vdash \)} & \colon a \to (b \to a) \\
\text{\( g \)} = \lambda b \to a & \quad \text{id} \colon a \to a \\
\text{\( i1 \)} & \colon a \to Either a b \\
\text{\( i1 \ a \)} = \text{Left} a & \quad \pi_1 \colon (a, b) \to a \\
\text{\( i2 \)} & \colon b \to Either a b \\
\text{\( i2 \ b \)} = \text{Right} b & \quad \pi_2 \colon (a, b) \to b \\
\text{\( (\nabla) \colon (a \to c) \to (b \to c) \to Either a b \to c \)} & \quad (\triangle) \colon (a \to b) \to (a \to c) \to a \to (b, c) \\
\text{\( (f \nabla g) \ (\text{Left} \ a) \)} = f \ a & \quad (f \triangle g) \ a = (f \ a, g \ a) \\
\text{\( (f \nabla g) \ (\text{Right} \ b) = g \ b \)} & \quad (f \times g) \ (a, b) = (f \ a, g \ b) \\
\text{\( (+) \colon (a \to b) \times (c \to d) \to Either \ a \ c \times Either \ b \ d \ (\times) \colon (a \to b) \times (c \to d) \to (a, c) \to (b, d) \)} & \quad (f + g) \ (\text{Left} \ a) = \text{Left} (f \ a) \\
\text{\( (f + g) \ (\text{Right} \ b) = \text{Right} (g \ b) \)} & \\
\end{align*}
\]

The simplification by rewriting of point-free functions is performed through the application of a rich set of algebraic laws for point-free program calculation, strongly based on mathematical equational reasoning [Cun05]. The rewrite system for point-free calculation relies on similar concepts to the rewrite system for 2LT, such as GADTs and strategic combinators, with the difference that it rewrites point-free expressions into equivalent point-free expressions. It also introduces the term one-level transformation for the simplification of point-free functions.

**Coupled transformations** The paper [CV07a] focus on more challenging coupled transformations that involve not only the data instances corresponding to a source format, but also other software artifacts dependent on the format. Examples of such artifacts are functions that consume (queries) or generate (producers) values of the bound format. Consider the following diagram where \( p \) and \( q \) are a data producer and a query on \( A \):

\[
\begin{align*}
A & \supseteq B \\
p' & \text{from} \quad q' \\
x & \to \\
p & \downarrow \text{to} \quad q \\
A & \supseteq B \\
y & \downarrow \\
\end{align*}
\]

The challenge is to calculate \( p' \) and \( q' \) by fusing the compositions \( to \circ p \) and \( q \circ from \) such that they work on \( B \) directly, rather than via \( A \).
Various languages allow specific query languages for selection and transformation of portions of documents. Such queries are defined generically for different data types, and only specify specific behaviors for a few relevant subtypes. This is a well-known feature of XML query languages, that allow selection of element nodes without exhaustively specifying intermediate nodes, making them more concise and reusable.

Consider, for instance, the XPath expression //a for selecting all elements at any depth that exist in an XML document. The sentence “at any depth” does not assume any compromise with the XML Schema of the document, since the query can follow different paths to collect elements with name a: /b/a, /c/b/a, etc.

New front-ends had to be designed for translating XPath expressions and alike into point-free functions, for which the simplification calculus has already been demonstrated.

One-level combinators for specialization In [CV07b, FP07], a calculus is described for the conversion of structure-shy expressions, loosely bound to schema structure, into structure-sensitive equivalents, that are specific for the structure of documents, and vice-versa. The rewrite process encompasses the translation of XPath queries into strategic functions [LP03, Lämm06], that are themselves converted into point-free functions.

A variety of contexts can be found for optimized programs, whenever the same query needs to be run against documents conforming to the same schema multiple times or the overhead of schema-specialization is not significant due to the size of documents. For example, web services commonly involve extraction of information from XML databases. Such extractions can be expressed according to previously well-defined selection functions in the XPath language and are likely to be performed several times (imagine a regular PHP website based on a XML database). Software maintenance is also strictly related to generation of tests and summary reports on data stored in XML databases. Tests and reports tend to be executed on a regular basis. Any other contexts in which query-specialization may play a role involve application integration relying on data retrieval from XML databases.

2.2 Do stateful refinements make sense?

Previously, we have justified the existence of lenses (that are stateful abstractions) with the need to keep a record of the information that might be deleted when computing a view. Despite refinements generally dispense a state-based synchronization process (in the backward, a new source format is generated from the target format, instead of being edited as an update to the old source format), by not allowing the deletion of type information, users may replace value data with other data. Such is the case for a refinement \textit{addalt} that extends a concrete type \(A\) with an right-alternative type \(B\):

\[
\begin{array}{c}
\xymatrix{ 
A \ar@{<->}[r] & A + B \\
& \lambda \text{Left } a \rightarrow a \\
\end{array}
\]

\textit{addalt} :: Type \(b \rightarrow \text{Rule}\)

\textit{addalt} \(b\ a\) = return (Rep (Ref Left (\(\lambda\text{Left } a \rightarrow a\))) (Either \(a\ b\)))

For this refinement, whenever a value of the added alternative is provided, the backward transformation “crashes” because \textit{from} is a partial function, only defined for the range of \(to\). Note that this behavior is not problematic nor unexpected in the refinement theory, as long as \textit{from} is modeled as a partial function. Nevertheless, we can do more to avoid this behavior. A possible solution is making \textit{from} a total function by providing a default value whenever the added alternative exists. The default value must be passed as an extra argument to \textit{addalt}:
addalt :: Type b → b → Rule
addalt b d a = let from' (Left a) = a
                from' (Right b) = d
                in return (Rep (Ref Left from') (Either a b))

However, we may argue how good is the quality of the abstract information generated by from because, similarly to a pure stateless abstraction, the refinement has lost track of the original abstract information. This is where the concept of a stateful refinement makes sense: if the abstract specification is propagated when calculating a representation, the abstraction relation can later retrieve the initially replaced information. In comparison to view-update abstractions (lenses), we introduce the $a$ term for representation-update refinements.

We can extend the Haskell implementation for representation-update refinements by adding a new abstract-aware backward transformation $\text{back}$:

\begin{align*}
\text{data } & \text{RefU } a b = \text{RefU} \{ \text{to} :: a \to b, \text{from} :: b \to a, \text{back} :: (b, a) \to a \} \\
\text{data } & \text{Rep } a \text{ where} \\
& \ldots \\
& \text{RepU} :: \text{RefU } a b \to Type b \to \text{Rep } (Type a)
\end{align*}

Put in a diagram:

\begin{center}
\begin{tikzpicture}
\node (A) at (0,0) {$A$};
\node (C) at (2,0) {$C$};
\node (back) at (1,2) {$\text{back}$};
\node (from) at (1,-2) {$\text{from}$};
\path[->] (A) edge[bend right] node[above] {$\text{to}$} (C);
\path[->] (C) edge[bend right] node[below] {$\text{from}$} (A);
\path[->] (C) edge node[below, near start] {$\pi_1$} (back);
\path[->] (back) edge node[above, near start] {$\pi_1$} (from);
\end{tikzpicture}
\end{center}

addaltU, a stateful version of addalt can be expressed as:

\begin{center}
\begin{tikzpicture}
\node (A) at (0,0) {$A$};
\node (C) at (2,0) {$A + B$};
\node (back) at (1,2) {$\text{back'}$};
\node (from) at (1,-2) {$\text{from'}$};
\path[->] (A) edge[bend right] node[above] {$\text{Left}$} (C);
\path[->] (C) edge[bend right] node[below] {$\text{from'}$} (A);
\path[->] (C) edge node[below, near start] {$\pi_1$} (back);
\path[->] (back) edge node[above, near start] {$\pi_1$} (from);
\end{tikzpicture}
\end{center}

\begin{align*}
\text{addaltU} :: & \text{Type b } \to \text{ b } \to \text{ Rule} \\
\text{addaltU } b d a = & \text{let from' } (\text{Left } a) = a \\
& \text{from' } (\text{Right } b) = d \\
& \text{back'} (\text{Left } a, a') = a \\
& \text{back'} (\text{Right } b, a') = a' \\
& \text{in return } \text{Rep } (\text{RefU } \text{Left from'} \text{back'}) (\text{Either } a \ b)
\end{align*}

Consider the two possible behaviors of addaltU if an abstract data instance exists for the backward transformation: when a value of the added alternative exists, the old abstract value is returned; otherwise, the abstract value is updated normally with the new information from the concrete representation.
A stateful refinement is said to be well-behaved if it comprises the connectivity requirements of two refinements:

\[
\begin{align*}
\begin{array}{c}
\text{A stateful refinement is said to be well-behaved if it comprises the connectivity requirements of two refinements:} \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
A \xrightarrow{\text{to}} & (\text{Left } a, a) \\
A \xleftarrow{\text{back}} & (\text{Right } b, a)
\end{align*}
\]

Consequently, the representation function \(\text{to}\) must be injective and total and the abstraction functions \(\text{from}\) and \(\text{back}\) are both surjective, possibly partial:

\[
\begin{align*}
\forall a \in A. \text{ from (to } a) &= a \quad (2.1) \\
\forall a \in A, a' \in A. \text{ back (to } a, a') &= a \quad (2.2)
\end{align*}
\]
Chapter 3

Lenses

Computing is full of situations where one wants to transform a data format into a different format, maintaining the interoperability between the original format and the view. One way to address such bidirectional transformations is via two-level transformations based on calculational data refinement. Program refinement is the transformation of an abstract specification into a concrete low-level implementation. Another approach to bidirectional two-level data transformation is the well-known view-update problem, where a concrete model is abstracted into a view. Value changes made to the destination format are performed as updates to the original structure.

In relational theory, a database view is a virtual database representation created by issuing a query over the the original database, in the sense that data changes in the table alter the data shown in the view. An user interacts with a view by issuing queries and update requests on views that must be translated into requests on the underlying database. Although mapping of queries does not present particular problems, when translating a view update there is, in general, more than one possible update to the original database state.

Consider, for instance, the example from Figure 3.1. A view v is created by selecting all rows in t for which the column a has the string ‘world’. If a new value is inserted into the view, we can think of three different database updates, presented in decreasing order of concrete data preservation: since the inserted row in the view is conflicting with a row on the original table, we leave t intact; we can be more...
modest and admit the view updates to overwrite conflicting data; or, in the other extreme, we can also assume that only data in the view matters.

The view-update problem [BS81] addresses the difficulty of choosing an unique database update for each view update. Foster et al [FGM+07] have introduced lenses, a bidirectional two-level approach to the view-update problem used in the context of a data synchronization tool called Harmony. The synchronization of heterogeneous data formats can be performed by transforming them into a common image that maintains the interoperability.

![Figure 3.2: Synchronization of two tables using lenses.](image)

A diagram for such a scenario is presented in Figure 3.2. Through lens transformations, tables \( t_1 \) and \( t_2 \) can be abstracted into the same view \( t_s \). The transformations painted in red control the update of information from \( t_1 \) to \( t_2 \), and the orange ones work for the opposite from \( t_2 \) to \( t_1 \). The forward transformation \( \text{get} \), painted in blue, is used to transform each format (\( t_1 \) or \( t_2 \)) into the common view \( t_s \).

The fundamental part of the synchronization process is in how we choose to combine the \( t_s \) created from \( t_1 \) with the \( t_s \) created from \( t_2 \). A concrete synchronization example is presented later in Section 3.9.

### 3.1 Lenses for trees

In their initial approach, lenses are considered for an universe of unordered, edge-labeled trees, without repeated labels. A database table from the previous example can be simulated as a tree of the form \( \{0 \mapsto \text{hello}, 1 \mapsto \text{world}, 2 \mapsto \text{hello} \} \).

Trees are drawn horizontally from left to right. The braces refer to tree nodes, and \( x \mapsto \ldots \) denotes a child labeled with the string \( x \). Tree leafs of the special form \( \{k \mapsto \} \} \} \) are often written just \( k \). According to the authors, this simple form of trees allows more straightforward lens definitions that ignore problems with order. Ordered structures, such as lists, can be represented by assuming a predefined notation: tree nodes named \( \ast h \) and \( \ast t \) denote list head and tail structures, respectively.

The formal language for lenses defines types as sets of elements that belong to some universe, in this case, unordered trees. The semantics for lenses comprise two partial functions \( \text{get} :: \mathcal{C} \to \mathcal{A} \) and \( \text{put} :: \mathcal{A} \times \mathcal{C} \to \mathcal{C} \), where the concrete and abstract structures, \( \mathcal{C} \) and \( \mathcal{A} \), are subsets of \( \mathcal{U} \), the type universe.

For a lens from \( \mathcal{C} \) to \( \mathcal{A} \) to be well-behaved, written \( l \in \mathcal{C} \Rightarrow \mathcal{A} \), it must obey to the following properties:

\[
\forall c \in \mathcal{C}, \text{put} (\text{get} c, c) \sqsubseteq c \tag{3.1}
\]

\[
\forall c \in \mathcal{C}, a \in \mathcal{A}, \text{get} (\text{put} (a, c)) \sqsubseteq a \tag{3.2}
\]

The preorder \( f \sqsubseteq g \) indicates the partiality of \( \text{get} \) and \( \text{put} \). Formally, it expresses that, for two continuous functions \( f :: \mathcal{A} \to \mathcal{B} \) and \( g :: \mathcal{A} \to \mathcal{B} \), \( f \) is an approximation of \( g \), ie, \( f \) is less defined than \( g \) if for every value \( x \in \mathcal{A} \), \( f x \) is less defined or equally deterministic to \( g x \):

\[
f \sqsubseteq g \iff \forall x \in \mathcal{A}. f x \sqsubseteq g x \tag{3.3}
\]

In practice, we expect lenses to be total. The reason for considering partial lenses is because the lenses for transforming ordered data, such as lists, are defined by recursion and, thus, must be computed.
as chains of increasingly defined partial functions. However, the authors guarantee that all their lens combinators can actually be proved total.

In [FGM+07], the definition of lenses does not contain the backward transformation create to that we have declared in the introduction (Chapter 1). The solution for dealing with the creation of concrete models whenever no old concrete model is available comprises enriching the universe \( U \) with a special placeholder \( \Omega \) for “missing” information. By convention, \( \Omega \) is only used in lenses when it is the concrete argument of the put function. In all the other scenarios, it is propagated: get \( \Omega = \Omega \), put \( (\Omega, c) = \Omega \). C \( \mapsto \) A encloses the set of well-behaved lenses from \( C_\Omega \) to \( A_\Omega \).

At this point, you may enquire whether should not a concrete value be available for put if lens transformations traditionally start from a concrete model; it actually happens whenever value information is added to the abstract view.

Consider the lens combinator map \( l \in C \mapsto A \) that maps lens \( l \) to each subtree of the root. Given the identity lens \( \text{id} \in C \mapsto C \), calculating an abstract view of \( \{a, b\} \) through map \( \text{id} \) is the same as applying get to all the childs of the concrete tree: \( \{\text{get } a, \text{get } b\} = \{a, b\} \). Additionally, the operation that adds a new value \( c \) to the abstract view, generates an the updated view \( \{a, b, c\} \). This all seems stupidly intuitive, but assume that now we want to push the changes back to the concrete tree: it consists in applying put to all the childs of the updated abstract tree. Since no concrete pair for \( c \) exists in the original tree, a “missing” annotation \( \Omega \) is passed to put: \( \{\text{put } (a, a), \text{put } (b, b), \text{put } (c, \Omega)\} = \{a, b, c\} \).

However, since the backward transformation for id \( \{\text{put } (a, c) = a\} \) ignores the concrete value, no problem arises. If instead we employ a lens that returns a constant value \( \text{const} \) for which \( \text{get } c = v \) and \( \text{put } (a, c) = c \), the “missing” tree will get propagated to the concrete tree:
\[
\{\text{get } a, \text{get } b, \text{get } c\} = \{v, v, v\}, \{\text{put } (a, a), \text{put } (b, b), \text{put } (c, \Omega)\} = \{a, b, \Omega\}.
\]

The solution found in [FGM+07] is to consider a default tree \( \Omega \) for the cases when the concrete tree is in fact missing. Consequently, the constant lens becomes:
\[
\text{get } c = v \\
\text{put } (a, c) = \begin{cases} 
  c & \text{if } c \neq \Omega \\
  d & \text{if } c = \Omega 
\end{cases}
\]

\[\forall d \in C, \text{const } v \in C \mapsto \{v\} \]

The constant lens plays a crucial role in the definition of more complex tree combinators, such as filtering \((\forall d \in C_p, \text{filter } p \ d \in C \mapsto C_{(p)}\)
\). The abstract domain \( C_p \) prevents filter from handling abstract childs that do not satisfy the predicate. A predicate \( p \) is semantically represented as a set of children names. Selecting the children of tree \( C \) whose names satisfy \( p \) is denoted by \( C_p \). The reverse operation, \( C_p \) ignores the children whose names belong to \( p \). Naturally, the concatenation of the two sets \( C_p, C_p \) returns the original set \( C \).

More combinators are provided for transformation of trees, such as \( \forall l \in C \mapsto B, k \in B \mapsto A \). \( l; k \in C \mapsto A \), that applies two lenses in sequence, and \( \text{hoist } n \in \{n \mapsto C\} \mapsto C \) that removes an edge at the top of a source tree. Particularly interesting cases are the combinators for conditional lenses.

The first conditional lens, cond, operates on concrete trees. It is parameterized with a predicate \( Pc \), expressed as a set of trees, and two lenses \( l_1 \) and \( l_2 \). In the forward direction, it applies the get of \( l_1 \) if the concrete tree satisfies \( Pc \), or the get of \( l_2 \) otherwise. In the backward direction the behavior is similar; it applies the put of \( l_1 \) if it satisfies \( Pc \), or the put of \( l_2 \) otherwise. Note that the predicate forces the concrete domains of \( l_1 \) and \( l_2 \) to be disjoint, while their abstract domains are identical.

\[
\text{get } c = \begin{cases} 
  \text{get } l_1 \ c \ & \text{if } c \in Pc \\
  \text{get } l_2 \ c \ & \text{if } c \notin Pc 
\end{cases} \\
\text{put } (a, c) = \begin{cases} 
  \text{put } l_1 \ (a, c) \ & \text{if } c \in Pc \\
  \text{put } l_2 \ (a, c) \ & \text{if } c \notin Pc 
\end{cases}
\]

\[\forall l_1 \in C \cap Pc \mapsto A, l_2 \in C \setminus Pc \mapsto A. \ \text{cond } Pc \ l_1 \ l_2 \in C \mapsto A \]

A different way of defining a conditional lens is by basing the decision whether to use \( l_1 \) or \( l_2 \) on the abstract domain, whenever possible. Therefore, the abstract conditional acond, in the get direction, operates similarly to cond. However, to define the put direction, it applies the put of \( l_1 \) if the first
argument (the abstract tree) satisfies $Pa$ (the abstract predicate), or the put of $l_2$ otherwise. Note that if the concrete tree passed to put does not belong to the concrete domain of the chosen lens ($l_1$ or $l_2$), that same tree is replaced for the “missing” tree $\Omega$.

\[
\begin{align*}
g & = \begin{cases} 
g l_1 & \text{if } c \in Pc \\
g l_2 & \text{if } c \notin Pc
g & = \begin{cases} 
\text{put } l_1 (a, c) & \text{if } a \in Pa \land c \in Pc \\
\text{put } l_1 (a, \Omega) & \text{if } a \in Pa \land c \notin Pc \\
\text{put } l_2 (a, c) & \text{if } a \notin Pa \land c \notin Pc \\
\text{put } l_2 (a, \Omega) & \text{if } a \notin Pa \land c \in Pc
\end{cases}
\end{cases}
\end{align*}
\]

$\forall l_1 \in C \cap Pc \overset{\Omega}{=} A \cap Pa, l_2 \in C \backslash Pc \overset{\Omega}{=} A \backslash Pa$. acond $(Pa) (Pa) l_1 l_2 \in C \overset{\Omega}{=} A$

A general conditional cond is also provided by combining the behaviors of $ccond$ and $acond$, but we will skip its definition.

### 3.2 Lenses for relational databases

Remembering that the notion of view update was inherited from database theory, the authors of the lenses framework have considered the design of a new bidirectional language with a set of primitives and a type system targeted at relational data [BPV06]. For that purpose, they have developed a relational algebra where every expression denotes both a view definition and a view update policy.

Associations between the data inside tables are expressed as sets of functional dependencies of the form $X \rightarrow Y$, that warrant the left-attribute $X$ of each dependency to uniquely determine the attribute $Y$. Unlike primary key constraints, the values of the determinant attribute are not required to be unique.

The special domain for which table views are guaranteed to have reasonable view updates is restricted to databases where functional dependencies are considered to be in tree form. The set of functional dependencies $F$ for some table is in tree form if it represents a tree where each attribute node $X$ functionally depends on the attributes that build the path to $X$. For example, the set $\{A \rightarrow B, A \rightarrow C\}$ defines the tree $\begin{array}{c} A \\
B \\
C \end{array}$. Conversely, the set $\{A \rightarrow C, B \rightarrow C\}$, or the graph $\begin{array}{c} A \\
B \swarrow \\
C \nwarrow \end{array}$ is not in a tree form.

Though the intricate algebra is too heavy for a quick presentation, we resume the application of essential primitives for fusion, projection and selection with an example.

Consider a database with two tables, Albums and Tracks and the transformation example described in Figure 3.3.

Using the combinators provided by the bidirectional language, we can encode a transformation that greatly resembles the SQL syntax. In the example, we join tables Albums and Tracks into one table (Tracks1), construct table Tracks2 by dropping the Date attribute from Tracks1 and end up with table Tracks3 that collects the filtered records from Tracks2 with Quantity greater then 2. After applying the forward transformation, the resulting table Tracks3 gathers some of the information from the original database and is a database on its own right. Assume that we choose to increase the rating of Lullaby to 4, change the incorrect album of Lovesong to Disintegration, updating its quantity to 7, and delete the row with track Trust.

By applying the implicit backward transformation provided by the bidirectional expression language, we can propagate the update to the original database. Analyzing the results, the update increased the Rating of both rows with the track Lullaby. This happens because the functional dependencies need to be preserved: since attribute Track determines Rating, changing the rating for some track implies changing the rating for all rows with such track. The second update consisted in changing the album into Disintegration for all records with track Lovesong. Since no functional dependency exists between these attributes, this change is not propagated to the other rows with the same track. Note how the quantity for album Disintegration was also updated. For last, the row for track Trust was deleted. The only remark is that this operation in Tracks3 imposed the deletion of the information about the album that was associated to the track Trust. However, that information was successfully recovered from the original database and preserved in table Albums.

For a more detailed explanation on this specific example, please refer to [BPV06], from where it was copied.
3.3 Lenses for strings

To overcome issues with order, a language for bidirectional transformation of string data (Boomerang) is presented in [BFP08], extending lenses with mechanisms to correctly handle the reordering of concrete information and inspired in concepts from relational databases known under the data lineage problem, that we furtherly present.

In a similar way to the view-update problem, that tackles the problem of issuing translations for view updates, the data lineage problem bidirectionalizes database transformations by tracing data items back to the source items that originated them. Applied in data-warehousing, it generally involves the collection of data from multiple database sources and has been formalized initially for relational materialized views [CWW00] and lately for general data-warehouse transformations [CW03].

However, the data lineage problem is not a “sibling” to the view-update problem. It is concerned with identifying which specific data is the origin of particular records in a database view, and not how updates to the view would modify the original records. Also, the view-update problem is concerned with generating consistent database updates from any modifications on views and does not necessarily establish a relation between the records in a source database and the records on a corresponding view.

We continue with a simple example: imagine a single source table \( t(\text{laptop}, \text{brand}, \text{size}, \text{price}) \) that represents the pricing table of a laptop reseller and a view that creates a short version of such table \( v(\text{laptop}, \text{price}) \) that discards the detailed information

<table>
<thead>
<tr>
<th>laptop</th>
<th>brand</th>
<th>size</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacBook Air</td>
<td>Apple</td>
<td>13.3</td>
<td>1700</td>
</tr>
<tr>
<td>VAIO PMT9300</td>
<td>Sony</td>
<td>13.3</td>
<td>2200</td>
</tr>
<tr>
<td>ThinkPad Z61m IBM</td>
<td>15.4</td>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>

Considering that the two tables (the view may be virtual) are stored in a database behind a web front-end, an user that is curious about some laptop model may want to see the relevant detailed information bound to it. Tracing the provenance of a tuple involves inspecting the tuples (it can be more than one, but not for this example) in the original table that produced it, what is easy to achieve by key lookup.
For instance, for the tuple \((\text{MacBook Air}, 1700)\) in \(v\), by looking up for the key \(\text{MacBook Air}\) in \(t\) we get \((\text{MacBook Air}, \text{Apple}, 13.3, 1700)\).

Suppose now that we are dealing with text dumps of the same data, formatted to look more or less the same. Assuming a default set of string combinators, writing a bidirectional transformation (a lens in this case) to create a similar view should be straightforward:

```
"MacBook Air, Apple, 13.3, 1700
VAIO PMT9300, Sony, 13.3, 2200
ThinkPad Z61m, IBM, 15.4, 2000"
```

If we allow updates to the abstract string, for instance, sorting the table according to the laptop's price

```
"MacBook Air, 1700
ThinkPad Z61m, 2000
VAIO PMT9300, 2200"
```

the transformation that combines it with the original concrete structure would yield an output

```
"MacBook Air, Apple, 13.3, 1700
ThinkPad Z61m, Sony, 13.3, 2000
VAIO PMT9300, IBM, 15.4, 22000"
```

Although this lens is well-behaved, this output is not the expected: the information for \(\text{VAIO PMT9300}\) and \(\text{ThinkPad Z61m}\) are mixed up in the updated concrete string. This happens because, for this example, the backward transformation is processing the data based on line associations and the reordering of the abstract string breaks that positional alignment.

Returning to the context of Boomerang, if we can discover which exact concrete data generated specific abstract data, we avoid proceeding according to positional associations. Boomerang is based on the newly termed dictionary lenses, that permit the declaration of reorderable chunks in the concrete string.

Each chunk is indexed according to a key substring (that is always identified in the language) and stored in a dictionary that is used by the backward transformations. A Boomerang lens for the previous example is the following:

```
let ALPHA = [A-Za-z] +
let comp = key ALPHA . copy "", "
  . del (ALPHA . ", "
  . del (ALPHA . ", "
  . copy ALPHA

let comps = "" | <comp> . ("\n" . <comp>) *
```

The language syntax is built around regular operations, that shall be easy to understand for the average programmer. The language combinators rely on expression pattern-matching, where \(\text{copy}\) performs identity and merely copies an input string to the output, and \(\text{del}\) deletes an input string and outputs the empty string. The noticeable combinators are the specific dictionary ones: angled brackets identify reorderable concrete chunks, and \(\text{key}\) associates a key to each of such chunks. In practice, the language does not force unique keys in the pure relational sense: if repeated keys exist, they are matched by position as usual.

Abstracting the implementation details, consider the following discussion on the semantic properties of dictionary lenses. A lens is oblivious \([\text{FGM}+07]\) when its backward forward transformation \(\text{put}\) ignores the concrete argument and is similar to \(\text{create}\)
∀ c ∈ C, c′ ∈ C, a ∈ A. put (a, c) = put (a, c′)  \tag{3.4}

Therefore, quasi-obliviousness can be seen as partially ignoring the concrete domain. More concretely, it is described as an extensional property of lenses that coarsens the concrete domain C with an equivalence relation ∼ on C. A lens l ∈ C ⇔ A is quasi-oblivious with respect to ∼, denoting l ∈ C/∼ ⇔ A, if it obeys the following property:

∀ c ∈ C, c′ ∈ C, a ∈ A. \frac{c ∼ c'}{put (a, c) = put (a, c')} \tag{3.5}

The property (3.5) states that a quasi-oblivious lens is allowed to ignore the details at the concrete level that are equivalent through c ∼ c′.

Interestingly, the condition for oblivious lenses can also be expressed in terms of equivalences: if ∼ is the total relation \text{Tot}(C), all elements in C are related:

⟨∀ c ∈ C, c′ ∈ C' :: a\text{Tot}(C)b ∨ b\text{Tot}(C)a⟩ \tag{3.6}

Dictionary lenses live in the semantic space of quasi-oblivious lenses. The operation that parses a dictionary, indexing string chunks in respect to keys is in fact ignoring the order of the input and can be expressed as an equivalence that ignores that same order.

Remark that the notation for lenses has changed: l ∈ C ⊵ A gave place to l ∈ C ⇔ A. The first reason is that lenses do not consider a missing value identifier (Ω) anymore. Instead, a \textit{create} backward transformation is employed for the cases when an old concrete model is not available. The advantage in this solution is that Ω values are no longer propagated through transformations, and the kind of optionality they propose, since \textit{create} requires some decision to be made about missing values. Not considering lenses defined by recursion, unidirectional transformations are now said to be total. A well-behaved lens now comprises the properties expressed before in (1.3,1.4,1.5).

### 3.4 Schema-aware lenses

Applying lenses to manipulate non-recursive textual data formats (simple XML formats, LaTeX, SwissProt, iCalendar, etc) directly is appealing for the easiness of use it provides. However, this brings problems with order, that can be overcome by mapping the formats to more structured representations.

Exploiting schema information, as proposed in [FGK+07], has a number of advantages that avoid conflicts during transformations. The first intuitive application of schemas is to check whether a specific format belongs to the domain of a lens l ∈ C ⇔ A. Second, schemas can express structure invariants. Both in lenses for trees and strings, we have seen approaches to represent database tables and the need to preserve some of the type constraints such as primary keys.

The 2LT Framework, as a two-level approach to schema evolution, can help in integrating these concepts with lenses, in the sense that it is based on a type-safe Haskell implementation: bidirectional transformations are guided by a type transformation, that the library uses to check if the bidirectional value transformations are well-typed and thus operate between the transformed types. If they are not, the type checker shall issue a type error; the library is compositional, and the composition of two transformations from A to B and from B′ to C is only possible if the type checker can prove that B = B′. The internal type representation supports all the type invariants that can be expressed for Haskell data types, being the most traditional example the encoding of relational tables as finite maps, that guarantee the key invariants.

However, a more generic type representation has the cost of having to develop separate interfaces for separate languages, where those interfaces are responsible for the conversion to and from the universal representation. In terms of type-safety, the library first reads a schema and the corresponding documents are read in order to the schema information. If conflicts arise, a parsing error is launched.
3.5 Quotient lenses

The advantages of a precise type system (as the ones based on regular expressions for strings and on sets of elements for trees) can easily turn into a disadvantage because lenses become sensible to inessential details such as whitespaces, that must match after a round-trip. Loosening the lens properties is appropriate for such cases, as we have seen when defining quasi-obliviousness, that introduced an equivalence relation on the concrete domain to ignore the information bound to the sequential order of elements.

In an attempt to make lenses applicable to wider domains, a general theory of quotient lenses (q-lenses for short) is proposed in [FPP08], denoting bidirectional transformations that are well-behaved modulo two equivalences, one on the concrete domain ($\sim_C$), as for quasi-oblivious lenses, and another on the abstract domain ($\sim_A$). A q-lens $l$ in relation to these equivalences is expressed as $l \in C/\sim_C \to A/\sim_A$.

The properties for acceptability and stability of lenses are also loosened with the equivalence relations for a q-lens:

$$\forall a \in A. \ get(\ create\ a) \sim_A a \quad (3.7)$$
$$\forall a \in A, c \in C. \ get(\ put\ (a, c)) \sim_A a \quad (3.8)$$
$$\forall c \in C. \ put(\ get\ c, c) \sim_C c \quad (3.9)$$

Although the domains are relaxed on the equivalences, lens transformations still operate on exact structures: we have $\ get :: C \to A$ and not $\ get :: C/\sim_C \to A/\sim_A$. To ensure that quotiented transformations treat equivalent structures equivalently, additional properties are imposed:

$$\forall c \in C, c' \in C. \ c \sim_C c' \Rightarrow \ get c \sim_A \ get c' \quad (3.10)$$
$$\forall c \in C, c' \in C, a \in A, a' \in A. \ a \sim_A a' \Rightarrow \ put a c \sim_C \ put a' c' \quad (3.11)$$
$$\forall a \in A, a' \in A. \ a \sim_A a' \Rightarrow \ create a \sim_C \ create a' \quad (3.12)$$

By enriching the domains and codomain types of lenses with equivalences, quotient lenses support a way to abstract from typical parsing and pretty-printing operations (such as dealing with whitespaces) that are the “job” of the equivalences. We can, for instance, define some lens $l \in B/\sim_B \to A/\sim_A$. To ensure that quotiented transformations treat equivalent structures equivalently, additional properties are imposed:

$$\forall a \in A. \ canonize(\ choose\ a) = a \quad (3.13)$$

Note that (3.11) relate to (3.5) for quasi-oblivious lenses, except that the results of the transformations have to be equivalent (modulo $\sim_C$) and not equal.

We introduce the “redundant” canonizer definition for being close to the syntax in [FPP08]. However, and again, they are semantically pure abstractions.

Assembling the new names to the property for abstractions (1.2):
The difference to lenses is that canonizers \((C \ni B)\) do not take into account an original concrete format when putting back updates, in opposition to lenses \((C \ni B)\). Canonizers are more flexible since they impose weaker laws (remember the discussion in Chapter 1).

Continuing the example, defining a canonizer only becomes useful if we can compose \(cn\) with \(l\), yielding a new q-lens \(l' \in C \ni A/\sim_A\). Expressed in a diagram, it resembles the composition of lenses (we omit the equivalences for simplicity):

\[\begin{align*}
C & \xrightarrow{\text{canonize } cn} B \\
B & \xrightarrow{\text{choose } cn} \ni A \\
A \times B & \xrightarrow{\text{id} \times \text{canonize } cn} \ni A \times C \\
\ni A \times C & \xrightarrow{\text{create } l'} \ni A
\end{align*}\]

In order to guarantee that the lens \(l'\) is well-behaved, \(\text{put } l' = \text{choose } cn \circ \text{put } l \circ (\text{id} \times \text{canonize } cn)\) has to be injective (as we will see later in Section 3.8), but from the properties for abstractions we just know that \(\text{canonize } cn\) is a surjective function. Therefore, a new equivalence \(\sim_C\) is put on \(C\), stating that all values in \(C\) that are canonized to equivalent values in \(B\) are equivalent:

\[\forall c \in C, c' \in C. c \sim_C c' \equiv \text{canonize } cn c \sim_B \text{canonize } cn c'\]  
(3.14)

The condition is, in fact, holding the required injectivity of \(\text{canonize}\) and turning it into a bijective function. Precisely, it is deriving an isomorphism from the canonizer.

The \(\text{lquot}\) combinator, that computes the q-lens \(l'\), performs the left composition of a canonizer with a q-lens, and comprises

\[\begin{align*}
\text{get } c & = \text{get } l (\text{canonize } cn c) \\
\text{put } a & = \text{choose } cn (\text{put } l (a, \text{canonize } cn c)) \\
\text{create } a & = \text{choose } cn (\text{create } l a)
\end{align*}\]

\[\forall cn \in (C \ni B/\sim_B, l \in B/\sim_A \ni A/\sim_A). \text{lquot } cn l \in C/\sim_C \ni A/\sim_A\]

The \(\text{rquot}\) combinator is symmetric and composes a canonizer to the right of a q-lens. As long as it imposes the same equivalence on \(C\), it constructs a well-behaved lens.

### 3.6 "Lenses" for structured-documents editors

Related topics on lenses and reversible transformations have been researched by the Programmable Structured Documents group at the University of Tokyo. The first contribution is the injective language \(\text{Inv}\), embedding various program derivation and inversion techniques for point-free functions [MHT04b], among which the automatic derivation of injective variants for non-injective functions. An high-level bidirectional language \(X\) [MHT04b] has also been developed, where programmers write injective forward transformations in a functional way and the synchronization process is automatically derived by algebraic reasoning. This is integrated into a structure editor for XML [HMT04] that calculates transformations through editing operations.

The employed transformations, although named lenses, resemble the properties of refinements \((A \ni C)\): the backward transformation is stateless, no loss of information occurs in the forward direction, and
the property stipulated for the transformations is (1.1)\(^3\), stating that pushing an unedited view back always gives the original document. The difference is that an additional property is considered (3.15), baptized \( \text{putgetput} \), to guarantee that when a representation is updated, computing \( \text{put} \) backwards to calculate an updated source and generating a new representation with \( \text{get} \) is sufficient to propagate all the changes.

\[
\forall a \in A. \text{put} (\text{get} (\text{put} a)) \sqsubseteq \text{put} a \tag{3.15}
\]

The preorder \( \sqsubseteq \) is used instead of equality (\( = \)) because \( \text{put} \) is a possibly partial function (as \( \text{from} \) is for refinements), and the domain of \( \text{get} \) may be different to the range of \( \text{put} \).

Since the language deals with refinements, data duplication is made possible. Other scenarios, such as information removal, are handled by introducing editing tags to the models: the underlying principle is that a non-injective function \( f : A \to B \) can be augmented into an injective function \( f' : A \to B \times H \), where \( H \) is an history of edits necessary for guaranteeing invertibility properties.

### 3.7 Lenses with view complement

Recent work by the same group focused on the semantics of \textit{view-update under constant complement}. The approach in [MHN'07] considers the construction of an injective function from the view function (\( \text{get} \)) in order to ensure that it is invertible. The view complement function \( \text{get}^c : C \to A' \) is a function from the concrete source to another view that stores the information lost by the view function \( \text{get} : C \to A \) that will be later used for the backward transformation. The constructed injective function corresponds to the tupled function \( \text{get} \Delta \text{get}^c : C \to A \times A' \).

Through bidirectionalization, a backward transformation function is derived from a view function such that the two satisfy the bidirectional properties for acceptability (1.4), stability (1.5) and composability (1.6) for very well-behaved lenses, and the following additional property for undoability:

\[
\forall c \in C, a \in A. \text{put} (\text{get} c, \text{put} (a, c)) \sqsubseteq c \tag{3.16}
\]

Undoability denotes that all concrete updates can be reverted with updates on the abstract view.

Given the previous properties for partial function transformations, we can write the following diagram:

```
\begin{align*}
\text{C} & \xrightarrow{\text{get} \Delta \text{get}^c} \times \text{A'} \\
\text{A} \times \text{C} & \text{put} \quad \text{id} \times \text{get}^c
\end{align*}
```

Here, the inequation \( C \succeq A \times A' \) denotes a lens \( C \triangleright A \) augmented with a complement function \( \text{get}^c : C \to A' \), such that \( \text{get} \Delta \text{get}^c \) is injective. Through the diagram, we get that \( \text{put} : A \times C \to C \) can be defined as \( (\text{get} \Delta \text{get}^c) \circ (\text{id} \times \text{get}^c) \), since \( \text{get} \Delta \text{get}^c \) is invertible.

Apart from the “composed injectivity”, the only requirement on \( \text{get}^c \) is that it provides the information inexistent in \( A \), and can have any counter-domain, permitting a greater freedom in the derivation process. Let, for example, \( \text{add} \) be the transformation expressed by \( \text{add} (x, y) = x + y \). Then, \( \text{add} \) can have more than one complement function

\[
\begin{align*}
\text{fst} (x, y) &= x \\
\text{sub} (x, y) &= x - y \\
\text{id} (x, y) &= (x, y)
\end{align*}
\]

each leading to a different backward function:

\[
\begin{align*}
\text{put} (a, (x, y)) &= (x, a - y) \\
\text{put} (a, (x, y)) &= ((a + (x - y)) / 2, (a - (x - y)) / 2) \\
\text{put} (a, (x, y)) &= (x, y), \text{if} a = x + y
\end{align*}
\]

\(^3\)In the referenced document it is formulated with lens nomenclature and is found under the name \( \text{getput} \).
Such backward functions have different updatability according to the view complement: the more information from the original concrete model preserved by \( get^c \), the less modifications are permitted in the view, because the backward function derived from the view complement function should forbid any changes to the information the complement has kept.

A language is defined to derive the complement functions and the resulting backward functions automatically. The algorithm tries to find the smallest view complement function that is injective, operating on functions by syntactic analysis of the point-wise function definitions. For example, testing the injectivity of some function can be done through pattern-matching and variable inspection. The authors enunciate two cases where a function is non-injective: if variables in the left-hand side disappear in the right-hand side; or if the ranges of two right-hand sides overlap. For example, the function \( fst\ (x, y) = x \) is not injective because the variable \( y \) disappears in the right-hand side. For the backward transformation to be derived, the system still has to be capable of inverting function \( get\triangle get^c \). Consider for instance the injective function:

\[
\begin{align*}
  f\ (A\ x) &= B \\
  f\ (A\ y) &= C
\end{align*}
\]

To calculate the inverse function \( f^o \) of \( f \), we only need to swap the patterns with the results:

\[
\begin{align*}
  f\ B &= A\ x \\
  f\ C &= A\ y
\end{align*}
\]

Although the language allows the definition of isomorphisms from views, it does not support duplication of data: for the case when \( get::A \rightarrow A \times A \), there is no possible complement function that will make \( get\triangle get^c \) injective. Remark that data duplication was possible in the previous language, where \( get \) was assumed to be injective, as described in Section 3.6.

### 3.8 Mathematical properties of lenses

A (stateful) lens can be express in terms of (stateless) abstractions, ie, one \( \triangleright \)-diagram can be subsumed under a group of \( \triangleright \)-diagrams. Oliveira\cite{Oli07} showed that the equations for acceptability (1.3,1.4) and stability (1.5) of a lens meet the connectivity requirements of \( \triangleleft \)-diagrams (or \( \triangleright \)-diagrams, conversely):

Each diagram comprises the properties for valid abstractions that, if held together, guarantee the properties for a well-defined lens. The first diagram resembles a canonizer, as no state is required, and simply certifies that the stateless part of a lens is well-behaved. The second diagram encloses the generation of a stateful abstraction inside a stateless one, where \( \pi_1^o \) is any function that creates some state to drive the backward transformation: it ensures that the stateful part of a lens generates an unique concrete update for each abstract value. At last, the third diagram reverts the stateful part of a lens to a refinement, and ensures that the range of the stateful backward transformation is the whole concrete domain.

From the properties from each diagram, we know that for a lens to be well-behaved:

- \( get \) has to be surjective, from (1.3) and (1.4));
- \( create \) has to be injective and total, from (1.3);
- \( put \circ \pi_1^o \) has to be injective and total from (1.4);
- \( get\triangle id \) has to be injective and total from (1.5);
\* put has to be surjective - requisite of (1.5).

Considering the non-trivial cases, get\(\triangle id\) is always an injective function, even if get is not, because the identity function guarantees an unique result for each argument. For it to be total, get must also be a total function. For put\(\circ \pi_1\) to be total, put must necessarily be a total function, but for it to be injective, put does not have to be an injective function on the whole domain \(A\times C\). As an example, consider the lens \(ldropR \in (A\times B \rightarrow A)\) that removes the second element \(B\) of a binary product \(A\times B\).

\[
\begin{align*}
get &= \pi_1 \\
create &= \text{id}\triangle b \\
put &= \text{id} \times \pi_2
\end{align*}
\]

Clearly, put is not injective since \(\pi_2\) is not \((\text{put} \ (a, (a', b)) = (a, (a'', b)) = (a, b))\), but it still satisfies (1.4) as long as it is injective on the abstract type \(A\). In lens literature [FGM+07], this property is described as the semi-injectivity of put.

\[
\begin{align*}
get (\text{put} \ (a, (a', b))) &= get (a, a') = a \\
get (\text{put} \ (a, (a'', b))) &= get (a, a'') = a
\end{align*}
\]

To warm up, a total lens from \(C\) to \(A\) is well-behaved if there is a surjective function get\(:: C \rightarrow A\) (the abstraction function), an injective function create\(:: A \rightarrow C\) (the stateless representation function) and a surjective and semi-injective function put\(:: A\times C \rightarrow C\) (the stateful representation function).

An interesting property of lenses is obliviousness. In such a case, put can be defined at the cost of create: put\((a, c) = create \ a\). For \(l\) to be well-behaved, it is required by (1.4) that:

\[
\begin{align*}
\forall \ c \in C, \ a \in A. \ get \ (\text{put} \ (a, c)) &= a \\
\iff \ \{ \ l \text{ is oblivious} \} \\
\forall \ a \in A. \ get \ (\text{create} \ a) &= a \\
\iff \ \{ \text{removing variable quantifiers} \} \\
get \circ \create &= \text{id}
\end{align*}
\]

Knowing that, from (1.3), create\(\circ get = \text{id}\), both get and create must be injective and surjective functions and, therefore, bijective functions. Given that, we can conclude that any oblivious lens is an isomorphism, as proved in [Oli07].

### 3.9 A lens library

After studying the recent contributions toward bidirectional transformation scenarios involving lenses, we devote this section to the implementation of a prototype library in Haskell, as an extension to the 2LT Framework. Our implementation directly follows the concepts presented for the 2LT Framework. We define lenses in a type-safe manner, in the sense that lens definition can be constrained to specific types and their validation relies on the type-checker.

Similarly to the data type Ref for refinements, the basic definition of a lens is given by the type Lens:

\[
\text{data } \text{Lens } c \ a = \text{Lens} \{ \text{get} :: c \rightarrow a, \text{put} :: (a, c) \rightarrow c, \text{create} :: a \rightarrow c \}
\]

With this single line of code, we can start defining some traditional lens combinators.

The first example is the identity lens \((id. \in C \rightarrow C)\), that simply copies the concrete format to the abstract domain:

\[
\begin{align*}
id. &= \text{Lens } c \ c \\
id. &= \text{Lens } \text{id} \ \text{fst} \ \text{id}
\end{align*}
\]

Another trivial example are lenses inc\(\in \text{Int} \rightarrow \text{Int}\) and dec\(\in \text{Int} \rightarrow \text{Int}\), that increment/decrement the value of an integer in the forward direction, and decrement/increment that the abstract argument value in the backward direction. The definition is straightforward:
The fundamental principle behind lenses is that views permit the removal of information from a source data type. One of the simplest examples is the lens \( \forall b \in B. \text{dropR} b \in A \times B \succeq A \), that as the name says, drops the second element of a pair. 

\[
\text{dropR} :: b \rightarrow \text{Lens} \ (a, b) \ a
\]

\[
\text{dropR} \ d = \text{Lens \ get'} \ \text{put'} \ \text{create'}
\]

where

\[
\begin{align*}
\text{get'} \ (a, b) &= a \\
\text{put'} \ (a', (a, b)) &= (a', b) \\
\text{create'} \ a &= (a, d)
\end{align*}
\]

In the \textit{get} direction, the lens simply removes the second element of the source pair. In the \textit{put} direction, the left element of the concrete pair is updated with the abstract value, and the right element of the pair is fetched from the old concrete pair. When no old concrete pair is available, \textit{create} inserts the default value \( d \) passed as an argument to \textit{dropR}.

A similar lens to \textit{dropR} is \( \forall c \in C. \text{erase} c \in C \succeq 1 \), that deletes the source type returning the unity type (\( ) \).

\[
\text{erase} :: c \rightarrow \text{Lens} \ c \ ()
\]

\[
\text{erase} \ d = \text{Lens} \ (() \ \text{snd} \ d)
\]

Since our library relies on small transformation steps, the perhaps more important combinator is composition. For lenses, likely to refinements, it is trivial to define:

\[
\text{comp} :: \text{Lens} \ b \ a \rightarrow \text{Lens} \ c \ b \rightarrow \text{Lens} \ c \ a
\]

\[
\text{comp} \ l_1 \ l_2 = \text{Lens} \ \text{get'} \ \text{put'} \ \text{create'}
\]

where

\[
\begin{align*}
\text{get'} \ c &= \text{get} \ l_1 \ (\text{get} \ l_2 \ c) \\
\text{put'} \ (a, c) &= \text{put} \ l_2 \ (\text{put} \ l_1 \ (a, \text{get} \ l_2 \ c), c) \\
\text{create'} \ a &= \text{create} \ l_2 \ (\text{create} \ l_1 \ a)
\end{align*}
\]

Formally, the type for composition is \( \forall l_1 \in B \succeq A, \ l_2 \in C \succeq B. \ \text{comp} \ l_1 \ l_2 \in C \succeq A \). For \textit{get} and \textit{create}, composition means simply composing the value transformations for \( l_1 \) and \( l_2 \). In the \textit{put} direction, the \textit{put} of \( l_2 \) is applied after the \textit{put} of \( l_1 \), that needs a concrete old value of \( B \) to be provided with \textit{get} \( l_2 \).

A more complex combinator is \( \forall l \in C \succeq A. \ \text{list} \ l \in [C] \succeq [A] \), that maps a lens over all elements of a list.

\[
\text{list} :: \text{Lens} \ c \ a \rightarrow \text{Lens} \ [c] \ [a]
\]

\[
\text{list} \ l = \text{Lens} \ \text{get'} \ \text{put'} \ \text{create'}
\]

where

\[
\begin{align*}
\text{get'} \ c &= \text{map} \ (\text{get} \ l) \ c \\
\text{put'} \ ([], \_ \_\_) &= [] \\
\text{put'} \ (a : \text{as}, [] \_\_\_) &= \text{create} \ l \ a : \text{put'} \ (\text{as}, [] \_\_\_) \\
\text{put'} \ (a : \text{as}, \_\_c : \text{cs}) &= \text{put} \ l \ (a, c) : \text{put'} \ (\text{as}, \_\_c \_\_cs) \\
\text{create'} \ a &= \text{map} \ (\text{create} \ l) \ a
\end{align*}
\]

Again, \textit{get} and \textit{create} are constructed by simply mapping the transformations from the argument lens \( l \) to all the elements of the list. However, for \textit{put} to be well-behaved, it needs to guarantee that no abstract information is lost amid the concrete update: when a value exists in both the abstract and concrete list, \textit{put} \( l \) is mapped as normal; if the abstract string is longer than the concrete string, exceeding elements are propagated as concrete updates via \textit{create}; if the concrete string is longer than the abstract string, extra list elements are ignored.
3.9.1 Lenses for relational tables

Inspired by the work presented in Section 3.2, we define some lens combinators for relational tables. However, we restrict the space of database tables to dictionaries, ie, maps from keys to values. This simplification allows us to ignore tree form conditions and many of the intermediate operations used in [BPV06].

In Haskell, a dictionary can be encoded with the Map datatype, found under the Data.Map module. This module provides several combinators for map processing, such as filtering, union, difference, mapping, folding and lookup among others. However, the library does not provide common operations from the relational algebra, such as joins and projections, because the result type of the result tables depend on the types of the argument tables that will be merged or projected. In Haskell we cannot define a function without knowing its result type, making it impossible to define a general join operation. For instance, we could define a function join’::Map a b → Map a c → Map a (b, c), but if the structure of the maps change, we need to define “replicated” combinators such as join''::Map a (b, c) → Map a (b, d) → Map a (b, c, d).

We could explore other data representations of maps, but leave that for future work.

For our library, we will keep concepts simple and define specific instances of join and projection. Note that it is easy to define new combinators for new situations. For the functions not explained here, please consult the documentation.

When defining functions, we will assume a more relational notation. The natural join of two tables M and N is written M▷◁N. For similar binary operations, such as intersection, union and difference, we write M∩N M∪N and M\N, respectively. Projection is also considered. For instance, ΠABM corresponds to selecting the columns with names A and B from the table M.

Selection The first lens we define is selection. When dealing with predicates, we abuse from the notation and state that the intersection P∩M corresponds to selecting all the record from table M that satisfy predicate P. Consider the following definition for select P ∈ M ⊇ P∩M:

\[
\text{select :: Ord k ⇒ (k → a → Bool)} → \text{Lens } (\text{Map } k \ a) (\text{Map } k \ a)
\]

\[
\text{where get' } r = \text{p\cap r}
\]

\[
\text{put' } (s, r) = s \cup (\neg p\cap r)
\]

\[
\text{create'} s = s
\]

The get component of the select p lens performs a relational selection on a table, filtering out the record that do not satisfy the argument predicate p. In the put direction, the lens returns the union of the abstract table with the records from the original table that did not satisfy the predicate. An important detail for the well-behavedness of select is that the M∪N prefers records from M when duplicate keys are encountered.

Projection Projection is another relational operation that can be given bidirectional semantics. We consider a basic lens, called drop, that projects away one single column from a map with one key column and two value columns. More formally, our lens is well-defined for the domain drop c ∈ A → B×C⊇A → B, where c is a default value for the dropped column C.

\[
\text{drop :: Ord k ⇒ c → Lens } (\text{Map } a (b, c)) (\text{Map } a \ b)
\]

\[
\text{where get' } r = \Pi_{a,b}m
\]

\[
\text{put' } (s, r) = r.X =_c s
\]

\[
\text{create'} s = \text{map } (\lambda b → (b, c))
\]

If state is not considered, the forward direction projects away the second column for values and the backward direction reattaches the dropped column and completes the records with default values.

The put semantics are less obvious and correspond to the right outer join of the concrete map R and the abstract map S, considering d a default value that fills missing information (RX =_d S). The right

outer join computes all combinations of \( R \) and \( S \) that are equal on their common attributes, in addition to the tuples in \( S \) that have no matching in \( R \). For this specific application it has the following definition:

\[
X ::= \text{Map } a (b, c) \rightarrow \text{Map } a b \rightarrow c \rightarrow \text{Map } a (b, c) \\
RX = d \ s = \text{mapWithKey} (\lambda a b \rightarrow (b, \text{snd} (\text{findWithDefault} (\bot, c) a) r)) s
\]

\( RX = d S \) as used in the \textit{put} direction of \textit{drop}, maps over the keys of the abstract map, and tries to find an associated value for the dropped column, otherwise the default value is inserted. Consider the next example:

\[
\begin{array}{c|c|c}
\text{R} & \text{S} & \text{RX}\_\text{xy}\_S \\
\hline
A & B & C \\
a1 & b1 & c1 \\
a2 & b1 & c2 \\
a3 & b3 \\
\hline
A & B \\
a1 & b1 \\
a2 & b2 \\
a3 & b3 \\
\hline
A & B & C \\
a1 & b1 & c1 \\
a2 & b2 & c2 \\
a3 & b3 & d \\
\end{array}
\]

For the first row, \( a1 \) exists both in \( S \) and \( R \), and the tuple \((a1, b1, c1)\) is inserted in table \( RX = d S \). In the second row, \( a2 \) exists in both \( S \) and \( R \) again, but \( b2 \) prevails over \( b1 \) in tuple \((a2, b2, c2)\). In the last row, \( a3 \) exists only in \( S \), and therefore a default is assigned for the missing column, generating the tuple \((a3, b3, d)\).

**Join** The natural join of two maps that share the same key attributes is trivial to define. In the following definition, \textit{find} is the function that looks up for the value associated to an argument key, assuming that the key exists in the map.

\[
\triangleright \triangleright \text ::= \text{Ord } a \Rightarrow \text{Map } a b \rightarrow \text{Map } a c \rightarrow \text{Map } a (b, c) \\
\text{a\triangleright b} = \text{foldr} (\lambda k m \rightarrow \text{insert } k (\text{find } d k a, \text{find } d k b) m) \text{ empty } ks \\
\text{ where find} = \text{findWithDefault} \bot \\
ks = \text{List.intersect} (\text{keys } a) (\text{keys } b)
\]

The last relational operation we define is a bidirectional inner join. Although there are many possible combinations for joining two tables, we provide the simplest behavior as a demonstration. The lens \( \text{join} \in (A \rightarrow B) \geq (A \rightarrow C) \) joins two tables sharing the same key column.

\[
\text{join ::= Ord } a \Rightarrow \text{Lens } (\text{Map } a b, \text{Map } a c) (\text{Map } a (b, c)) \\
\text{join} = \text{Lens get’ put’ create’} \\
\text{ where get’ } (a, b) = a\triangleright b \\
\text{ put’ } (s, (a, b)) = (m, n) \\
\text{ where } m0 = \Pi_{a,b}s \cup a \\
n = \Pi_{a,c}s \cup b \\
m = m0 \setminus \Pi_{a,b} l \\
l = m0 \setminus \Pi_{a,c} (n\setminus s) \\
\text{create’ } s = (\Pi_{a,b}s, \Pi_{a,c}s)
\]

The \textit{get} function performs the natural join of the argument tables, while the \textit{create} function creates projections from the joined table into the original tables. The complexity lies in the \textit{put} function. Like \textit{create}, it creates projections from the joined table, but merges them with the original concrete tables. However, this definition is not able to distinguish the deleted rows from the ones that did not exist in the original tables, and in order to guarantee well-behavedness, we simulate the application of \( \textit{put} \circ \textit{get} \) to \( m0 \) in \( l \), that collects the inconsistent values, and remove the records present in \( l \) from \( m0 \), returning the final table \( m \). Since the conflicting records were already removed from \( m \), there is no need to remove them from \( n\mathbf{5} \).

\(^5\text{This comportment is replicated from [BPV06].}\)
3.9.2 Checking for well-behavedness

Note that these definitions simply guarantee that lenses are well-typed, and do not enforce any of the semantic properties for well-behavedness. Additionally, Haskell functions are possibly partial whilst lenses ideally consider total functions. The developer shall guarantee that all possible patterns are covered to improve the safety of his lenses.

In order to provide the developer with more guarantees about the lens he encodes, we will use the Haskell library QuickCheck [CH00] to automatically test lens programs. QuickCheck receives the code of the program and a specification in the form of properties (common boolean functions) that it has to satisfy, and it generates a finite number of random cases to test if the properties hold.

The properties for a well-behaved lens can be expressed in QuickCheck as follows:

\[
\text{getput} :: \text{Eq } c \Rightarrow \text{Lens } c \text{ a } \rightarrow \text{c } \rightarrow \text{Bool} \\
\text{getput } lns \text{ c } = \text{put } lns \text{ ((get } lns \text{ c), c) } \equiv \text{c} \\
\text{putget} :: \text{Eq } a \Rightarrow \text{Lens } c \text{ a } \rightarrow \text{c } \rightarrow \text{a } \rightarrow \text{Bool} \\
\text{putget } lns \text{ c } a = \text{get } lns \text{ (put } lns \text{ (a, c)) } \equiv \text{a} \\
\text{createget} :: \text{Eq } a \Rightarrow \text{Lens } c \text{ a } \rightarrow \text{a } \rightarrow \text{Bool} \\
\text{createget } lns \text{ a } = \text{get } lns \text{ (create } lns \text{ a) } \equiv \text{a}
\]

Each of them receives as arguments a lens to test, concrete and or abstract values that QuickCheck will generate, and returns a boolean (True of False) that says if the lens passed the test or not. The general property for lens well-behavedness is defined by testing the three necessary properties altogether.

\[
\text{wb} :: (\text{Eq a}, \text{Eq c}) \Rightarrow \text{Lens } c \text{ a } \rightarrow \text{c } \rightarrow \text{a } \rightarrow \text{Bool} \\
\text{wb } lns \text{ c } a = \text{getput } lns \text{ c } \land \text{putget } lns \text{ c } a \land \text{createget } lns \text{ a}
\]

Now, let’s perform a test over one of the previously defined combinators. We can write a specific property to test the well-behavedness of the \text{list} combinator:

\[
\text{list-prop } a \text{ c } = \text{wb } (\text{list inc}) \text{ a } c
\]

You should retain two ideas from this definition. First, since the \text{list} combinator receives an argument lens, we need to provide some other combinator (in this case we provide \text{inc}). Since the properties for \text{list} that we want to test are mostly related to the order and length of lists, the specific lens passed to \text{list} is not relevant. However, for more complex scenarios, this can be crucial to defining a good test.

Second, the type of the \text{list} combinator is polymorphic, and may need to be instantiated. For instance, if we had defined \text{list-prop} with the identity lens \text{id} \in A \rhd A that is also polymorphic, a monomorphic type would need to be declared to the property, since QuickCheck cannot “invent” a specific type to test the property:

\[
\text{list-prop}' \text{ a c } = \text{wb } (\text{list } \text{id}) \text{ a } c \\
\text{where types } = \text{c} :: [\text{Int}]
\]

Given this definition, we can test our list combinator; the output of running the reference function \text{quickCheck} for our property is:

\>` quickCheck list_prop
\> OK, passed 100 tests.

We have passed one hundred randomly generated tests. For a better account of the type of errors QuickCheck returns, imagine that we turn \text{list} not well-behaved by ignoring abstract elements that exceed the length of the concrete list:

\[
\text{list'} :: \text{Lens } c \text{ a } \rightarrow \text{Lens } [c] [a] \\
\text{list'} l = \text{Lens } \text{get'} \text{ put'} \text{ create'} \\
\text{where get'} c = \text{map } (\text{get } l) \text{ c}
\]

34
Putting the same property for list', we get the following output:

```haskell
goto (\a, as) = goto (a : as)
goto (a : as, c : cs) = goto (a : cs) : goto (as, cs)
create' a = map (create l) a
```

Running the same property for list', we get the following output:

```haskell
> quickCheck list_prop
Falsifiable, after 3 tests:
[2,-2,-2,2]
[3,-1]
```

The error tells us after how many tests the property failed to hold, and presents the specific example for which it failed. In a closer inspection, it is easy to see that the property failed because the abstract list [3, -1] was smaller than the concrete list [2, -2, -2, 2], and thus the error.

We can also write test properties for our relational lenses. First of all, we need to provide the generation of arbitrary Map values, as expressed in the following instance of class Arbitrary:

```haskell
instance (Ord a, Arbitrary a, Arbitrary b) => Arbitrary (Map a b) where
    arbitrary = liftM Map . fromList $ liftM2 zip arbitrary arbitrary
```

Since repeated keys in the list replace old keys, it is safe to derive a map from a list. For example, `fromList [(0, 1), (0, 2)] = \{0 \mapsto 2\}`.

Encoded the arbitrary generation of values, consider a possible test property for select:

```haskell
p :: Int -> Char -> Bool
p 'a' = False
p _ = True
select_prop a c = wb (select p) a c
    where types = c :: Map Int Char
```

Here, considering predicate p, select p is the lens that filters out, in the forward direction, all records with value 'a'. This lens is proved to be well-behaved:

```haskell
> quickCheck select_prop
OK, passed 100 tests.
```

We can also define tests for drop and join

```haskell
drop_prop a c = wb (drop 'c') a c
    where types = c :: Map Int (Char, Char)
join_prop a c = wb join a c
    where types = c :: (Map Int Char, Map Int Char)
```

and prove that they are valid lenses:

```haskell
> quickCheck drop_prop
OK, passed 100 tests.
> quickCheck join_prop
OK, passed 100 tests.
```

3.9.3 A synchronization example

The example in Figure 3.4 illustrates the synchronization of two schemas corresponding to the address books of a cellphone and a laptop computer.
The most intuitive method to represent an address book is to create a map from an identifier, the name or nickname of a person or entity, to a tuple containing the information associated to the identified element. The information stored in an address book may vary according to the device it is developed for or the application scenario it addresses, but there are in general fields common to all representations, such as phone numbers and e-mail addresses.

Consider the following definitions in Haskell:

```haskell
type Id = String
type Phone = Int
type Mail = String
type Group = String
type Ringtone = String
type IsCompany = Bool

type Cell = Map Id (Phone, Mail, Group, Ringtone)
type Laptop = Map Id (Phone, Mail, IsCompany)
```

A cellphone address book is here represented with the type `Cell`, and is assumed to append a phone number, an e-mail address, a group and a ringtone to an identifier, that is typically the name of a friend.

```haskell
cell :: Cell
cell = { "Diana" ↦→ (96345125, "diaju@mail.com", "Family", "Enchanted")
    , "Hugo" ↦→ (931169956, "hugo@gmail.com", "Friends", "Beez")
    , "Joao" ↦→ (913423451, "joao@bragatel.pt", "Other", "HappySong")
}
```

On the other side, we consider a much simplified laptop’s address book that simply binds a phone number and an e-mail address to a contact, but that permits the distinction between people and enterprises contacts via an extra yes/no tag (it could be a checkbox).

```haskell
laptop :: Laptop
laptop = { "Apple" ↦→ (13134, "support@apple.com", True)
    , "Manuel" ↦→ (931236532, "vilas@quintas.pt", False)
}
```
Given the two representations, it is be straightforward to convert them to an intermediate schema that stores the most relevant information about each contact, that we denote as the schema for synchronization $\text{Sync}$, containing only an identifier, a phone number and an e-mail address.

\[
\text{type } \text{Sync} = \text{Map Id (Phone, Mail)}
\]

The first step for make the synchronization possible is to define two bidirectional transformations to translate between the specific address books and the universal synchronization schema. As long as the last one contains less information than each schema, the transformations are said to be abstractions and therefore lenses.

The lens that transforms a cellphone’s address book into the universal representation is achieved by dropping the fields from $\text{Cell}$ that are not considered in $\text{Sync}$. For this specific case, the lens $\text{drop34of4}$ is employed, denoting the third and fourth elements of a tuple with four elements are removed from the source schema. Although the definition of this lens is not provided, it is straightforward to derive from $\text{drop}$ for relational tables. When dropping information, it is also convenient to provide default information for whenever the backward function has to insert new records in the address book.

\[
\text{lcell} :: \text{Lens Cell Sync}
\]
\[
\text{lcell} = \text{drop34of4} (\text{"None"}, \text{"Default"})
\]

The transformation from a laptop’s address book into the universal representation is similar. However, a selection has to be made on the records to select the ones that correspond to person’s contacts and ignore the contacts for companies, that are not considered in the universal representation. The predicate $\text{isCompany}$ defines the filtering function passed to $\text{select}$ and returns True for every record that does not correspond to a company’s contact.

\[
\text{isCompany} :: \text{Id } \rightarrow (\text{Phone, Mail, IsCompany}) \rightarrow \text{Bool}
\]
\[
\text{isCompany } (\_, \_, b) = \neg b
\]

The overall transformation from $\text{Laptop}$ to $\text{Sync}$ is created by composing $\text{select isCompany}$ with the lens that drops the third element (the tag declaring that the contact denominates to a company or not) from a tuple with three elements.

\[
\text{llaptop} :: \text{Lens Laptop Sync}
\]
\[
\text{llaptop} = \text{comp (drop3of3 False)} (\text{select isCompany})
\]

Defined the lens transformations, we need to define the synchronization policies for each schema. The synchronization of an address book is performed by converting into $\text{Sync}$, merging it the conversion of the other schema to $\text{Sync}$, and putting back the changes to the original schema. If we want to perform a fresh translation from an address book representation to another, we can always apply $\text{create}$ after converting one schema into $\text{Sync}$ to generate a new address book with default information.

For this example, we define that the synchronization process of one schema prefers the information from the other schema when conflicts are encountered. Conflicts result from duplicated information in both representations, and require a decision to be made about how the information from both schemas shall be “mixed”.

To update a an old address book with the information from other address book, the following functions can be invoked, for cellphones, and laptops, respectively:

\[
\text{updateCell} :: (\text{Laptop, Cell}) \rightarrow \text{Cell}
\]
\[
\text{updateCell} (l, c) = \text{put lcell ((get llaptop l) \cup (get lcell c), c)}
\]
\[
\text{updateLaptop} :: (\text{Cell, Laptop}) \rightarrow \text{Laptop}
\]
\[
\text{updateLaptop} (c, l) = \text{put llaptop ((get lcell c) \cup (get llaptop l), l)}
\]

To complete our example, we want to synchronize $\text{cell}$ by considering the information from $\text{laptop}$. The result is the new cellphone’s address book.
updateCell (laptop,cell)

cell' = { "Diana" ↦→ (96345125,"diaju@mail.com","Family","Enchanted")
, "Hugo" ↦→ (931169956,"hugo@gmail.com","Friends","Beez")
, "Joao" ↦→ (913423451,"joao@bragatel.pt","Other","HappySong")
, "Manuel" ↦→ (931236532,"vilas@quintas.pt","None","Default")
}
The contact for Manuel has been updated into the new cellphone's address book and default information was appended to the new fields Group and Ringtone.

The opposite synchronization can also be performed. If updating laptop with the information from cell we get:

updateLaptop (cell,laptop)

laptop' = { "Apple" ↦→ (13134, "support@apple.com", True)
, "Diana" ↦→ (96345125, "diaju@mail.com", False)
, "Hugo" ↦→ (931169956, "hugo@gmail.com", False)
, "Joao" ↦→ (913423451, "joao@bragatel.pt", False)
, "Manuel" ↦→ (931236532, "vilas@quintas.pt", False)
}
The greatest difference is that the contact for Apple is preserved, since it is valid in a laptop’s address book. The contacts for Diana, Hugo and Joao were also inserted, with tag False denoting that they are not company contacts.

Note that the due to our specified update policies (the information from the “other” schema always prevails), two consecutive updates should produce the same output as a single one, what is in fact true.

updateLaptop (cell,updateLaptop (cell,laptop))
laptop'' = { "Apple" ↦→ (13134, "support@apple.com", True)
, "Diana" ↦→ (96345125, "diaju@mail.com", False)
, "Hugo" ↦→ (931169956, "hugo@gmail.com", False)
, "Joao" ↦→ (913423451, "joao@bragatel.pt", False)
, "Manuel" ↦→ (931236532, "vilas@quintas.pt", False)
}

3.9.4 A rewrite system for lenses

We have defined a core library for lenses in Haskell. The next step is to extend it with a rewrite system for implementing strategies based on bidirectional lens transformations. We provide the definitions for a type-changing rewrite system in Haskell for lenses that is a pure re-implementation of the previous one for refinements.

Given a lens over values of specific types, we may add type-awareness to a Lens as a Rep over types

data View c where
  View :: Lens c a → Type a → View (Type c)

and abstract a View into a generic Rule, partial, that is type-changing.

type Rule = ∀ c. Type c → Maybe (View (Type c))

Creating generic rules from specific lens combinators is also straightforward.

38
For the id lens, we can write the most trivial rule:

\[
\text{nop :: Rule}
\]
\[
\text{nop = return \circ \text{View id}}
\]

Following the same concepts, the composition of lens rules can be defined as follows:

\[
(\gg) :: \text{Rule} \rightarrow \text{Rule} \rightarrow \text{Rule}
\]
\[
(r \gg s) c = \text{do}
\]
\[
\text{View } g \ b \leftarrow r \ c
\]
\[
\text{View } f \ a \leftarrow s \ b
\]
\[
\text{return } (\text{View } (\text{comp } f \ g) \ a)
\]

More generic combinators similar to the ones for refinements are defined:

\[
\text{many :: Rule} \rightarrow \text{Rule}
\]
\[
\text{once :: Rule} \rightarrow \text{Rule}
\]

A rule for the lens combinator erase can also be provided:

\[
\text{eraser :: Type } a \rightarrow a \rightarrow \text{Rule}
\]
\[
\text{eraser } a \ x \ b = \text{do}
\]
\[
\text{Eq} \leftarrow \text{teq } a \ b
\]
\[
\text{return } (\text{View } (\text{eraseL } x) \ One)
\]

\[
\text{dropOne can be used to remove the unity types generated by erase. dropOnes is a rewrite strategy}
\]
\[
\text{that applies dropOne many times.}
\]
\[
\text{dropOne :: Rule}
\]
\[
\text{dropOne } (\text{Prod } a \ One) = \text{return } (\text{View } (\text{Lens } \text{fst} (\lambda x,_) \rightarrow (x,()))) \ a
\]
\[
\text{dropOne } (\text{Prod } One \ a) = \text{return } (\text{View } (\text{Lens } \text{snd} (\lambda x,_) \rightarrow (((),x))) \ (\lambda x \rightarrow (((),x))) \ a
\]
\[
\text{dropOne } \_ = \text{mzero}
\]
\[
\text{dropOnes :: Rule}
\]
\[
\text{dropOnes = many } (\text{once } \text{dropOne})
\]

Notwithstanding, the situation gets trickier if we need to provide specific default values to generic rules. Consider, for example, the case for the dropR :: b \rightarrow \text{Lens } (a,b) \ a lens combinator. If we are writing a generic rule \text{rdropR}, the naive tendency is to write the following definition:

\[
\text{rdropRWrong :: b \rightarrow Rule}
\]
\[
\text{rdropRWrong } d \ (\text{Prod } a \ b') = \text{return } (\text{View } (\text{dropR } d) \ a)
\]

However, this definition does not compile, since the type checker cannot assure that type \text{b} (the type for \text{d}) is equal to the type represented in \text{b}'. Remark that this situation would never occur for refinements: if we are adding type information, we know the specific type that we are adding and can provide default values for it; the problem only exists if we want to remove structure inside rules, where we only know which type it will be on-the-fly when computing the type transformation.

The solution is to provide a definition for generic default values. These are encoded as generator functions that create a value from a type definition.

\[
\text{type Default = } \forall a. \text{Type } a \rightarrow a
\]

An example default generator can create values for different types:

\[
\text{gen :: Default}
\]
\[
\text{gen Int = 0}
\]
\[
\text{gen } (\text{List } \_) = []
\]
\[
\ldots
\]
Using this definition, we can now define $r_{dropR}$:

$$
\begin{align*}
    r_{dropR} :: & \text{Default} \to \text{Rule} \\
    r_{dropR} d (\text{Prod} \ a \ b) = & \text{return} (\text{View} (r_{dropR} (d \ b)) a)
\end{align*}
$$

Automatically-driven transformation strategies are best known for data mapping scenarios between different programming approaches. For instance, when translating XML schemas into SQL schemas, recursive and hierarchical structures must be transformed into key references and flat relational tables. All the scenarios to which lenses are traditionally applied involve the synchronization of redundant data models, operationalized in an intermediate representation to which both can be transformed. These require the user to know the destination type of the rewriting process. An algorithm can be inferred for automatically transform the meta-models for both models until they reach some type that is common to both, but it is hard to guarantee that the solution is deterministic, and how computationally expensive it will be.

On the other side, if refinements are useful to descend into software layers, lenses can be seen in the opposite direction, to create simplified high-level views of complex concrete implementations.

As a simple example of how lenses rewrite strategies can be composed and applied, consider the following tree composed from Int and String types, and a strategy $ints$ that removes all the String types from the a source type. $treeC$ is a possible concrete string.

\[
tree = \text{Prod Int} (\text{Prod} (\text{Prod String Int}) (\text{Prod (Prod Int (Prod Int String)) String}))
\]

\[
ints = \text{many (once (erase String "none")) } \bowtie \text{ dropOnes }
\]

\[
treeC = (1, (("Hugo", 2), ((3, (4, "ZeZe"))), "Andre")))
\]

We can create a view resulting from applying the transformation $ints$ to $tree$, resulting in the type $abstree$, with no strings:

\[
treeview = \text{fromJust (ints tree)}
\]

\[
abstree = \text{Prod Int} (\text{Prod Int (Prod Int Int))}
\]

After, an abstract tree $treeA$ can be computed from the concrete tree $treeC$, edited to the tree $treeA'$, for which the changes can be put back into the updated concrete tree $treeC'$.

\[
\begin{align*}
    \text{view} :: & \text{View c} \to \text{Type a} \to c \to \text{Maybe a} \\
    \text{updt} :: & \text{View c} \to \text{Type a} \to a \to c \to \text{Maybe c} \\
    treeA = & \text{view treeview abstree treeC} \\
    = & \text{Just (1, (2, (3, 4)))} \\
    treeA' = & (1, (5, (3, 4))) \\
    treeC' = & \text{updt treeview abstree treeA' treeC} \\
    = & \text{Just (1, (("a", 5), ((3, (4, "b")), "c")))}
\end{align*}
\]

Here, $\text{view}$ and $\text{updt}$ are the Haskell functions used to apply the forward and backward transformations of a view to input types and values.
Chapter 4

Towards mixing refinements and abstractions

When performing a bidirectional transformation over a data model, often is the case where we need to perform a transformation that contradicts the data flow of the general bidirectional transformation (adding data in abstractions or removing data in refinements). Typical examples involve the creation of an abstraction capable of duplicating data or, conversely, a refinement capable of removing duplicated data.

The problem is that a valid refinement is not a well-behaved abstraction unless it is an isomorphism, and vice-versa. Consider the refinement \( \text{dupR} \) that duplicates information to the right:

\[
\begin{array}{c}
\text{id} \triangle \text{id} \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
a \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
\leq \\
\pi_1 \\
\pi_1
\end{array}
\begin{array}{c}
A \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
A \times A \\
\downarrow \\
\downarrow
\end{array}
\]

and is clearly well-behaved, and an attempt to encode a lens \( \text{dupR}' \) with the same behavior:

\[
\begin{array}{c}
\text{id} \triangle \text{id} \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
a \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
\geq \\
\pi_1 \\
\pi_1
\end{array}
\begin{array}{c}
A \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
A \times A \\
\downarrow \\
\downarrow
\end{array}
\]

Since \( \text{create} \) is not injective, it is not a valid abstraction. Through the connectivity of \( \geq \)-diagrams, for two distinct values \( a, a' \in A \):

\[
\begin{array}{l}
\text{get (create} (a, a')) = (a, a') \\
\iff \\
\{ \text{create} = \pi_1 \} \\
\{ \text{get} = \text{id} \triangle \text{id} \} \\
\{ (a, a) = (a, a') \} \\
\{ \text{equality} \} \\
\text{FALSE}
\end{array}
\]

For the opposite attempt, given a well-behaved lens \( \text{dropL} \) that removes the left element of a pair with duplicated data

\[
\begin{array}{c}
\text{id} \triangle \text{id} \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
a \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
\geq \\
\pi_2 \\
\pi_2
\end{array}
\begin{array}{c}
A \times A \\
\downarrow \\
\downarrow
\end{array}
\begin{array}{c}
A \\
\downarrow \\
\downarrow
\end{array}
\]

a similar refinement \( \text{dropL}' \), is also not valid, since \( \text{to} \) is not injective (\( \text{to} (a, a) = \text{to} (a', a) = a \)):
We can enumerate different changes to transformation properties that would allow this interplay: by allowing to and create to be partial, by weakening the connectivity requirements, by extending the type system beyond regular types, by constraining bidirectional transformations to well-behaved domains [ASV07], or by loosening transformation domains module value equivalences [FPP08]. Given that all the others compromise, to some degree, the semantic foundations of bidirectional two-level transformations, we will focus on the last proposals and debate on the implications they comprise.

4.0.5 Type extensions and dependent types

For two-level transformations, although the existing distinction between type and value transformations would normally make us state that they are independent (meaning that they can be evaluated separately), that is not always true. The “violation” happens if we consider the case for dependent types, namely types that depend on values, allowing invariants to be captured by types. Since regular functional languages do not support the declaration of algebraic data types depending on values, dependent types find application in the design of experimental functional languages like Dependent ML [XP99] and formal proof assistants like Coq [DFH+93]. Classical examples are: vectors with \( n \) elements - \([A]^n\); and the subset of the type of integers \( \text{Int} \) that contains a single value \( i \) - \( \{i\} \). In Dependent ML, \([a]^n\) can be defined as follows.

```plaintext
datatype 'a list with nat =
    nil (0)
  | { n : nat } cons (n + 1) 'a * 'a list (n)
```

The previous code can be read as follows: define the datatype \('a list (n)\), where \( a \) is a polymorphic type argument (annotated with an apostrophe) and \( n \) a natural number that determines the size of the list. The list has constructors \( \text{nil} \) for the empty list and with size 0, and \( \text{cons} \) that inserts an element to the head of the list and increments the size by one.

Consider the diagrams from Figure 4.1. If types \( A \) and \( B \) are independent from values \( a \) and \( b \), then we can perform both the type transformation \( T \) and the pair of value transformations \((f, g)\) independently, in an asynchronous way. Conversely, if \( A \) and \( B \) are possibly dependent on \( a \) and \( b \), the conditions \( i \) and \( j \) must be tested before and after evaluating the value transformations \( f \) and \( g \), to render certain that \( a \in A \) and \( b \in B \). Such a system is said to run in \text{lockstep} mode, in a remark to the synchronized transformation imposed on the type and value transformations: transformation is progressed step-wise from one well-defined state to the next well-defined state.

For the lenses we have seen in Chapter 3, the type systems were specific to the application domains (regular expressions for strings and sets of trees for trees) and deeply bound to the concrete data they
are dealing. As long as the type system is able to distinguish value details, it is a value dependent type system. Although the evaluation of transformations has to run in lockstep mode, the type system is more powerful in what it can distinguish and can assign extremely strict domains to transformations.

On the other side, the type system for the 2LT Framework is more general and has the expressive power inherent to Haskell inductive data types, that are value independent. This has the natural advantage to be independent from specific application domains, since it provides an universal representation of types. Also, type transformations can be executed at once and before the value transformations are computed. However, when it comes to fine structure details, value-awareness at the type-level provides a better control of the transformation properties.

In the next sections, we present different type extensions that naturally arise from the required balance between strictness and expressiveness in a type system.

4.0.6 Considering type invariants

The most crucial detail in the design of bidirectional transformation system are the properties of the backward transformations; for the perfect scenario, we want transformations to preserve information and, thus, guarantee that the backward transformation completely undoes application of the forward transformation.

Although bidirectional languages are generally built for less restrictive scenarios that do not preserve information, there is always, at some extent, a concern about the reversibility of transformations, either on the design of an invertible language [MHT04a], or for preserving important structural information such as referential integrity constraints in schema refinements [ASV07].

In general, incoherent transformations can be assigned stricter domains for which they are well-behaved. Recapitulating $dupR'$ as an abstraction with types $C \geq C \times C$, we can prove that it is well-behaved for a stricter codomain $A = \{(c, c) \mid c \in C\}$, denoting $A$ as the subset of $C \times C$ for which both elements contain same value.

A different notation is to express the same constraint as a type invariant, being $A = (C \times C)_{\pi_1=\pi_2}$. Here, $\pi_1 = \pi_2$ can be read as $\forall x \in (C \times C). \pi_1(x) = \pi_2(x)$.

In fact, the invariant is guaranteeing that no information is added in the type transformation and holding an isomorphism $C \cong (C \times C)_{\pi_1=\pi_2}$ that comprises both $dupR'$ and a constrained $dropL'$ that does not drop information: $(C \times C)_{\pi_1=\pi_2} \leq C$.

Still, the 2LT library imposes that application of transformations is value-independent, meaning that transformation domains cannot be restricted to specific inhabitants of some type.

Therefore, type invariants can be seen in two ways:

- to define a type system that handles some type $T$ and $T_{inv}$ (where $inv$ is an invariant) as distinct types and invariant evaluation is made by the type checker. The difference between types $T$ and $T_{inv}$ shall be no smaller than the one between $Int$ and $Char$.
- to allow exception behavior in the value mappings if they consume or generate values that violate invariants. This technique denotes partiality on the encoding, but is more flexible, because it permits values to be inconsistent during updates. For example, if we are duplicating information with $dupR$, the codomain of the abstraction requires the two values to be the same. If the invariant was checked whenever values are created, $(c, c')$ would not be a possible value, but as long as we just check the invariants only when applying value mappings, another update may change $(c, c')$ into a compliant value such as $(c', c')$. After an update, values violating the invariants will trigger the corresponding exceptions.

4.0.7 Considering value equivalences

Although quotient lenses were developed to prevent lens transformations to explicitly handle irrelevant details in the data formats, the study in [FPP08] reports what is said to be an unexpected benefit towards the handling of less strict transformations. In an opposing direction to type invariants, the types of a transformation can be relaxed with equivalence relations until they are insensible to the misbehaved parts of the transformation.
An example is the quotient lens $qdupR \in A/\approx (A \times A)/\approx^{\star \text{Tot}(A)}$ that duplicates information to the right:

$\begin{align*}
\text{get} &= \text{id}\triangle \text{id} \\
\text{create} &= \pi_1 \\
\text{put} &= \pi_1 \circ \pi_1
\end{align*}$

In the previous diagram, the combinator $\star$ lifts $\times$ to equivalence relations. If $R$ and $S$ are equivalence relations on $A$ and $B$, the equivalence relation $R \star S$ is expressed as:

$$(a, b)(R \star S)(a', b') \iff aR a' \land bS b'$$  \hspace{1cm} (4.1)

By saying that all the elements on the right side of the abstract type $A \times A$ are equivalent, we are instructing $qdupR$ to ignore duplicate values. From the acceptability requirements of $\text{get}$ and $\text{put}$:

\[
\begin{align*}
\text{get} (\text{put} ((a', a''), a)) &= (\approx^{\text{Tot}(A)}) (a', a'') \\
\text{get} (a') &= (\approx^{\text{Tot}(A)}) (a', a'') \\
(a', a') &= (\approx^{\text{Tot}(A)}) (a', a'') \\
a' &= a' \land a'' \text{ Tot}(A) a''
\end{align*}
\]

The advantage of loosening the types instead of restricting them is that we do not need to check for invariants when applying the transformations, but instead checking equivalences when testing the well-behavedness of the bidirectional transformations. However, it still feels like a less elegant and intuitive method for expressing type restrictions. For the case of $ldupR$, the original or the added type are treated according to different equivalences, what would lead to differently coarsened properties for composed lenses, depending on whether we choose the left or right element of the pair. This asymmetric treatment is not shared by the type invariant for $dupR'$, that constrains both elements of the pair equally; selecting a single element of the duplicated pair still enforces the invariant to hold for the pair.

Semantically, both type invariants and value equivalences are limiting type information (stating the some type $X$ can only contain one inhabitant $x$ strangely resembles stating that all inhabitants $x$ of $X$ are equivalent - they both impose a unique distinct value for $X$) and can be considered similar.

### 4.1 A lens library with type invariants

In this section, we will extend the previous lens library to handle a simple encoding of type invariants. In Haskell, an invariant for some type can be seen as a monadic function that tests value membership:

\[
\text{inv} :: \text{MonadPlus } m \Rightarrow a \rightarrow m a
\]

We will consider the monad $E$, that complements an argument type with an optional error message as a string.

\[
\text{type } E = \text{Either String}
\]

Since we are dealing with lenses, invariants are more likely to constrain the target schema, for instance, when we want to express refinements as well-behaved lenses. We will only consider invariants for the abstract domain. For simplicity reasons, we also do not consider the propagation of source invariants to invariants on the target structures. For more information on such mechanisms please refer to [ASV07]. Extending the $\text{Lens}$ representation to include an invariant, we get the following definition:

\[
\text{data } \text{Lens } c a = \text{Lens}\{\ldots, \cdot :: a \rightarrow E a\}
\]
If we apply the invariants to all transformations, we need to convert those to the $E$ monad:

$$getI :: \text{Lens } c \ a \to c \to E \ a$$
$$getI \ l = l \circ \text{get } l$$
$$createI :: \text{Lens } c \ a \to a \to E \ c$$
$$createI \ l = \text{liftM} (\text{create }) \circ \ l$$
$$putI :: \text{Lens } c \ a \to (a, c) \to E \ c$$
$$putI \ l \ (a, c) = \text{liftM} (\lambda x \to \text{put } l \ (x, c)) \ l_a$$

For a lens that is already well-behaved we simply skip the invariant and always return the argument value:

$$\text{dropR} :: b \to \text{Lens } (a, b) \ a$$
$$\text{dropR} \ d = \text{Lens} \ldots \ldots \text{return}$$

However, consider now the definition for $\text{list}$, a lens combinator for mapping an argument lens over the elements of a list, with invariants. Although $\text{list}$ is already a well-behaved lens, we can handle invariants in an interesting way:

$$\text{list} :: \text{Lens } c \ a \to \text{Lens } [c] \ [a]$$
$$\text{list} \ l = \text{Lens} \ldots \ldots \text{inv'}$$
  where $\text{inv'} = \text{foldM} (\lambda x \ y \to \text{liftM} (r+) (\text{catchError} (\text{liftM} ([][]) \ b_h) \ \text{handler})) []$
  handler $e = \text{return} []$

With the above invariant, the $\text{list}$ combinator becomes more intelligent. In the put direction, if a value violates the invariant, we can safely ignore it, as long as the concrete and abstract types are monoids\(^1\) (in terms of code, it would make sense to issue a warning for ignored values. In Haskell these could be stored in a $\text{Writer}$ monad for instance). This is expressed using the class method $\text{catchError}$ from the $\text{MonadError}$ class.

### 4.1.1 Well-behaved lenses modulo invariants

As we now support type invariants, the properties for well-behavedness of lenses must also work on the $E$ monad:

$$\text{wbI} :: (\text{Eq } a, \text{Eq } c) \Rightarrow \text{Lens } c \ a \to a \to c \to \text{Bool}$$
$$\text{wbI} \ lns \ a \ c = \text{wbE} (\text{do } x \leftarrow \text{getputI} \ lns \ c \ y \leftarrow \text{putgetI} \ lns \ a \ c \ z \leftarrow \text{creategetI} \ lns \ a \ \text{return} \ (x \land y \land z))$$

$$\text{wbE} :: E \ \text{Bool} \to \text{Bool}$$
$$\text{wbE} (\text{Left } _) = \text{True}$$
$$\text{wbE} (\text{Right } b) = b$$

$$\text{getputI} :: \text{Eq } c \Rightarrow \text{Lens } c \ a \to c \to E \ \text{Bool}$$
$$\text{getputI} \ lns \ c = \text{do } a' \leftarrow \text{getI} \ lns \ c \ c' \leftarrow \text{putI} \ lns \ (a, c) \ \text{return} \ (c' \equiv c)$$

$$\text{putgetI} :: \text{Eq } a \Rightarrow \text{Lens } c \ a \to a \to c \to E \ \text{Bool}$$
$$\text{putgetI} \ lns \ a \ c = \text{do } a' \leftarrow \text{getI} \ lns \ c' \ a' \leftarrow \text{getI} \ lns \ c' \ \text{return} \ (a' \equiv a)$$

$$\text{creategetI} :: \text{Eq } a \Rightarrow \text{Lens } c \ a \to a \to E \ \text{Bool}$$

---

\(^1\)A monoid is a type with a single, associative binary operation and an identity element.
createget lns a = do c′ ← create lns a
a′ ← get lns c′
return (a′ ≡ a)

When a randomly generated value violates a type invariant and an error is returned, we simply ignore it, as encoded in \( wbE \).

### 4.1.2 Conditional lenses

An if-then-else statement is the primordial conditional combinator. For it to be a valid lens, given a predicate \( p \), it must transform a concrete type \( A \) into distinct types \( A \in p \) and \( A / \in p \) in order to guarantee that the backward transformation is injective. Since predicate selection is an injective function, the lens conditional combinator \( \forall p \in (A \rightarrow \text{Bool}) \). \( p \in (A \equiv (A \in p + A / \in p)) \) is an isomorphism.

In terms of encoding in Haskell, note that if the abstract values violate the type invariants, an error is launched.

\[
predicate :: (c \rightarrow \text{Bool}) \rightarrow \text{Lens c} (\text{Either c c})
predicate p = \text{Lens get'} \ put' \ create' \ inv'
\]

```
where get' a = (p a)
put' (a, c) = create a
create' (Left a) = a
create' (Right a) = a
inv' (Left a) = if (p a)
    then (return (Left a))
    else (throwError "left value must satisfy predicate")
inv' (Right a) = if (p a)
    then (throwError "right value must not satisfy predicate")
    else (return (Right a))
```

Suppose the lens \( \text{predicate } (<3) \in (\text{Int} \equiv \text{Int}_{<3} + \text{Int}_{\geq 3}) \). If we execute a backward transformation for which the abstract value violates the abstract domain, we get an invariant exception:

```
> putI (predicate (<3)) (Right 2,1)
Left "right value must not satisfy predicate"
> putI (predicate (<3)) (Left 4,1)
Left "left value must satisfy predicate"
```

To ensure that our lens is well-behaved, we can write some property

\[
\text{predicate_prop a c = wb (predicate (<3)) a c}
\]

```
where types = c :: Int
```

and evaluate it with QuickCheck:

```
> quickCheck predicate_prop
OK, passed 100 tests.
```

Continuing, imagine a similar lens over lists such that \( \text{list } (\text{predicate } (<3)) \in (\text{Int} \equiv [\text{Int}_{<3}] + [\text{Int}_{\geq 3}]) \). In this case, since the list accepts empty values, we may filter out the elements that are violating the abstract domain and prevent the exception messages:

```
> putI (list (predicate (<3))) ([Left 1,Right 2],[1,2])
Right [1]
> putI (list (predicate (<3))) ([Right 0,Left 1,Right 2],[1,2])
Right [1]
> putI (list (predicate (<3))) ([Left 1,Left 2],[1,2])
Right [1,2]
```
Note that this behavior is completely type-safe, since in the best scenario the incoherent data shouldn’t even be accepted by the language. This method can be exploited for any lens for which the invariants are placed on types that are monoids.

Since the type system is independent from the values, it is advisable to avoid declaring type invariants when not strictly necessary because they will still trigger undesirable run-time error situations. The `predicate` lens will be used as the basilar stone to implement conditional lens combinators in our library: by converting a conditional statement into a sum of types and declaring conditional lenses on sums rather than on value-dependent domains will guarantee that all are well-behaved lenses without the need to use invariants.

**Concrete conditional** Recall the conditional combinators from [FGM+07]. In Harmony, the type system is based on sets of values and therefore strict domains can be bound to conditional lenses by performing set operations such as intersection. The concrete conditional over a sum of types that denotes predicate selection is now expressed as \( \forall l_1 \in C_1 \supseteq A, l_2 \in C_2 \supseteq A. \ ccond \in C_1 + C_2 \supseteq A. \)

The Haskell encoding is as follows:

```haskell
  ccond :: Lens a c -> Lens b c -> Lens (Either a b) c
  ccond l1 l2 = Lens get' put' create' return
  where get' (Left x) = get l1 x
        get' (Right x) = get l2 x
        put' (x, Left  y) = Left  (put l1 (x, y))
        put' (x, Right y) = Right (put l2 (x, y))
        create' x = Left (create l1 x)
```

From this definition, we can read that: in the `get` direction, if the concrete value is a left alternative (belongs to the predicate) `get l1` is applied, otherwise it chooses `get l2`; in the `create` direction, no concrete value is available and so we choose to apply `create l1`, but `create l2` is still valid since `l1` and `l2` share the same abstract domain; in the `put` direction, if the concrete value belongs to the predicate, `put l1` is applied, otherwise `put l2` is.

Notwithstanding, defining the conditional over sums does not restrict its expressivity. By composition with `predicate`, we can achieve the same behavior as with the original `ccond` combinator from [FGM+07], but in a more elegant and controlled implementation.

Consider the following example for a lens that increments an integer if it is smaller than 1, and decrements it otherwise:

```
  ex1 = ccond inc dec `comp' predicate (<1)
```

```
> getI ex1 0
Right 1
> getI ex1 2
Right 1
> putI ex1 (5,1)
Right 6
```

For the above examples, 0 is smaller than 1 and is incremented, 2 is greater than 1 and is incremented, and in the backward direction, the concrete value 1 is equal to 1, and therefore 5 is viewed as the decrement of 6.

**Abstract conditional** The abstract conditional \( \forall l_1 \in A \supseteq B, l_2 \in C \supseteq D. \ acond \in A + C \supseteq (B + D) \) can be defined according to the same principles:

```haskell
  acond :: Lens a b -> Lens c d -> Lens (Either a c) (Either b d)
  acond l1 l2 = Lens get' put' create' return
  where get' (Left x) = Left (get l2 x)
```

47
In the get direction, acond copies ccond, but in the backward direction it makes the decision on whether to apply \(l_1\) or \(l_2\) based on an abstract predicate. The put transformations whose concrete argument was populated with \(\Omega\) are now replaced by invocations to \(create\).

We can also compose acond with predicate:

\[
\text{ex2} = \text{acond inc dec 'comp' predicate (} < 1 \text{)}
\]

For the above examples, 0 is smaller than 1 and so it is incremented and put to the left, 2 is greater than 1 and so it decremented and put to the right, \(Left\ 5\) is on the left and so it is decremented by the inverse of \(inc\), and \(Right\ 5\) is on the right and so it is incremented by the inverse of \(dec\).

The difference in results is because acond splits both the concrete and the abstract domains, where ccond only splits the concrete domain. For ccond, the decision on whether \(l_1\) or \(l_2\) is applied in the backward direction is based on the concrete argument. For acond, the disjoint abstract domain makes that selection explicit: abstract values on the left side are putted back through \(l_1\), or via \(l_2\) otherwise.

**List filtering** Another interesting but difficult to express conditional lens is the combinator for lists that filters out all the elements that do not satisfy a predicate. Assuming again that predicate shall do the “dirty work”, list filtering can be seen as a lens from a list of alternatives to a single alternative that is the chosen one: \(\text{filterL} \in [A + B] \rightarrow [A]\) as the name suggests, is filtering the left alternatives from a list. The definition is as follows:

\[
\text{filterL} \::= \text{Lens} [\text{Either a b}] [a]
\]

\[
\text{filterL} = \text{Lens get' put' create' return}
\]

where get' [] = []

get' (Left \(x:ls\)) = x : get\_l
get' (Right \(x:ls\)) = get\_l
put' (l,[]) = map Left l
put' ([],Left \(x:xs\)) = put\_l ([],xs)
put' (l,Right \(x:xs\)) = Right x : put\_l (l, xs)
put' (y:ys,Left \(x:xs\)) = Left y : put\_l (ys, xs)
create' a = map Left a

In the get direction, filterL keeps the values that correspond to left-side alternatives and discards the rest. In the create direction, the abstract values are copied directly from the abstract list to the left sides of the concrete list. However, the put direction of filterL is harder to reason about because it should preserve the original concrete order: if the abstract list contains more elements than the concrete one, then create is applied to them; if the concrete list is longer than the abstract list, values are simply
discarded; if an element in the abstract list is matched with another in the concrete list that belongs to the predicate, then the concrete value is updated; otherwise, if the concrete value does not belong to the predicate, it remains intact and the abstract element is not consumed.

Considering the habitual composition with \textit{predicate}, we can filter the elements of a list that are smaller than 1:

\[
\text{ex3} = \text{filterL \text{'}\text{comp}\text{' list (predicate (<1))}
\]

\begin{verbatim}
> getI ex3 [-1,0,1,2]
Right [-1,0]
> putI ex3 ([1,2],[0,2,3])
Right [1,2,3,2]
\end{verbatim}

For the first example, the list \([-1,0]\) is returned and contains the elements in smaller than 1. In the \textit{put} direction, the values from the abstract list are merged with the values from the concrete list that greater or equal than one (note that 0 is not in the resulting list).

\textbf{Relational selection}  Previously (Chapter 3), we have defined a lens combinator for the selection of records in a database table. Recalling the definition of \textit{select} \( P \in M \geq P \cap M \), the abstract domain disallows the abstract table to include records that do not satisfy the predicate \( P \). In order to guarantee such constraint, we introduce a type invariant:

\[
\text{select} \colon \text{Ord} \ k \Rightarrow (k \to a \to \text{Bool}) \to \text{Lens (Map k a) (Map k a)}
\]

\[
\text{select} \ p = \text{Lens} \ldots \ldots \text{inv'}
\]

\[
\begin{align*}
\text{where} \ & \text{inv'} \ m = \text{if} \ (\text{null} \ (\neg \ p \cap m)) \\
& \text{then} \ \text{return} \ m \\
& \text{else} \ \text{throwError} \ "\text{abstract table must satisfy predicate}"
\end{align*}
\]

It filters the records of the abstract table that satisfy the predicate, and checks if the resulting table is empty; if it is, then the whole abstract table satisfies the predicate and the lens is well-behaved, otherwise an exception is returned.
Chapter 5

Generical Model Transformations

In this section we discuss the case of model transformations that are symmetric, this is, stateful in both directions, in opposition to refinements, that are non-stateful, and lenses, that are stateful only in the backward direction.

A dually stateful transformation between a pair of meta-models $M \times N$ can be expressed as a relation $R \subseteq M \times N$, such that the following diagram holds [Ste07]:

$$
\begin{array}{c}
\begin{array}{c}
M \\
\downarrow R \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\downarrow \pi_2 \\
N \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\uparrow \pi_1 \\
M \times N \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\swarrow \to R \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\searrow \end{array}
\end{array}
\end{array}

Here, $\pi_1$ and $\pi_2$ denote the states for the transformations: model $N$ is the state for $\rightarrow R$; and model $M$ is the state for $\leftarrow R$. In fact, the models that represent the state cannot be retrieved either from the source or target models: for instance, we can go from $M \times N$ to $M$ through $\pi_1$, but can never go from $M$ to $M \times N$ via $\pi_1$, since $\pi_1$ is not injective.

The smallest example, applied to the context of graphical user interfaces [Mee98], is the case of two rulers, where the relation between them is equality, forcing them to have the same value. Thus, if the value of one ruler is changed, it shall be propagated to the other ruler. Note that this example denotes a bijective relation, but that must not necessarily happen. Consider, for instance, the relation $\leq$ between two numeric objects $A$ and $B$. If $a \in A$ has value 4 and $b \in B$ has value 5, if we change $a$ to 6, then different values $b' \in B$ can replace $b$ and maintain $a \leq b'$, such as 6, 15, 100. The question here is which value to choose. One solution is to opt for the more intuitively natural solution in terms of user expectations; in this example, it clearly is $b' = 6$.

biXid Another option is to “live” with the ambiguity inherent to having many-to-many relations between the models in a non-deterministic way. A naturally ambiguous approach is the biXid language for bidirectional XML transformation [KH06]. It adopts the programming-by-relation paradigm, where the user specifies the relations between the models and the language derives the transformations such that they satisfy the relation. The authors identify ambiguity to arise in three possible scenarios. First, if one format has a richer markup structure that the other, a certain kind on information on the other format may be represented in several ways, denoting a one-to-many relation. Cases may also exist where many-to-many relations are implied. The two other possible sources of ambiguity are: the freedom of order allowed for some structures, where any sequence is allowed; and when information representable in one format does not exist in the other and something must be filled in those fields.
Triple Graph Grammars Yet another approach to model transformation are triple graph grammars [KS06, GW06], where three subgraphs are created, one for the source model, other for the target model, and the third to establish the correspondence between the previous two graphs. Each graph needs a corresponding graph schema. Recapitulating, models are represented by graphs and meta-models are graph schemas.

The basic idea is that the source and target graphs evolve simultaneously by applying graph grammar transformation rules, and the third graph tracks the correspondences between them. However, in practice we want non-simultaneous updates and an incremental model synchronization approach requires bijective bidirectional model transformations. For that case, one of the two models (source or target) can be transformed independently, whereas the updates are propagated to the other through the correspondence graph.

Model synchronization In [XLH+07a], full semantics are given to model transformations that consider updates both to the source and the target models, in the context of a model synchronization framework inspired on the concepts from view-update lenses. For instance, in the simplified diagram for lenses (we are omitting create)

\[ S \xrightarrow{\text{put}} T \times S \xrightarrow{\pi_1} T \]

the “job” of \( \pi_1 \) is to say that the source type \( S \) in \( T \times S \) is stored in some sort of state and shall not be modified. Modifications to \( S \) only occur through propagation of updates on \( T \).

Allowing updates to the source model implies a redesign of the diagram:

\[ (S \times T) \xrightarrow{\text{sync}} T \]

\[ \xrightarrow{\pi_1} \]

\[ \xrightarrow{\pi_1 \circ \pi_1} \]

\[ \pi_1 \circ \pi_1 \]

\[ S \xrightarrow{\text{sync}} T \]

\[ (S \times T) \times S \xrightarrow{\text{sync}} T \]

The greatest change is that it no longer suffices to propagate the updates to the source model in order to achieve a synchronized state. Since both the source and target models can be modified, synchronization implies changes to both and a synchronized pair is returned. The new signature for the backward transformation \( \text{sync}_{f} :: (S \times S) \times S \rightarrow S \times T \) reflects this behavior. Note that however, \( \text{sync}_{f} \) is only stateful on the original source model (such as lenses are), since the original target model can be obtained by applying \( f \) to the state \( S \).

In the diagram, the forward transformation is also defined as comprising a state target model \( T \), although \( f :: S \rightarrow T \) is a normal stateless function, comparable to \( \text{get} \). This is because \( \text{sync}_{f} \) produces a pair of models, which we shall be able to modify. The fortunate aspect is that the state \( T \) does not need to be used in the forward direction: we can always compute a modified target model by applying \( f \) to a modified source model.

More formally, the authors specify the semantics of model synchronization according to four important properties: stability, preservation, propagation and composability.

Stability, similarly to (1.5) for lenses, expresses that if neither \( S \) nor \( T \) are modified, then shall neither the synchronization process modify them:

\[ \forall s \in S. \text{sync}_{f} ((s,f(s)), s) = (s,f(s)) \quad (5.1) \]

The preservation property states that synchronization shall keep the modifications to the source and target models after synchronization:

\[ \forall s \in S, s' \in S, \psi_t \in (T \rightarrow T). \text{sync}_{f} ((s', \psi_t(f(s))), s) = (s', \psi_t(f(s))) \quad (5.2) \]
This property is inspired in (1.4) for lenses, with the difference that it considers modifications to both the source and the target models instead only on the source model.

The propagation property says that the synchronization after transforming a modified source model shall preserve the original modifications:

\[ \forall s \in S, s' \in S, \psi_t \in (T \to T). \, \text{sync}_f ((s', \psi_t(f(s))), s) = (s'', \psi_t(f(s''))) \] (5.3)

This property is also inspired in (1.4) for lenses, but it concerns a two-way propagation of modifications.

Finally, the composability property, equivalent to (1.6) for very-well behaved lenses, states that synchronizing twice shall result in the same modifications as synchronizing once:

\[ \forall s \in S, s' \in S, t' \in T. \, \text{sync}_f (\text{sync}_f ((s', t'), s), s) = \text{sync}_f ((s', t'), s) \] (5.4)

The authors exemplify their synchronization framework with a traditional transformation from class models to relational database models. Assumed that the above properties hold, and if the source and target meta-models are available, the consistency of the models can be automatically maintained during transformation and synchronization processes. Additionally, the system can track inappropriate modifications, based on the synchronization properties, and report them to the users.
Bibliography


