Universidade do Minho
Escola de Engenharia

Hugo José Pereira Pacheco

Bidirectional Data Transformation by Calculation

July 2012

Hugo José Pereira Pacheco Bidirectional Data Transformation by Calculation

UMinho | 2012

Universidade do Minho
Escola de Engenharia

Hugo José Pereira Pacheco

Bidirectional Data Transformation by Calculation

MAPi Doctoral Programme in Computer Science

Supervisor:
Professor Doutor Manuel Alcino Pereira da Cunha
Co-supervisor:
Professor Doutor José Nuno Fonseca de Oliveira

July 2012

Universidade do Minho, ___/___/______

Assinatura: ____

Acknowledgments

Some PhD dissertations are the awaited culmination of a long and steep path since starting as an undergraduate until acquiring a doctorate degree. In my case, I have hardly ever considered enrolling in a PhD until I found myself doing one and, whenever asked about why I chose to follow this path, I believe to have (almost) always replied that it was simply to prove to myself that I was capable to see it to the end: “Congratulations, Hugo. You’ve made it.”

Still, it is true that I have enjoyed this research and life experience immensely. For convincing me and giving me the opportunity, I must thank my supervisor Alcino Cunha. He is one of the responsible individuals for my initiation into the research world, back to when I was an undergraduate, and has always kept the door open for me. During these years, he has always had a critical opinion about work and has taught me greatly how to sharpen my research capabilities. As his first PhD student, I am especially grateful for the dedication to my project and for all the invaluable time invested in my supervision. I believe that we have both learned a lot from this experience.

I must also thank José Nuno Oliveira for accepting being my co-supervisor. Since I first had him as lecturer, I have always admired his seamlessly passionate classes, and his earnest enthusiasm about research is inspiring for anyone around him. Although less present, he has always been eager to discuss new ideas and existing problems, and is an advocate of much of the theory that made the work defended in this thesis possible.

I am also extremely grateful to Zhenjiang Hu for inviting me for a short visit to Tokyo in the Autumn of 2010. Besides being very enjoyable, the few months spent in Tokyo have helped me greatly in improving the impact and quality of this work and have expanded my horizons for the future beyond return.

Not less important, I would like to thank my family, girlfriend and close friends for the unceasing support through all the periods of my life. And to all the others that have helped me through this particularly long journey.

Bidirectional Data Transformation by Calculation

The advent of *bidirectional programming*, in recent years, has led to the development of a vast number of approaches from various computer science disciplines. These are often based on domain-specific languages in which a program can be read both as a forward and a backward transformation that satisfy some desirable consistency properties.

Despite the high demand and recognized potential of intrinsically bidirectional languages, they have still not matured to the point of mainstream adoption. This dissertation contemplates some usually disregarded features of bidirectional transformation languages that are vital for deployment at a larger scale. The first concerns *efficiency*. Most of these languages provide a rich set of primitive combinators that can be composed to build more sophisticated transformations. Although convenient, such compositional languages are plagued by inefficiency and their optimization is mandatory for a serious application. The second relates to *configurability*. As update translation is inherently ambiguous, users shall be allowed to control the choice of a suitable strategy. The third regards *genericity*. Writing a bidirectional transformation typically implies describing the concrete steps that convert values in a source schema to values a target schema, making it impractical to express very complex transformations, and practical tools shall support concise and generic coding patterns.

We first define a point-free language of bidirectional transformations (called lenses), characterized by a powerful set of algebraic laws. Then, we tailor it to consider additional parameters that describe updates, and use them to refine the behavior of intricate lenses between arbitrary data structures. On top, we propose the *Multifocal* framework for the evolution of XML schemas. A *Multifocal* program describes a generic schema-level transformation, and has a value-level semantics defined using the point-free lens language. Its optimization employs the novel algebraic lens calculus.

Transformação Bidirecional de Dados por Cálculo

O advento da *programação bidirecional*, nos últimos anos, fez surgir inúmeras abordagens em diversas disciplinas de ciências da computação, geralmente baseadas em linguagens de domínio específico em que um programa representa uma transformação para a frente ou para trás, satisfazendo certas propriedades de consistência desejáveis.

Apesar do elevado potencial de linguagens intrinsicamente bidirecionais, estas ainda não amadureceram o suficiente para serem correntemente utilizadas. Esta dissertação contempla algumas características de linguagens bidirecionais usualmente negligenciadas, mas vitais para um desenvolvimento em mais larga escala. A primeira refere-se à *eficiência*. A maioria destas linguagens fornece um conjunto rico de combinadores primitivos que podem ser utilizados para construir transformações mais sofisticadas que, embora convenientes, são cronicamente ineficientes, exigindo ser otimizadas para uma aplicação séria. A segunda diz respeito à *configurabilidade*. Sendo a tradução de modificações inerentemente ambígua, os utilizadores devem poder controlar a escolha de uma estratégia adequada. A terceira prende-se com a *genericidade*. Escrever uma transformação bidirecional implica tipicamente descrever os passos que convertem um modelo noutro diferente, enquanto que ferramentas práticas devem suportar padrões concisos e genéricos de forma a poderem expressar transformações muito complexas.

Primeiro, definimos uma linguagem de transformações bidirecionais (intituladas de lentes), livre de variáveis, caracterizada por um poderoso conjunto de leis algébricas. De seguida, adaptamo-la para receber parâmetros que descrevem modificações, e usamo-los para refinar lentes intrincadas entre estruturas de dados arbitrárias. Por cima, propomos a plataforma *Multifocal* para a evolução de modelos XML. Um programa *Multifocal* descreve uma transformação genérica de modelos, cuja semântica ao nível dos valores e consequente otimização é definida em função da linguagem de lentes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the ever growing number of programming languages and software development frameworks, data transformations have become a ubiquitous tool to “bridge the gap” between technology layers and facilitate the sharing of information among software applications. These transformations are often specified in a typeful manner, in the sense that the input and output artifacts are required to be valid against some schema, meta-model or data format. Focusing on XML, an XSLT program implements a transformation between XML documents pertaining to different XML Schemas. In many scenarios, transformations occur at the level of types, this is, they convert schemas possibly conforming to higher-level schemas. Since XML schemas can themselves be represented as XML documents, XSLT can uniformly be used to specify schema transformations.

Real-world examples of type-level transformations include *format evolution* scenarios (Hainaut et al., 1994; Lämmel and Lohmann, 2001), such as small format enrichment, removal or refactoring steps as applied during system maintenance, or grammar modifications imposed by the natural evolution of programming languages. Similarly typical examples encompass less conservative format changes, involving cross-paradigm *data mapping* scenarios (Lämmel and Meijer, 2006; Melnik et al., 2007; Berdaguer et al., 2007), that promote the interoperability between UML models and Java source code implementations or the persistence of XML documents or object models to storage efficient relational databases.

What these transformation scenarios have in common is that, in order to maintain consistency, the transformation of a type calls for the coupled transformation of the data instances conforming to that type. When a schema is adapted in the context of system

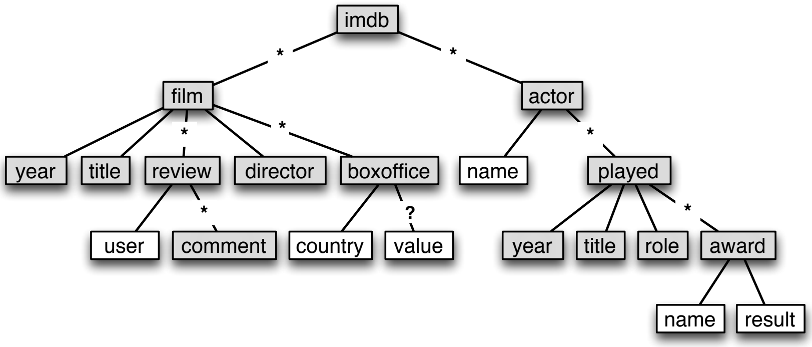


Figure 1.1: Representation of a movie database schema inspired by IMDb (http://www.imdb.com).

maintenance, the existing instances must be adapted to conform to the new schema. This problem has been studied for the cases of relational database schemas (Hainaut et al., 1994) and XML schemas (Lämmel and Lohmann, 2001). When the grammar of a programming language is modified, existing libraries and applications must be upgraded to the new language version (Lämmel, 2001; Vermolen and Visser, 2008). When an XML schema or object model specifying the logic of an application is migrated to a persistent SQL database schema, the required data mapping involves also the transformation of the XML documents or programs into SQL databases (Berdaguer et al., 2007; Melnik et al., 2007).

Consider an XML Schema to model a movie database adapted from (Berdaguer et al., 2007), represented graphically in Figure 1.1. In the picture, grey boxes denote XML Schema elements and white ones model XML Schema attributes. As a very simple example of a two-level transformation, if we choose to rename the `film` tag to the `movie` tag, by modifying the corresponding XML Schema to

```
<xs:element name="imdb"/>
 <xs:complexType>
  <xs:sequence>
   <xs:element name="movie" type="Movie"
    minOccurs="0" maxOccurs="unbounded"/>
   <xs:element name="actor" type="Actor"
    minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
...
```

the naturally induced migration on an underlying XML document

```
<imdb>
 <film>
  <year>2005</year>
  <title>Kill Bill: Vol. 1</title>
  <director>Quentin Tarantino</director>
  <boxOffice country="USA" value="22089322"/>
  <boxOffice country="Japan" value="3521628"/>
 </film>
 <actor>...</actor>
</imdb>
```

is to rename all film elements to movie elements:

```
<imdb>
 <movie>
  <year>2005</year>
  <title>Kill Bill: Vol. 1</title>
  <director>Quentin Tarantino</director>
  <boxOffice country="USA" value="22089322"/>
  <boxOffice country="Japan" value="3521628"/>
 </movie>
 <actor>...</actor>
</imdb>
```

Most existing transformation languages do not allow this. For example, we can use XSLT to separately specify a transformation between schemas and between documents, but the second is not a byproduct of the first. In this case, consistency between both transformations must be verified manually. As any *ad hoc* approach, this solution does not scale well as the complexity of the transformations increases, and makes the software development process much more expensive and error-prone.

By contrast with the standard *one-level transformations*, *two-level transformations* (Lämmel and Lohmann, 2001) denote type-level transformations of data formats coupled with value-level transformations of data instances corresponding to those formats, such that they are consistent by construction. Unlike before, in this setup the value-level transformations are not typed *a priori* for fixed formats, but are truly dependent on the type-level stage since the target type is computed dynamically as we carry the type-level transformation. Lämmel and Lohmann (2001) study the properties of such transformations and identify categories of transformation steps (for XML schema evolution), but stop short of formalizing and implementing a general framework for two-level transformation. In fact, they identify this additional step as an important research "challenge":

> *We have examined typeful functional XML transformation languages, term rewriting systems, combinator libraries, and logic programming.*

> *However, the coupled treatment of DTD transformations and induced XML transformations in a typeful and generic manner, poses a challenge for formal reasoning, type systems, and language design.*

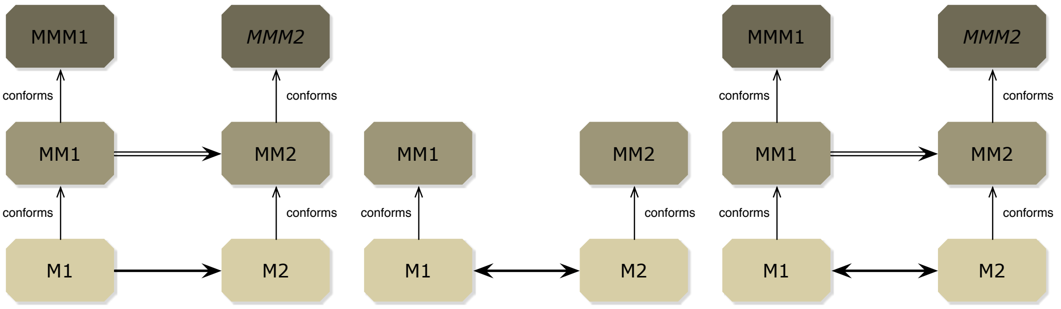
Coupling is not limited to schema and instance transformations, but a reoccurring phenomenon in computer science. Acknowledging this fact, Lämmel (2004a,b) identified a general class of *coupled transformations* comprising the transformation of software artifacts that induce the reconciliation of other related artifacts. Two-level data transformations arise then as a particular instance of such class, where the coupled artifacts are a data format on the one hand, and the data instances that conform to that format on the other hand.

Another prominent instance of coupling are *bidirectional transformations*, as a "mechanism for maintaining the consistency of two (or more) related sources of information" at the same level (Czarnecki et al., 2009). In a simplistic vision of software transformations, (two-level or not) *unidirectional transformations* would suffice. However, users generally expect transformations to be bidirectional, in the sense that, if after transformation both source and target instances co-exist[1] and sometimes evolve independently, changes made to one of the instances can be safely propagated to its connected pair in order to recover consistency.

For example, in our previous XML example, if we insert a new film in the source document, we should insert a corresponding movie in the target document, and vice-versa. This time, the coupling does not occur vertically, between type and value transformations, but horizontally, between forward and backward value transformations that propagate updates, as depicted in Figure 1.2.

Many of the above transformation scenarios are also explanatory of the need for bidirectionality. When format evolution serves only internal data storage, a forward conversion of old to new data may be sufficient, but when these formats concern data exported to other applications or different versions of the same application, both forward and backward data conversions may be needed on a repetitive or continuous basis to restore conformance. When the schema of an application is mapped onto a relational database schema, not only do we need to migrate the original application to a database, but also to transform further updates to the database back to a consistent application. Standard transformation languages are again unsatisfactory since they only consider

[1] We will denote the domains of bidirectional transformations as source and target, as read from left to right, and name the corresponding unidirectional transformations as forward and backward.



(a) Unidirectional 2LT (b) Bidirectional 1LT (c) Bidirectional 2LT

Figure 1.2: Different coupled transformation scenarios.

unidirectional transformations.

The naive way to write a bidirectional transformation is to engineer two unidirectional transformations together and manually prove that they are somehow consistent. This *ad hoc* solution is notoriously expensive and error-prone, because we have to write two transformations and not any two transformations are consistent for a specific purpose. It is also likely to cause a maintenance problem, since any change in a data format implies a redefinition of both transformations, and a new consistency proof.

Similarly to two-level transformations, a better approach is to design a domain-specific language in which every expression denotes both transformations, which are then guaranteed to be consistent by construction in the respective semantic space. Still, identifying the particular domains and language restrictions that best meet a specific application scenario, attaining expressiveness while holding the robustness imposed by the consistency constraints, is a difficult challenge as pointed out by Foster (2009)[2]:

> *The main challenge in the design of a bidirectional language lies in balancing the tradeoffs between syntax that is rich enough to express the queries and update policies demanded by applications, and yet simple enough that correctness can be verified using straightforward, compositional, and, ultimately, mechanizable checks.*

In response to this challenge, many intrinsically bidirectional transformation framework have been proposed in various computer science domains, including reversible computing (Mu et al., 2004; Yokoyama et al., 2008), data serialization (Kennedy, 2004;

[2] Debate on this topic has been initiated by Foster and coauthors in the seminal 2005 version of (Foster et al., 2007), published at the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

Atanassow and Jeuring, 2007), heterogeneous data synchronization (Kawanaka and Hosoya, 2006; Brabrand et al., 2008; Foster et al., 2007), string processing (Bohannon et al., 2008; Barbosa et al., 2010), software model transformation (Schürr, 1995; Stevens, 2007; Xiong et al., 2009; Diskin et al., 2011b), graph transformation (Hidaka et al., 2010), schema evolution (Berdaguer et al., 2007; Pacheco and Cunha, 2012), relational databases (Bancilhon and Spyratos, 1981; Dayal and Bernstein, 1982; Bohannon et al., 2006), functional programming (Liu et al., 2007; Matsuda et al., 2007; Voigtländer, 2009; Pacheco and Cunha, 2010) and user interfaces (Meertens, 1998; Hu et al., 2008). Most of these approaches are, as proposed by Foster (2009), based on *combinator-based* domain-specific languages, such that they provide a rich set of combinators that allow users to combine primitive transformations into sophisticated transformations in a compositional way.

Unfortunately, despite the convenience of compositional approaches, the resulting transformations can suffer from poor efficiency, due to the cluttering of intermediate data structures. Moreover, if manual design of bidirectional transformations is tedious and error-prone, manual optimization is a much more thankless (not to say impossible) task. In fact, the problem of optimization of bidirectional programs remains largely unaddressed by the community, although early recognized in the pioneering work on bidirectional *lenses* by Foster et al. (2007), and poses probably the greatest challenge:

> *Is there an algebraic theory of lens combinators that would underpin optimization of lens expressions in the same way that the relational algebra and its algebraic theory are used to optimize relational database queries? [...] This algebraic theory will play a crucial role in a more serious implementation effort.*

In the 2LT (*Two-Level Transformation*) project (Visser, 2008), we and others[3] have taken up these three challenges altogether: to develop a two-level bidirectional framework with the distinguished feature of being amenable to the optimization of transformations by calculation. The resulting *2LT* framework (Cunha et al., 2006a; Berdaguer et al., 2007) is implemented in *Haskell*, and proposes a combinator-based domain-specific language in which each combinator denotes a type-level transformation and two value-level transformations of conforming instances.

[3] The 2LT project is the result of an extensive line of work developed in the last 5 years at the University of Minho, and is accessible at http://2lt.googlecode.com.

The two-level component of the framework is defined by instantiating a well-known suite of combinators that have been previously defined for *strategic rewriting* (Visser, 2005; Lämmel, 2003), and popularized by systems such as *Stratego* (Visser, 2001) or *Strafunski* (Lämmel and Visser, 2003). Like most existing XML transformation and querying languages such as XSLT, XQuery or XPath, strategic programs enjoy a *structure-shy* flavor, as termed by Lieberherr (1995), that allows to specify transformations that modify only specific interesting bits of a structure in a concise and generic way without having to describe how to traverse the remaining structure.

The framework is also designed with calculation in mind. For this purpose, value-level transformations are also defined using a combinatorial approach, but with a careful choice of combinators with powerful algebraic laws that mitigate the optimization problem. In particular, we use the so-called *point-free* style (using a variable-free, combinator-based programming language) popularized by the *algebra of programming* community (Backus, 1978; Bird and de Moor, 1997). This allows us to not only derive efficient value-level transformations by calculation, but also to perform consistency proofs in the purely equational point-free calculus. Furthermore, the usage of structure-shy strategies for two-level transformations makes optimization a mandatory feature. Though the strategies can be applied to specific types to obtain corresponding bidirectional transformations, these will likely include many redundant intermediate steps and traverse whole input structures, in the image of the original strategies. An additional point-free transformation step is able to identify and fuse these redundancies to derive more optimal value-level transformations.

The bidirectional component of the framework is built upon the *data refinement* theory from (Morgan and Gardiner, 1990; Oliveira, 2008), used to automatically derive concrete implementations from abstract specifications by calculation, for example in the context of relational database design (Oliveira, 2004)[4].

For example, using the language from (Berdaguer et al., 2007) we can write a two-level strategy that refines the schema from Figure 1.1 into the schema from Figure 1.3 by adding a new alternative for TV series after renaming films to movies. Then, if we apply the forward transformation to compute the same target as before (assuming we choose not to insert default series in the resulting XML document), and edit the target by correcting the year of the first "Kill Bill" movie and inserting a new review

```
<imdb>
```

[4] Throughout this thesis, we will often refer to concrete and abstract models, under the notion that concrete ones contain more information than abstract ones.

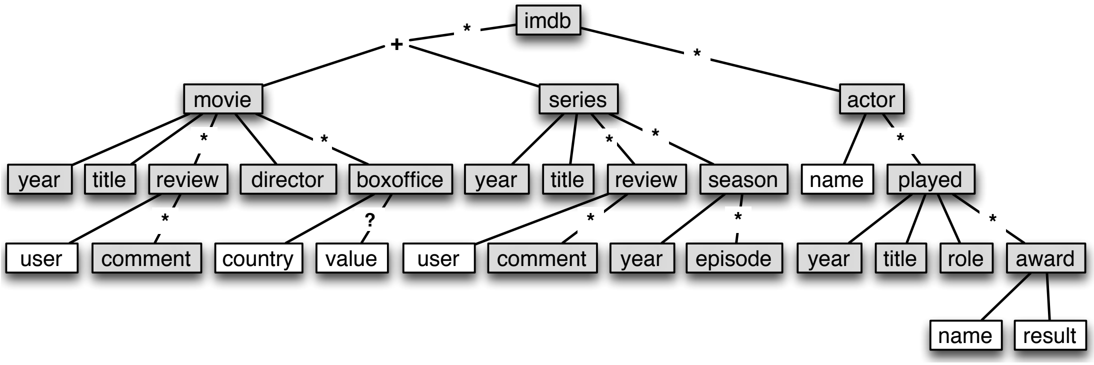


Figure 1.3: Evolution by refinement of the movie database schema from Figure 1.1.

```
 <movie>
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  <review user="emma" comment="Gorgeous!"/>
  <director>Quentin Tarantino</director>
  <boxOffice country="USA" value="22089322"/>
  <boxOffice country="Japan" value="3521628"/>
 </movie>
 <actor>...</actor>
</imdb>
```

backward execution is able to bring those changes back into a new source document by erasing irrelevant information and renaming movies to films:

```
<imdb>
 <film>
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  <review user="emma" comment="Gorgeous!"/>
  <director>Quentin Tarantino</director>
  <boxOffice country="USA" value="22089322"/>
  <boxOffice country="Japan" value="3521628"/>
 </film>
 <actor>...</actor>
</imdb>
```

Nevertheless, the *no information loss principle* of refinements (detailed in Chapter 3) permits a simple bidirectionalization, but at the cost of a rather limited updatability. Since the emphasis is placed on the forward transformation, if a user updates the target to values outside the range of the forward transformation, a corresponding backward transformation may reasonably fail, convert each series in the target into a movie in the source, or simply discard series elements, as they are not representable on the abstract side. However, a severe problem arises when editing the abstract, source side of the transformation. For instance, if we insert a new film in the source

```
<imdb>
 <film>
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  ...
 </film>
 <film>
  <year>1994</year>
  <title>Pulp Fiction>
  ...
 </film>
 <actor>...</actor>
</imdb>
```

while already having a target instance containing TV series

```
<imdb>
 <movie>
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  ...
 </movie>
 <series>
  <year>1999</year>
  <title>The Sopranos</title>
 </series>
 <actor>...</actor>
</imdb>
```

and convert the modified source into a new target

```
<imdb>
 <movie>
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  ...
 </movie>
 <movie>
  <year>1994</year>
  <title>Pulp Fiction>
  ...
 </movie>
 <actor>...</actor>
</imdb>
```

we get a mangled target instance that does not contain the original TV series. In truth, the forward transformation does not try to synchronize the source update with the existing target value, but rather creates a new concrete value afresh solely from the abstract information. Although this scheme is reasonable for the concrete-to-abstract direction, in which the concrete instance contains all the necessary information to create a new abstract instance, it is not satisfactory for the abstract-to-concrete transformation, that needs additional guidance on how to generate the pieces of concrete information not present in the abstract side.

This synchronization pattern has been extensively studied in the database community under the *view-update problem* (Bancilhon and Spyratos, 1981; Dayal and Bernstein, 1982). A database view is a virtual "window" created by issuing a query over the original database, that must be kept synchronized, such that data changes in the database shall alter the data shown in the view. If users are allowed to interact with views as with regular database tables, updates on views must likewise be translated into requests on the respective 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 (since abstracting is not injective, i.e., different tables can be abstracted to the same view), and the difficulty of choosing the update with the minimal "side-effects" on the source in accordance to sufficiently complex constraints is a problem that has been shown to be intractable (Buneman et al., 2002).

One of the most successful approaches to bidirectional transformations is the *Focal* language proposed by Foster et al. (2007), to tackle precisely the view-update problem for the domain-space of trees, whose building blocks are the so-called *lenses*. Lenses are in a sense duals of refinements. The forward transformation of a lens computes a view from a concrete data model, while its backward transformation takes both an updated view and the original model to return an updated concrete model.

Still, bidirectional languages like *Focal* are at best typed but not two-level. On top of that, the programming style that grants them bidirectionality is usually more biased towards structure-sensitive constructs, to be able to identify precisely the concrete steps required to translate between source and view documents.

Lenses, as a framework for data abstraction, can be of great value in scything through the complexity of large software systems. However, in order to express transformations over these systems, two-level and structure-shy specifications are imperative to reduce the specification cost and foster reusability. Possible application scenarios for structure-shy two-level lens programs may include the automated abstraction of implementation details at particular positions of a specification, or the creation of views for independent understanding of the components of a software system.

1.1 Goals and Contributions

The goal of this dissertation is to show that lenses, though pragmatically different, generalize refinements and can serve as an underlying theory for two-level transformations. While refinement scenarios are inherent to strategic rewriting techniques, like the automatic mapping of abstract schemas to more concrete ones (Berdaguer et al., 2007), abstraction scenarios typically involve more surgical steps that factor out or preserve specific pieces of information and motivate a transformation language with different features. On that account, we propose the *Multifocal* framework for the specification, optimization and execution of strategic two-level views over XML Schemas and algebraic data types, whose corresponding data migration transformations respect a strong bidirectional semantics.

Using *Multifocal*, we can write our running transformation (in the reverse direction) as the following strategic lens:

```
everywhere (try (at "film" (rename "movie")))
>> everywhere (try (at "series" erase))
```

This lens transformation is not only specified in a concise way close to its informal definition, but can also overcome the limitations of two-level refinements and correctly propagate modifications to source or view models. In particular, the forward transformation of this lens discards TV series, and its backward transformation recovers series in the source model, together with updating the remaining information about movies:

```
<imdb>
 <movie>
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  ...
 </movie>
 <series>
  <year>1999</year>
  <title>The Sopranos</title>
 </series>
 <movie>
  <year>1994</year>
  <title>Pulp Fiction>
  ...
 </movie>
 <actor>...</actor>
</imdb>
```

The development of *Multifocal* has motivated the following broad contributions:

Language & Calculus Two of the central aspects of the design of a bidirectional language are the semantic space over which it is defined – typically bound to a particular application domain – and how is correctness guaranteed for programs within such space. The first contribution of this thesis is the design of a general-purpose bidirectional lens language over arbitrary inductive data types, used in modern functional programming languages like *Haskell*, *ML* or *F#*. Following a natural embedding of our lens language in a functional language, the type system can then reason statically if a given lens program is type correct. Parallel to type correctness is the semantic correctness of lens programs w.r.t the bidirectional properties that they must satisfy. For our lens combinators, we write each value-level transformation in a point-free style that unveils a powerful point-free calculus and allows proving correctness properties in a purely equational and often mechanizable way. In the *2LT* framework, the optimization of bidirectional programs is achieved by independent optimization of the value-level transformations, and is thus not truly based on an algebraic bidirectional theory. This causes a severe overhead in the optimization process, since we have to perform two (or more) optimizations instead of a single one. Moreover, for the case of lenses, the complexity of the backward transformation – that must consider both the modified target and the original source, a problem that didn't exist for refinements – makes it also harder for an automatic procedure to spot many optimization opportunities. Another contribution of this thesis is a definite answer to the second challenge identified in the introduction: an algebraic calculus for lenses that enables equational reasoning on lenses and allows to perform proofs about bidirectional programs as if regular functional programs. In particular, we establish that most of the standard point-free combinators can be lifted to lens combinators in our language and preserve their characterizing set of algebraic laws. As a result, we can show that optimization at the lens level subsumes the independent optimization of all the value-level transformations, opening interesting perspectives towards agile automatic lens optimization tools.

Alignment Despite the wide span of bidirectional programming languages, an orthogonal issue is that of *alignment*. In lenses, the forward transformation might discard source information, while the backward transformation shall recombine parts of the updated target with parts of the original source to produce an updated source. When the source or target schemas are recursive, ordered structures, to behave as desired lenses must be able to identify the modifications and establish correspondences between source and target models. For example, if we write a transformation that keeps only movie titles and later reorganize the movies in the target sequence, we want the abstracted information (year, director, etc) to be restored to the same movies in the source sequence, instead of being matched positionally, as our initial lens language would do. Inspired by Barbosa et al. (2010), our next contribution is to recast our point-free lens language as a set of alignment-aware combinators that compute, propagate and use these correspondences alongside value-level transformations. This interaction and the additional laws that guarantee that correspondences are propagated correctly are formalized using dependent type theory. Our extension is not only able to solve data alignment for typical mapping scenarios in the spirit of (Barbosa et al., 2010), but also to perform shape alignment for more intricate reshaping transformations between inductive data types.

Implementation The last contribution of this thesis is the implementation of the *Multifocal* framework providing strong types and general strategies for the design of two-level bidirectional lens transformations. On top of this framework, we develop the *Multifocal* language for structure-shy two-level evolution of XML Schemas, providing conforming bidirectional lenses to translate source and target XML documents for free. In comparison to a *Focal* lens program, that describes a bidirectional view between two particular tree structures, a *Multifocal* program describes a general type-level transformation that provides multiple focus points, in the sense that it produces a different view schema and a corresponding bidirectional lens for each XML Schema to which it is successfully applied. At the core of the framework, we provide a combinator library for the two-level evolution of arbitrary inductive data type representations, to which *Multifocal* programs can be translated. The value-level semantics of these combinators is defined according to our point-free lens language. Moreover, we instantiate a similar strategic rewrite system for the transformation of point-free programs with our algebraic lens laws, to enable the automatic optimization of lens transformations resulting from the evaluation of the two-level stage.

1.2 Overview of the Thesis

Chapter 2 introduces the point-free programming style and other formal notations used for the developments in the succeeding chapters.

Chapter 3 presents a detailed panorama of the state of the art on bidirectional transformations, with particular emphasis on lenses, and attempts to deliver a multi-perspective taxonomy for the classification of existing systems. It also situates our approach inside this design space. Despite the multitude and diversity of existing solutions, many of the fundamental problems of the area are not already well established, mostly due to the non-existence of a universal classifying system and to the lack of common theoretical grounds between many of the approaches. For these reasons, this chapter is *per se* a relevant contribution. Part of this material is discussed in a joint article currently under submission (Pacheco et al., 2012b).

Chapter 4 introduces our point-free lens language over arbitrary inductive data types and unveils the rich set of algebraic laws that rule the inherent lens calculus. The language and calculus presented in this chapter were published in two papers (Pacheco and Cunha, 2010, 2011).

Chapter 5 discusses our vision of alignment and proposes a formal solution tailored to our point-free lens language. This chapter is an extended version of a recent paper (Pacheco et al., 2012a).

Chapter 6 sheds some light into the technical details employed in the implementation of the *Multifocal* framework and provides a more comprehensive set of examples that demonstrate our approach. Some of the components of the rewrite systems for the transformation of inductive types and the optimization of point-free lenses were published in two papers (Cunha and Pacheco, 2011; Pacheco and Cunha, 2011). The *Multifocal* language for XML Schema evolution was detailed in a separate publication (Pacheco and Cunha, 2012).

Chapter 7 reviews the main contributions of the thesis, assesses the impact of the developed work in the bidirectional transformation community, and ends by identifying possible directions for future work.

Chapter 2

Point-free Programming

This chapter introduces the point-free programming style of and respective notation used throughout this thesis. It can be skipped on a first reading and consulted as a reference by readers familiar with algebraic programming and the point-free style.

2.1 Point-free Functional Calculus

The so-called *point-free* style of programming, popularized by John Backus in his 1977 Turing award lecture (Backus, 1978), is a variable-free style where functions are defined by composition of higher-order combinators that are characterized by a rich set of algebraic laws, making it particularly amenable to program calculation.

In this thesis, we will use a standard set of point-free combinators that can be found in related literature (Bird and de Moor, 1997; Gibbons, 2002; Meijer et al., 1991; Cunha et al., 2006b). The algebraic programming approach described in such literature lays its foundations on category theory in general (Mac Lane, 1998; Pierce, 1991), that studies abstract mathematical structures and the relationships between them. In our presentation, we will only use basic categorical concepts that are necessary to introduce our point-free language, instantiated for the concrete category of types and functions that realizes the algebraic approach to functional programming.

2.1.1 Basic Combinators

A category is a mathematical construction consisting of a collection of objects and a collection of arrows (or morphisms), satisfying the following conditions:

- Each arrow f has a domain object A and codomain object B and is denoted by $f : A \to B$;

- Two arrows $f : B \to C$ and $g : A \to B$ can be composed into a single arrow $f \circ g : A \to C$, such that the composition operation $\circ$ is associative:

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \circ\text{-Assoc}$$

- Every object A is connected to itself through an identity arrow $id_A : A \to A$, that is the unit of composition:

$$id \circ f = f = f \circ id \qquad id\text{-Nat}$$

Note that in the id-NAT law we have omitted the object subscripts for id. To simplify the notation, we will often drop such subscripts whenever they are irrelevant or can be inferred from the context, as is the case of "polymorphic" morphisms like id. The complete definition of id-NAT with subscripts would be $id_B \circ f = f = f \circ id_A$.

An example of a concrete category is SET, the category of sets and total functions, where the objects are sets (types) and arrows are typed total functions. For each type A, there is an identity function $id_A : A \to A$. Composition is the standard set-theoretical composition of functions with coinciding intermediate types.

Although we will often motivate the theory with examples in Haskell, a lazy functional programming language that supports partial function definitions, our semantic domain of choice will be the SET category. Most of the past research on algebraic programming was developed in this same SET category. The study of algebraic programming in the alternative CPO category, more adequate to model the partiality of functions and values typical of lazy functional languages, can be found in (Meijer et al., 1991; Cunha, 2005).

A morphism $f : A \to A$ between the same object A is called an *endomorphism*, or in SET, an *endofunction*. An *isomorphism* $f : A \to B$ is a special morphism, for which there exists a unique *inverse* morphism $f^{-1} : B \to A$ such that:

$$f^{-1} \circ f = id_A \quad \wedge \quad f \circ f^{-1} = id_B \qquad \text{Iso}$$

In this case, we say that the objects A and B are *isomorphic* and write $A \cong B$.

Functors A functor $\mathsf{F} : \mathbb{C} \to \mathbb{D}$ is a mapping between two categories $\mathbb{C}$ and $\mathbb{D}$, and is defined according to two mappings: one that associates to every object A of $\mathbb{C}$ a result object $\mathsf{F}\ A$ of $\mathbb{D}$; and another that, for every arrow $f : A \to B$ of $\mathbb{C}$, constructs an arrow $\mathsf{F}\ f : \mathsf{F}\ A \to \mathsf{F}\ B$ of $\mathbb{D}$ satisfying the following equations:

$$\mathsf{F}\ id_A = id_{\mathsf{F}\ A} \qquad \text{FUNCTOR-ID}$$

$$\mathsf{F}\ (f \circ g) = \mathsf{F}\ f \circ \mathsf{F}\ g \qquad \text{FUNCTOR-COMP}$$

Whenever the domain and codomain categories coincide, the functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ mapping a category $\mathbb{C}$ to itself is called an *endofunctor*.

A binary functor (or bifunctor) B is a mapping from a pair of categories to a category, such that if $f : A \to C$ and $g : B \to D$ then $\mathsf{B}\ f\ g : \mathsf{B}\ A\ B \to \mathsf{B}\ C\ D$. In SET, a bifunctor takes pairs of types into types and pairs of functions into functions, obeying the following conditions:

$$\mathsf{B}\ id_A\ id_B = id_{\mathsf{B}\ A\ B} \qquad \text{BIFUNCTOR-ID}$$

$$\mathsf{B}\ (f \circ g)\ (h \circ i) = \mathsf{B}\ f\ h \circ \mathsf{B}\ g\ i \qquad \text{BIFUNCTOR-COMP}$$

By applying a bifunctor $\mathsf{B} : \mathbb{C} \to \mathbb{D} \to \mathbb{E}$ to a type A of $\mathbb{C}$, we get the unary functor $\mathsf{B}\ A : \mathbb{D} \to \mathbb{E}$.

Natural Transformations A *natural transformation* η between functors $\mathsf{F} : \mathbb{C} \to \mathbb{D}$ and $\mathsf{G} : \mathbb{C} \to \mathbb{D}$, denoted by $\eta : \mathsf{F} \stackrel{\cdot}{\to} \mathsf{G}$, is an arrow that assigns to each object A in $\mathbb{C}$ an arrow $\eta_A : \mathsf{F}\ A \to \mathsf{G}\ A$ between objects of $\mathbb{D}$, such that, for any arrow $f : A \to B$, the following naturality condition holds:

$$\mathsf{G}\ f \circ \eta_A = \eta_B \circ \mathsf{F}\ f \qquad \eta\text{-NAT}$$

The notion of natural transformation can also be generalized to bifunctors. Whenever a natural transformation η is also an isomorphism, it is called a *natural isomorphism*.

Terminal and Initial objects A *terminal object* of a category $\mathbb{C}$ is an object T such that, for each object A of $\mathbb{C}$, there is exactly one arrow of type $A \to T$. All terminal objects are unique up to isomorphism, and are usually denoted by an object 1, where the unique arrow from A to 1 is popularly known as the *bang* morphism $!_A : A \to 1$.

In category theory, the uniqueness of the constructions is typically formulated as a *universal property*. The universal property for $!$ is expressed as follows:

$$f = !_A \ \Leftrightarrow\ f : A \rightarrow 1 \qquad \text{!-UNIQ}$$

Every terminal object of the SET category is isomorphic to the singleton set $1 = \{1\}$ that models a data type with a single value.

Other derived reflexivity and fusion laws can be easily proved from uniqueness:

$$!_1 = id_1 \qquad \text{!-REFLEX}$$
$$! \circ f = ! \qquad \text{!-FUSION}$$

An *initial object* of a category $\mathbb{C}$ is an object I such that, for each object A of $\mathbb{C}$, there is exactly one arrow of type $I \rightarrow A$. Likewise, all initial objects are unique up to isomorphism. The unique initial object is usually denoted by an object 0, and the unique arrow from 0 to A by an arrow $¡_A : 0 \rightarrow A$. In SET, every initial object is isomorphic to the empty type $0 = \{\,\}$, and the $¡_A$ arrow is the empty function. In our presentation, initial objects are only of theoretical significance, namely to represent data types with no values.

Points In categorical terms, an element of an object A can be represented by an arrow of type $1 \rightarrow A$, usually called a *point*. For an element a of type A, the *point* combinator $\underline{a} : 1 \rightarrow A$ denotes the corresponding point in the category. A morphism $c : B \rightarrow A$ is called *constant* if for any other two arrows $f, g : C \rightarrow B$ we have $c \circ f = c \circ g$. By composing points with !, we can define a constant morphism $\underline{a} \circ !_B : B \rightarrow A$ that ignores the argument element of B and always returns the fixed element a of A.

Products The product of two objects A and B is defined as an object $A \times B$ and two arrows $\pi_1 : A \times B \rightarrow A$ and $\pi_2 : A \times B \rightarrow B$. The universal property for products is expressed by stating that, for any two arrows $f : C \rightarrow A$ and $g : C \rightarrow B$, there exists a unique arrow $f \vartriangle g : C \rightarrow A \times B$ such that:

$$h = f \vartriangle g \ \Leftrightarrow\ \pi_1 \circ h = f \ \wedge\ \pi_2 \circ h = g \qquad \times\text{-UNIQ}$$

In SET, the product of two types A is isomorphic to the Cartesian product $A \times B = \{(x, y) \mid x \in A, y \in B\}$. The *projections* π_1 and π_2 project out the left and right components of a pair, respectively, and the *split* combinator $f \triangle g$ builds a pair by applying f and g to the same input value.

Other typical laws can be derived from the uniqueness of categorical products:

$$\pi_1 \triangle \pi_2 = id \qquad \times\text{-REFLEX}$$

$$\pi_1 \circ (f \triangle g) = f \quad \wedge \quad \pi_2 \circ (f \triangle g) = g \qquad \times\text{-CANCEL}$$

$$(f \triangle g) \circ h = f \circ h \triangle g \circ h \qquad \times\text{-FUSION}$$

Although less general, these laws are more amenable to calculation because they allow us to perform equational reasoning, i.e., to substitute (sub)terms in an expression by equivalent terms, according to the laws. Using uniqueness, on the other side, we must first prove the equivalences before substituting the terms, by processing the whole formula. For example, using ×-REFLEX we can directly replace the term $\pi_1 \triangle \pi_2$ with id, whereas using ×-UNIQ we have to first prove the right equation of the formula. The other downside of uniqueness (for calculation) is that even instantiating one side of the equation, there still remain uninstantiated variables in the other side, what requires the programmer to already have an idea of the proof instead of performing a simple substitution. For example, using ×-UNIQ we have to instantiate h to id, and solve the right equation for $f = \pi_1$ and $g = \pi_2$, to prove that it is equal to $\pi_1 \triangle \pi_2$.

Given two arrows $f : A \to C$ and $g : B \to D$, we can define a product arrow $f \times g : A \times B \to C \times D$ as follows:

$$f \times g = f \circ \pi_1 \triangle g \circ \pi_2 \qquad \times\text{-DEF}$$

In SET, this *product* combinator builds a new pair by applying two functions in parallel to the left and right elements of a pair, and can be seen as a bifunctor $\cdot \times \cdot : \text{SET} \to \text{SET} \to \text{SET}$:

$$id \times id = id \qquad \times\text{-FUNCTOR-ID}$$

$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-FUNCTOR-COMP}$$

Coproducts The coproduct of two objects A and B is defined as an object $A + B$ and two arrows $i_1 : A \to A + B$ and $i_2 : B \to A + B$. Dually to products, given two arrows $f : A \rightarrow C$ and $g : B \rightarrow C$, the uniqueness of an arrow $f \triangledown g : A + B \rightarrow C$ is guaranteed by the following universal property:

$$h = f \triangledown g \;\Leftrightarrow\; h \circ i_1 = f \;\wedge\; h \circ i_2 = g \qquad +\text{-UNIQ}$$

In SET, the coproduct of two types A and B is isomorphic to the disjoint sum $A + B = \{L\ x \mid x \in A\} \cup \{R\ y \mid y \in B\}$. The *injections* i_1 and i_2 build left and right alternatives (tagged with L and R, respectively) and the *either* (or *junction*) combinator $f \triangledown g$ applies a function f if the input is a left alternative or a function g otherwise.

We can also prove reflexivity, cancellation and fusion laws for coproducts:

$$i_1 \triangledown i_2 = id \qquad +\text{-REFLEX}$$

$$(f \triangledown g) \circ i_1 = f \quad \wedge \quad (f \triangledown g) \circ i_2 = g \qquad +\text{-CANCEL}$$

$$f \circ (g \triangledown h) = f \circ g \triangledown f \circ h \qquad +\text{-FUSION}$$

Given two arrows $f : A \rightarrow C$ and $g : B \rightarrow D$, we can build a coproduct arrow $f + g : A + B \rightarrow C + D$ as follows:

$$f + g = i_1 \circ f \triangledown i_2 \circ g \qquad +\text{-DEF}$$

The definition in SET of this derived *coproduct* combinator $f + g$ uses a function f to build a left alternative from a left alternative, or a function g otherwise to build a right alternative from a right alternative. As for products, it denotes a bifunctor $\cdot + \cdot : \text{SET} \rightarrow \text{SET} \rightarrow \text{SET}$ according to the following laws:

$$id + id = id \qquad +\text{-FUNCTOR-ID}$$

$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad +\text{-FUNCTOR-COMP}$$

Exponentials For a category $\mathbb{C}$ with terminal object and products, the exponential of two objects B and C is defined as an object C^B and a morphism $ap : C^B \times B \rightarrow C$ such that, for any morphism $f : A \times B \rightarrow C$, there exists a unique morphism $\overline{f} : A \rightarrow C^B$ such that the following universal property holds:

$$g = \overline{f} \;\Leftrightarrow\; f = ap \circ (g \times id) \qquad \text{EXP-UNIQ}$$

The uniqueness of exponential objects also entails the following derived laws:

$$\overline{ap} = id \qquad \textsc{Exp-Reflex}$$

$$ap \circ (\overline{f} \times id) = f \qquad \textsc{Exp-Cancel}$$

$$\overline{f} \circ g = \overline{f \circ (g \times id)} \qquad \textsc{Exp-Fusion}$$

In the point-free style, higher-order functions are definable through exponentiation. The exponential object C^B in Set denotes the set of all functions from B to C. The morphism ap is often called *apply* and applies a function to a value as follows:

$$\begin{aligned} &ap : B^A \times A \to B \\ &ap\ (f, x) = f\ x \end{aligned}$$

The morphism $\overline{\cdot}$ and its converse $\widehat{\cdot}$ are usually called *curry* and *uncurry*, respectively, and convert a function on pairs into a higher-order function, and vice-versa:

$$\begin{aligned} &\overline{\cdot} : (A \times B \to C) \to (A \to C^B) && \widehat{\cdot} : (A \to C^B) \to (A \times B \to C) \\ &\overline{f}\ a\ b = f\ (a, b) && \widehat{f}\ (a, b) = f\ a\ b \end{aligned}$$

The curry and uncurry combinators are each other's inverse, as stated below:

$$\overline{\widehat{f}} = f \quad \wedge \quad \widehat{\overline{f}} = f \qquad \textsc{Curry-Uncurry-Iso}$$

Given a type A of $\mathbb{C}$, we can turn the curry operation into an exponentiation functor $\cdot^A : \mathbb{C} \to \mathbb{C}$ that maps each type B into an exponential type B^A and provides an operation $f^A : B^A \to C^A$, for each function $f : B \to C$:

$$f^A = \overline{f \circ ap} \qquad \textsc{Exp-Def}$$

When the type superscript is not relevant, we replace it by the symbol $\bullet$. The following laws show that $\cdot^\bullet$ is truly a functor in Set:

$$id^\bullet = id \qquad \textsc{Exp-Functor-Id}$$

$$f^\bullet \circ g^\bullet = (f \circ g)^\bullet \qquad \textsc{Exp-Functor-Comp}$$

Natural Isomorphisms We can define other derived combinators that are useful "plumbing" devices to connect point-free expression with different intermediate but isomorphic types, much like wiring connectors in an electrical circuit. For example, the following combinators express the commutativity, associativity and distributivity of sums and products:

$$\begin{aligned}
&\mathit{swap} : A \times B \to B \times A\\
&\mathit{assocl} : A \times (B \times C) \to (A \times B) \times C\\
&\mathit{assocr} : (A \times B) \times C \to A \times (B \times C)\\
&\mathit{coswap} : A + B \to B + A\\
&\mathit{coassocl} : A + (B + C) \to (A + B) + C\\
&\mathit{coassocr} : (A + B) + C \to A + (B + C)\\
&\mathit{distl} : (A + B) \times C \to (A \times C) + (B \times C)\\
&\mathit{undistl} : (A \times C) + (B \times C) \to (A + B) \times C\\
&\mathit{distr} : A \times (B + C) \to (A \times B) + (A \times C)\\
&\mathit{undistr} : (A \times B) + (A \times C) \to A \times (B + C)
\end{aligned}$$

All these functions are natural isomorphisms in SET, as witnessed by the respective equational laws presented in Appendix A.

Conditionals In functional languages, predicates are specified as functions from some source type into a Boolean type. The type of Booleans can be modeled in SET as a set containing only two elements, denoted by $2 = \{\mathit{True}, \mathit{False}\}$. Also, there is an isomorphism $2 \cong 1 + 1$ allowing us to reuse the coproduct operations for Booleans. For a predicate $p : A \to 2$, we can define a guard combinator $p? : A \to A + A$ that applies the predicate while keeping a copy of the input as follows:

$$p? = (\pi_1 + \pi_1) \circ \mathit{distr} \circ (\mathit{id} \vartriangle p) \qquad \text{?-DEF}$$

For an input value a of A, $p?\ a$ tags the input as a left value $L\ a$ if it satisfies the predicate ($p\ a = \mathit{True}$) or as a right value otherwise ($p\ a = \mathit{False}$).

When combined with an either, it allows us to specify conditionals. Given a predicate $p : A \to 2$ and two functions $f : A \to B$,$g : A \to B$, the usual point-wise *conditional* expression (**if** $p\ a$ **then** $f\ a$ **else** $g\ a$) can be encoded as the point-free expression $(f \triangledown g) \circ p?$. We can also prove the following properties about conditionals:

$$(f \triangledown g) \circ p? \circ h = (f \circ h \triangledown g \circ h) \circ (p \circ h)? \qquad \text{?-FUSION}$$

$$(f \triangledown f) \circ p? = f \qquad \text{?-CANCEL}$$

2.1.2 Recursive Combinators

In functional languages, algebraic data types are inductively defined using a set of constructors, where each such constructor contains a set of values. For example, the types of naturals and lists can be encoded in Haskell as follows (using special syntax for lists):

$$\mathbf{data}\ Nat = Zero \mid Succ\ Nat$$
$$\mathbf{data}\ [a] = [\,] \mid a : [a]$$

Most of these recursive types commonly found in functional programming can be uniquely represented (up to isomorphism) as the least fixed point of a polynomial functor (Meijer et al., 1991). Given a *base functor* $\mathsf{F} : \textsc{Set} \to \textsc{Set}$, the inductive type generated by its least fixed point (to be introduced later) will be denoted by $\mu\mathsf{F}$.

A polynomial functor is a functor defined according to the following language:

$$\mathsf{F} = \underline{A} \mid \mathsf{Id} \mid \mathsf{F} \otimes \mathsf{F} \mid \mathsf{F} \oplus \mathsf{F} \mid \mathsf{F} \odot \mathsf{F}$$

The constant functor $\underline{A}$ encapsulates a data type A and the identity functor Id denotes recursive invocation. The remaining combinators model the product $\otimes$, coproduct $\oplus$ and composition $\odot$ of polynomial functors. For example, for naturals we have $Nat = \mu\mathsf{Nat}$, where $\mathsf{Nat} = \underline{1} \oplus \mathsf{Id}$, and for lists $[A] = \mu\mathsf{List}_A$, where $\mathsf{List}_A = \underline{1} \oplus \underline{A} \otimes \mathsf{Id}$[1]. The application of a polynomial functor F to a type A is isomorphic to a "flattened" sums-of-products type $\mathsf{F}\ A$, according to the following specialization equations:

$$\begin{aligned}
&\underline{C}\ A = C \\
&\mathsf{Id}\ A = A \\
&(\mathsf{F} \oplus \mathsf{G})\ A = \mathsf{F}\ A + \mathsf{G}\ A \\
&(\mathsf{F} \otimes \mathsf{G})\ A = \mathsf{F}\ A \times \mathsf{G}\ A \\
&(\mathsf{F} \odot \mathsf{G})\ A = \mathsf{F}\ (\mathsf{G}\ A)
\end{aligned}$$

[1] For a type A, we represent its base functor A by the same name but in the sans serif font of functors.

For the base functors of naturals and lists we have $\mathsf{Nat}\ A = 1 + A$ and $\mathsf{List}_X\ A = 1 + X \times A$.

Each data type $T = \mu\mathsf{F}$ comes equipped with two morphisms: $out_{\mathsf{F}} : \mu\mathsf{F} \to \mathsf{F}\ \mu\mathsf{F}$, that can be used to expose its top-level structure (in a sense, encoding pattern matching over that type), and $in_{\mathsf{F}} : \mathsf{F}\ \mu\mathsf{F} \to \mu\mathsf{F}$, that determines how values of that type can be constructed. They are each other's inverse as witnessed by the following law:

$$in_{\mathsf{F}} \circ out_{\mathsf{F}} = id \quad \wedge \quad out_{\mathsf{F}} \circ in_{\mathsf{F}} = id \qquad\qquad in\text{-}out\text{-}\textsc{Iso}$$

The typical list constructors can be defined as the point-free terms $nil = in_{\mathsf{List}_A} \circ i_1$ and $cons = in_{\mathsf{List}_A} \circ i_2$, such that $nil \triangledown cons = in_{\mathsf{List}_A}$, while the constructors for naturals are defined as $zero = in_{\mathsf{Nat}} \circ i_1$ and $succ = in_{\mathsf{Nat}} \circ i_2$, with $zero \triangledown succ = in_{\mathsf{Nat}}$.

In functional programming, inductive data types that are parameterized by type variables, as is the case of lists, are called polymorphic. The structure of a polymorphic type T with a type variable A can be represented as the fixed point $T\ A = \mu(\mathsf{B}\ A)$ of a partially applied polynomial bifunctor $\mathsf{B} : \textsc{Set} \to \textsc{Set} \to \textsc{Set}$ (Jansson and Jeuring, 1997), with $out_{\mathsf{B}\ A} : T\ A \to \mathsf{B}\ A\ (T\ A)$ and $in_{\mathsf{B}\ A} : \mathsf{B}\ A\ (T\ A) \to T\ A$. The language of polynomial functors can be generalized to a language of *polynomial bifunctors*:

$$\mathsf{B} = \underline{A} \mid \mathsf{Id} \mid \mathsf{Par} \mid \mathsf{B} \otimes \mathsf{B} \mid \mathsf{B} \oplus \mathsf{B} \mid \mathsf{F} \odot \mathsf{B}$$

This language includes the constant $\underline{A}$ that lifts a regular type A to a bifunctor, the product bifunctor $\otimes$, the coproduct bifunctor $\oplus$ and the composition bifunctor $\odot$ of a polynomial functor F after a bifunctor B. The combinators Par (that will be used to model type parameters) and Id (that, as before, is used to model recursive invocations) select the first and second type arguments of a bifunctor, respectively. For example, for lists we have $[A] = \mu((\underline{1} \oplus \mathsf{Par} \otimes \mathsf{Id})\ A)$.

Some algebraic data types are defined in terms of other user-defined data types, such as the type of n-ary leaf trees:

$$\mathbf{data}\ LTree\ a = Leaf\ a \mid Fork\ [LTree\ a]$$

In order to represent such types, the above grammar of polynomial functors is often extended to consider also compositions $\mathsf{F} \odot \mathsf{G}$ whose left side F can be not only a polynomial functor but also a type functor (the functor modeled by a polynomial type). The same can be said about binary functors, and bifunctor composition $\mathsf{F} \odot \mathsf{B}$ can be extended to allow F to be a type functor. The resulting grammar of functors (and corresponding family of fixed point types) is called *regular*. For n-ary leaf trees, we obtain $\mathit{LTree}\ A = \mu \mathsf{LTree}_A$, with the regular functor $\mathsf{LTree}_A = \underline{A} \oplus ([\,] \odot \mathsf{Id})$, or alternatively the bifunctor-based encoding $\mathit{LTree}\ A = \mu((\mathsf{Par} \oplus ([\,] \circledcirc \mathsf{Id}))\ A)$.

The functor mapping $\mathsf{F}\ f : \mathsf{F}\ A \to \mathsf{F}\ B$ is a function that preserves the functorial structure and modifies all the instances of the type argument A into instances of type B. It can be defined as a *polytypic* function that takes an arbitrary functor as a parameter and proceeds by induction on the structure of the argument functor. For polynomial functors, we define functor mapping as follows[2]:

$$\begin{array}{ll} \forall f : A \to B.\ \mathsf{F}\ f : \mathsf{F}\ A \to \mathsf{F}\ B & \\ \mathsf{Id}\ f & = f \\ \underline{C}\ f & = id \\ (\mathsf{F} \otimes \mathsf{G})\ f & = \mathsf{F}\ f \times \mathsf{G}\ f \\ (\mathsf{F} \oplus \mathsf{G})\ f & = \mathsf{F}\ f + \mathsf{G}\ f \\ (\mathsf{F} \odot \mathsf{G})\ f & = \mathsf{F}\ (\mathsf{G}\ f) \end{array} \qquad \textsc{Functor-Def}$$

In this definition, the argument function f is applied to the recursive occurrences of the functor, constants are left unchanged, and the remaining cases proceed by induction. The value-level mapping for type functors will be defined later in this section.

Catamorphisms A base functor also dictates a unique way of consuming and producing values of the corresponding data type, using well-known recursion patterns instead of defining functions by general recursion.

In categorical terms, given a functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$, an F-*algebra* is a morphism of type $\mathsf{F}\ A \to A$, where the object A is called the carrier of the algebra, and a F-*homomorphism* from an F-algebra $f : \mathsf{F}\ A \to A$ to an F-algebra $g : \mathsf{F}\ B \to B$ is a morphism $h : A \to B$ such that the property $h \circ f = g \circ \mathsf{F}\ h$ holds.

An *initial* F-*algebra* is an initial object in the category where algebras are objects and homomorphisms are arrows. Various data types (for finite data structure) used in functional programming can be represented by initial algebras of particular functors. For instance, for the polynomial functors of SET, such initial algebras exist and are unique up to isomorphism. We denote the unique initial algebra of a functor F by

[2] The bifunctor mapping $\forall f : A \to C, g : B \to D.\ \mathsf{B}\ f\ g : \mathsf{B}\ A\ B \to \mathsf{B}\ C\ D$ can be defined in a similar way for polynomial bifunctors.

$in_{\mathsf{F}} : \mathsf{F}\ \mu\mathsf{F} \rightarrow \mu\mathsf{F}$, where the type $\mu\mathsf{F}$ is the least fixed point of the functor.

The existence of an initial algebra (and an associated fixed point) means that, given any other algebra $g : \mathsf{F}\ A \rightarrow A$, there is a unique homomorphism $\llparenthesis g \rrparenthesis_{\mathsf{F}} : \mu\mathsf{F} \rightarrow A$ that makes the hereunder diagram commute:

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[d, "\llparenthesis g \rrparenthesis_{\mathsf{F}}"'] & \mathsf{F}\ \mu\mathsf{F} \arrow[l, "in_{\mathsf{F}}"'] \arrow[d, "\mathsf{F}\ \llparenthesis g \rrparenthesis_{\mathsf{F}}"] \\
A & \mathsf{F}\ A \arrow[l, "g"]
\end{tikzcd}$$

The resulting morphism is called a fold or a *catamorphism* and can express all functions defined by iteration. A catamorphism recursively consumes values of an inductive data type $\mu\mathsf{F}$ by replacing their constructors by those of the given algebra g. The above diagram can be rephrased into the following uniqueness law:

$$f = \llparenthesis g \rrparenthesis_{\mathsf{F}} \;\Leftrightarrow\; f \circ in_{\mathsf{F}} = g \circ \mathsf{F}\ f \qquad \llparenthesis\cdot\rrparenthesis\text{-}\textsc{Uniq}$$

From fold uniqueness, it is trivial to derive the typical reflexivity, cancellation and fusion laws, more amenable to equational reasoning:

$$\llparenthesis in_{\mathsf{F}} \rrparenthesis_{\mathsf{F}} = id \qquad \llparenthesis\cdot\rrparenthesis\text{-}\textsc{Reflex}$$

$$\llparenthesis g \rrparenthesis_{\mathsf{F}} \circ in_{\mathsf{F}} = g \circ \mathsf{F}\ \llparenthesis g \rrparenthesis_{\mathsf{F}} \qquad \llparenthesis\cdot\rrparenthesis\text{-}\textsc{Cancel}$$

$$f \circ \llparenthesis g \rrparenthesis_{\mathsf{F}} = \llparenthesis h \rrparenthesis_{\mathsf{F}} \;\Leftarrow\; f \circ g = h \circ \mathsf{F}\ f \qquad \llparenthesis\cdot\rrparenthesis\text{-}\textsc{Fusion}$$

A well-known example of a fold is the $map\ f : [A] \rightarrow [B]$ function that maps an argument function $f : A \rightarrow B$ over the elements of a list and can be defined as follows:

$$map\ f = \llparenthesis in_{\mathsf{List}_B} \circ (id + f \times id) \rrparenthesis_{\mathsf{List}_A} \qquad map\text{-}\textsc{Def}$$

Mapping can be generalized from lists to arbitrary types. Specifically, a polymorphic type $T\ A = \mu(\mathsf{B}\ A)$ can be made into a *type functor* $T : \textsc{Set} \rightarrow \textsc{Set}$ that associates to every type A a type $T\ A$, and whose action on functions $T\ f : T\ A \rightarrow T\ B$ is given by the following catamorphism, for any function $f : A \rightarrow B$:

$$T\ f = \llparenthesis in_{\mathsf{B}\ B} \circ \mathsf{B}\ f\ id \rrparenthesis_{\mathsf{B}\ A} \qquad \textsc{Map-Def}$$

We can prove that T is indeed a functor:

$$T\ id = id \qquad \text{MAP-FUNCTOR-ID}$$

$$T\ f \circ T\ g = T\ (f \circ g) \qquad \text{MAP-FUNCTOR-COMP}$$

From the fusion law for catamorphisms, we can derive a useful law stating that a catamorphism composed with its type functor can always be fused into a single catamorphism:

$$(\!|g|\!)_{\mathsf{B}\ B} \circ T\ f = (\!|g \circ \mathsf{B}\ f\ id|\!)_{\mathsf{B}\ A} \qquad (\!|\cdot|\!)\text{-MAP-FUSION}$$

Another example of a catamorphism is the function *filter_left* : $[A + B] \rightarrow [A]$ that filters all the left alternatives from a list of left or right elements. It can be defined in Haskell as follows:

```
filter_left :: [Either a b] → [a]
filter_left []               = []
filter_left (Left x : xs)    = x : filter_left xs
filter_left (Right x : xs) = filter_left xs
```

Using the natural isomorphisms presented before, it is not difficult to put together a point-free algebra with the intended behavior:

$$\mathit{filter\_left} = (\!|(in_{\mathsf{List}_A} \triangledown \pi_2) \circ coassocl \circ (id + distl)|\!)_{\mathsf{List}_{A+B}} \qquad \mathit{filter\_left}\text{-DEF}$$

The behavior of the *filter_left* catamorphism is clarified in the following diagram:

$$\begin{tikzcd}
{[A + B]} \arrow[d, "\mathit{filter\_left}"'] & & 1 + (A + B) \times [A + B] \arrow[ll, "in_{\mathsf{List}_{A+B}}"'] \arrow[d, "(\underline{1} \oplus \underline{A+B} \otimes \mathsf{Id})\ \mathit{filter\_left}"] \\
{[A]} & (1 + A \times [A]) + B \times [A] \arrow[l, "in_{\mathsf{List}_A} \triangledown \pi_2"] & 1 + (A + B) \times [A] \arrow[l, "coassocl \circ (id + distl)"]
\end{tikzcd}$$

Using fold fusion, we can also prove that *filter_left* is a natural transformation, by showing that mapping two functions to distinct alternatives of a sum before filtering is the same as mapping only the left function after filtering:

$$\mathit{filter\_left} \circ map\ (f + g) = map\ f \circ \mathit{filter\_left} \qquad \mathit{filter\_left}\text{-NAT}$$

Paramorphisms A notion which extends that of a catamorphism is the *paramorphism*, that can express all functions defined by primitive recursion (Meertens, 1992). In practice, this means that the result can depend not only on the recursive result, but also on the recursive occurrence of the type. In the SET category, for an inductive type $\mu\mathsf{F}$, given a function $g : \mathsf{F}\ (A \times \mu\mathsf{F}) \to A$, the paramorphism $\langle\!|g|\!\rangle_{\mathsf{F}} : \mu\mathsf{F} \to A$ is the unique function that makes the hereunder diagram commute:

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[d, "\langle\!|g|\!\rangle_{\mathsf{F}}"'] & \mathsf{F}\ \mu\mathsf{F} \arrow[l, "in_{\mathsf{F}}"'] \arrow[d, "\mathsf{F}\ (\langle\!|g|\!\rangle_{\mathsf{F}} \vartriangle id)"] \\
A & \mathsf{F}\ (A \times \mu\mathsf{F}) \arrow[l, "g"]
\end{tikzcd}$$

Notice how a copy of the recursive occurrence is made before the recursive invocation. The paramorphism recursion pattern is characterized by the following laws:

$$f = \langle\!|g|\!\rangle_{\mathsf{F}} \ \Leftrightarrow\ f \circ in_{\mathsf{F}} = g \circ \mathsf{F}\ (f \vartriangle id) \qquad \langle\!|\cdot|\!\rangle\text{-UNIQ}$$

$$\langle\!|in_{\mathsf{F}} \circ \mathsf{F}\ \pi_i|\!\rangle_{\mathsf{F}} = id \qquad \mathit{for}\ i = 1, 2 \qquad \langle\!|\cdot|\!\rangle\text{-REFLEX}$$

$$\langle\!|g|\!\rangle_{\mathsf{F}} \circ in_{\mathsf{F}} = g \circ \mathsf{F}\ (\langle\!|g|\!\rangle_{\mathsf{F}} \vartriangle id) \qquad \langle\!|\cdot|\!\rangle\text{-CANCEL}$$

$$f \circ \langle\!|g|\!\rangle_{\mathsf{F}} = \langle\!|h|\!\rangle_{\mathsf{F}} \ \Leftarrow\ f \circ g = h \circ \mathsf{F}\ (f \times id) \qquad \langle\!|\cdot|\!\rangle\text{-FUSION}$$

In a category with products and exponential objects like SET, a paramorphism can be encoded as a catamorphism that pairs with the result of each recursion step a copy of the input, to be used recursively. In the last step, the copy of the input is discarded:

$$\langle\!|f|\!\rangle_{\mathsf{F}} = \pi_1 \circ (\!|g \vartriangle in_{\mathsf{F}} \circ \mathsf{F}\ \pi_2|\!)_{\mathsf{F}} \qquad \langle\!|\cdot|\!\rangle\text{-DEF-CATA}$$

Whenever it ignores the recursive copy of the input, the paramorphism can be reduced to a simple catamorphism:

$$\langle\!|f \circ \mathsf{F}\ \pi_1|\!\rangle_{\mathsf{F}} = (\!|f|\!)_{\mathsf{F}} \qquad \langle\!|\cdot|\!\rangle\text{-CATA}$$

A famous example of a paramorphism is the factorial function that returns one for the argument zero and, for a natural number n greater than zero, calculates the factorial of $n - 1$ multiplied by the original argument n:

$\mathit{fact} :: \mathit{Nat} \to \mathit{Nat}$
$\mathit{fact}\ \mathit{Zero} = \mathit{Succ}\ \mathit{Zero}$

$$fact\ (Succ\ n) = mult\ (fact\ n, Succ\ n)$$

Here, the combinator $mult :: (Nat, Nat) \to Nat$ denotes multiplication of natural numbers. This Haskell point-wise definition can be translated to point-free as follows:

$$fact = (\!| succ \circ zero \triangledown mult \circ (id \times succ) |\!)_{\mathsf{Nat}} \qquad fact\text{-}\textsc{Def}$$

Anamorphisms The categorical dual of the catamorphism is the unfold or *anamorphism* recursion pattern, that provides a standard way of producing values of a recursive type.

Given a functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$, a F-*coalgebra* is a morphism of type $A \to \mathsf{F}\ A$, with the object A as carrier of the coalgebra, and a F-*cohomomorphism* from a F-coalgebra $f : A \to \mathsf{F}\ A$ to a F-coalgebra $g : B \to \mathsf{F}\ B$ is a morphism $h : A \to B$ such that the property $g \circ h = \mathsf{F}\ h \circ f$ holds.

A *terminal* F-*coalgebra* is a terminal object in the category with objects as coalgebras and cohomomorphisms as arrows. Coinductive data types such as streams, that allow potentially infinite values, can be represented by terminal coalgebras. We denote the unique terminal coalgebra of a functor F as $out_{\mathsf{F}} : \mathsf{F} \to \mathsf{F}\ \nu\mathsf{F}$, where $\nu\mathsf{F}$ is the coinductive data type represented as the greatest fixed point of F.

In the SET category, given a coalgebra $h : A \to \mathsf{F}\ A$, the anamorphism $[\!(h)\!]_{\mathsf{F}} : A \to \nu\mathsf{F}$ is a function that, given an element of A, builds a (possibly infinite) element of the coinductive data type $\nu\mathsf{F}$. The coalgebra h is used to decide when generation stops and, in case it proceeds, which "seeds" should be used to generate the recursive occurrences of $\nu\mathsf{F}$.

In this thesis, we will only be interested in a specific kind of unfolds, namely those that always terminate. Not only do we want all our anamorphisms to terminate, but we also want to be able to freely compose them with catamorphisms – this composition is not possible in general (in the SET category), because anamorphisms can generate infinite values of coinductive types that are not values of the inductive types consumed by catamorphisms.

Instead of following a *strong functional programming* approach (Turner, 1995; Barbosa, 2001), we restrict ourselves to *recursive* (Capretta et al., 2006) (or *reductive* (Backhouse and Doornbos, 2001)) *coalgebras*. The resulting morphism (to be called a *recursive anamorphism*) is then guaranteed to halt in finitely many steps. A recursive coalgebra $h : A \to \mathsf{F}\ A$ is essentially one that guarantees that all As contained in the resulting $\mathsf{F}\ A$ are somehow smaller than its input, what serves as a proof that the corresponding anamorphism terminates. Capretta et al. (2006) give a nice formal definition and provide a set of constructions for building recursive coalgebras out of simpler ones.

For the initial F-algebra $in_{\mathsf{F}} : \mathsf{F}\ \mu\mathsf{F} \rightarrow \mu\mathsf{F}$, the F-coalgebra $in_{\mathsf{F}}^{-1} : \mu\mathsf{F} \rightarrow \mathsf{F}\ \mu\mathsf{F}$ is a terminal recursive coalgebra: notationally, we will often reuse out_{F} to denote the recursive coalgebra in_{F}^{-1}. Given a recursive coalgebra $h : A \rightarrow \mathsf{F}\ A$, the recursive anamorphism $\llbracket h \rrbracket_{\mathsf{F}} : A \rightarrow \mu\mathsf{F}$ is the unique arrow that makes the hereunder diagram commute:

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[r, "in_{\mathsf{F}}^{-1}"] & \mathsf{F}\ \mu\mathsf{F} \\
A \arrow[u, "\llbracket h \rrbracket_{\mathsf{F}}"] \arrow[r, "h"'] & \mathsf{F}\ A \arrow[u, "\mathsf{F}\ \llbracket h \rrbracket_{\mathsf{F}}"']
\end{tikzcd}$$

The recursive anamorphism (over inductive types) obeys the same laws as the normal anamorphism (over coinductive types), defined as follows:

$$f = \llbracket h \rrbracket_{\mathsf{F}} \;\Leftrightarrow\; in_{\mathsf{F}}^{-1} \circ f = \mathsf{F}\ f \circ h \qquad \llbracket\cdot\rrbracket\text{-Uniq}$$

$$\llbracket in_{\mathsf{F}}^{-1} \rrbracket_{\mathsf{F}} = id \qquad \llbracket\cdot\rrbracket\text{-Reflex}$$

$$in_{\mathsf{F}}^{-1} \circ \llbracket h \rrbracket_{\mathsf{F}} = \mathsf{F}\ \llbracket h \rrbracket_{\mathsf{F}} \circ h \qquad \llbracket\cdot\rrbracket\text{-Cancel}$$

$$\llbracket h \rrbracket_{\mathsf{F}} \circ f = \llbracket h \rrbracket_{\mathsf{F}} \;\Leftarrow\; h \circ f = \mathsf{F}\ f \circ h \qquad \llbracket\cdot\rrbracket\text{-Fusion}$$

A trivial example of a recursive anamorphism between arbitrary polymorphic types is again type functor mapping:

$$T\ f = \llbracket \mathsf{B}\ f\ id \circ out_{\mathsf{B}\ A} \rrbracket_{\mathsf{B}\ B} \qquad \text{Map-}\llbracket\cdot\rrbracket\text{-Def}$$

Notice that $\mathsf{B}\ f\ id : (\mathsf{B}\ A) \dot{\rightarrow} (\mathsf{B}\ B)$ is a natural transformation between partially applied bifunctors and that a composition of a natural transformation with a recursive coalgebra is again a recursive coalgebra (Capretta et al., 2006, Proposition 3.9), so this is clearly a recursive anamorphism. In fact, every catamorphism $(\!| in_{\mathsf{G}} \circ \eta |\!)_{\mathsf{F}}$, where $\eta : \mathsf{F} \dot{\rightarrow} \mathsf{G}$ is a natural transformation can also be defined as a recursive anamorphism $\llbracket \eta \circ out_{\mathsf{F}} \rrbracket_{\mathsf{G}}$, and vice-versa:

$$(\!| in_{\mathsf{G}} \circ \eta |\!)_{\mathsf{F}} = \llbracket \eta \circ out_{\mathsf{F}} \rrbracket_{\mathsf{G}} \;\Leftarrow\; \eta : \mathsf{F} \dot{\rightarrow} \mathsf{G} \qquad (\!|\cdot|\!)\text{-}\llbracket\cdot\rrbracket\text{-Shift}$$

Given its definition as an unfold, type functor mapping can also be fused with anamorphisms as follows:

$$T\ f \circ \llbracket g \rrbracket_{\mathsf{B}\ A} = \llbracket \mathsf{B}\ f\ id \circ g \rrbracket_{\mathsf{B}\ B} \qquad \llbracket \cdot \rrbracket\text{-MAP-FUSION}$$

A classical example of a recursive anamorphism that cannot be defined using a catamorphism is the function $zip : [A] \times [B] \to [A \times B]$, that zips two lists together into a list of pairs:

$$\begin{aligned} &zip :: ([a],[b]) \to [(a,b)] \\ &zip\ (x : xs, y : ys) = (x,y) : zip\ (xs, ys) \\ &zip\ \_ \qquad\qquad\quad\ = [\,] \end{aligned}$$

This function can be redefined in the point-free style as follows:

$$zip = \llbracket (! + distp) \circ coassocl \circ dists \circ (out_{\mathsf{List}_A} \times out_{\mathsf{List}_B}) \rrbracket_{\mathsf{List}_{A \times B}} \qquad zip\text{-DEF}$$

The behavior of the zip anamorphism is illustrated in the following diagram:

$$\begin{tikzcd}
{[A \times B]} \arrow[rrr, "out_{1 \oplus \underline{A \times B} \otimes \mathsf{Id}}"] & & & 1 + (A \times B) \times [A \times B] \\
{[A] \times [B]} \arrow[u, "zip"] \arrow[rrr, "(!+distp) \circ coassocl \circ dists \circ (out_{\mathsf{List}_A} \times out_{\mathsf{List}_B})"'] & & & 1 + (A \times B) \times ([A] \times [B]) \arrow[u, "(1 \oplus \underline{A \times B} \otimes \mathsf{Id})\ zip"']
\end{tikzcd}$$

Here, $distp$ and $dists$ are the isomorphisms given by:

$$\begin{aligned} &distp : (A \times B) \times (C \times D) \to (A \times C) \times (B \times D) \\ &distp = (\pi_1 \times \pi_1) \vartriangle (\pi_2 \times \pi_2) && distp\text{-DEF} \\ &dists : (A + B) \times (C + D) \to (A \times C + A \times D) + (B \times C + B \times D) \\ &dists = (distr + distr) \circ distl && dists\text{-DEF} \end{aligned}$$

The coalgebra of zip guarantees that the output list stops being generated if at least one of the inputs is empty. Otherwise, both tails are used as “seeds” to recursively generate the tail of the output list. We can prove a naturality law for zip stating that mapping two functions in parallel either before or after zipping produces the same result:

$$zip \circ (map\ f \times map\ g) = map\ (f \times g) \circ zip \qquad zip\text{-NAT}$$

Hylomorphisms In algebraic programming, the composition of a catamorphism after an anamorphism is known as a *hylomorphism*, that can express arbitrary recursive functions. A hylomorphism solves a particular problem by first constructing an intermediate data structure from the source type, that is then destructed to produce a result of the target type. However as mentioned above, this composition is not possible in SET, as inductive and coinductive types do not coincide. Therefore, we will be interested in a particular kind of hylomorphisms that are guaranteed to terminate, namely those where the catamorphism is composed with a recursive anamorphism:

$$\llbracket g, h \rrbracket_{\mathsf{F}} = \llparenthesis g \rrparenthesis_{\mathsf{F}} \circ [\!(h)\!]_{\mathsf{F}} \qquad \llbracket \cdot, \cdot \rrbracket\text{-SPLIT}$$

These *recursive hylomorphisms* (the unique *coalgebra-to-algebra morphisms* of Capretta et al. (2006)) are quite amenable to program calculation because they enjoy a uniqueness law similar to the other recursion patterns:

$$\llbracket g, h \rrbracket_{\mathsf{F}} = f \;\Leftrightarrow\; g \circ \mathsf{F}\, f \circ h = f \qquad \llbracket \cdot, \cdot \rrbracket\text{-UNIQ}$$

An example of a function that is easy to encode as a hylomorphism is the uncurried addition of natural numbers:

```
plus :: (Nat, Nat) → Nat
plus (Zero, m)   = m
plus (Succ n, m) = Succ (plus (n, m))
```

The point-free hylomorphism for *plus* uses the intermediate fixed point $\mu(\underline{Nat} \oplus \mathsf{Id})$:

$$plus = \llbracket id \triangledown succ, (\pi_2 + id) \circ distl \circ (out_{\mathsf{Nat}} \times id) \rrbracket_{\underline{Nat} \oplus \mathsf{Id}} \qquad plus\text{-DEF}$$

The diagram for the *plus* hylomorphism is the following:

$$\begin{tikzcd}
Nat \times Nat \arrow[r, "distl \circ (out_{\mathsf{Nat}} \times id)"] \arrow[d, "plus"'] & 1 \times Nat + Nat \times Nat \arrow[r, "\pi_2 + id"] & Nat + Nat \times Nat \arrow[d, "(\underline{Nat} \oplus \mathsf{Id})\ plus"] \\
Nat & & Nat + Nat \arrow[ll, "id \triangledown succ"']
\end{tikzcd}$$

2.2 Point-free Relational Calculus

A generalization of the point-free functional calculus is the point-free relational calculus (Bird and de Moor, 1997; Oliveira, 2008, 2009), that allows the formalization of partially defined functions and multiple-valued relations, while preserving similar lifted combinators and algebraic laws. To accommodate this change, we move from the category SET of total functions to the more general category REL of relations where objects are types and arrows are relations. Since SET is a subcategory of REL, arrows in SET can be seen as a particular subclass of arrows in REL.

2.2.1 Relational Combinators

A relation $R : A \rightarrow B$ can be theoretically seen as a subset of the Cartesian product $A \times B$. We write $b\ R\ a$ if the pair (a, b) is in R and $R\ a$ for the set $\{ b \mid b\ R\ a \}$. The identity relation $id_A : A \rightarrow A$ is defined by $\{ (a, a) \mid a \in A \}$, and the composition of two relations $R : A \rightarrow B$ and $S : B \rightarrow C$ is defined by $c\ (R \circ S)\ a \ \equiv\ \exists b.\ b\ S\ a\ \wedge\ c\ R\ b$. Relational inclusion is defined set-theoretically by $R \subseteq S \ \equiv\ \forall a, b.\ b\ R\ a \Rightarrow b\ S\ a$. The smallest relation between two types A and B is the empty relation denoted by a bottom arrow $\bot : A \rightarrow B$, and the largest relation is their Cartesian product denoted by a top arrow $\top : A \rightarrow B$.

For every relation $R : A \rightarrow B$, there is a *converse* relation $R^\circ : B \rightarrow A$. Whenever the relation R is a bijective function (according to the notation introduced later in Section 2.2.2) its converse R° is the unique function R^{-1}. Two relations $R : A \rightarrow B$ and $S : A \rightarrow B$ can be combined using standard set operations such as *intersection* $(R\ \cap\ S)$, *union* $(R\ \cup\ S)$ and *difference* $(R - S)$.

Terminal and initial objects In REL, both the initial and terminal object are the empty set $0 = \{ \}$, and the empty relation $\bot$ is simultaneously the the unique arrow from 0 to A and from A to 0, for any type A.

Unit objects In an allegory (a category with some additional structure inspired by REL (Bird and de Moor, 1997), including a partial order $\subseteq$ to compare two morphisms with the same source and target objects), an object U is a *unit object* if it satisfies two properties. The first is that id_U is the largest arrow of type $U \to U$, this is:

$$R \subseteq id_U \;\Leftarrow\; R : U \to U$$

In the notation introduced later in Section 2.2.2, this is equivalent to saying that every such arrow is coreflexive. Second, for every object A, there must be an entire (and simple) arrow of type $A \to U$ (again using the nomenclature from Section 2.2.2).

In the REL category, a unit object is the singleton set 1 and $!_A : A \to 1$ is the unique arrow mapping every element of A to the sole element of the singleton set. The function $!_A : A \to 1$ is also unique in REL because the singleton set is a terminal object in its subcategory of functions SET. In REL, for any two types A and B, the largest relation over those types (their Cartesian product) is usually denoted by an arrow $\top : A \to B$. This relation can be defined by composing two unitary arrows $!_B^\circ \circ !_A$.

Products Modeling the product of two types A and B as the Cartesian product $A \times B$ and using the standard projection functions $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, the split combinator can be defined in REL to apply two relations in parallel:

$$R \vartriangle S = {\pi_1}^\circ \circ R \;\cap\; {\pi_2}^\circ \circ S \qquad \vartriangle\text{-DEF}$$

According to this definition, the split $(b, c)\ (R \vartriangle S)\ a$ is defined whenever $b\ R\ a$ and $c\ S\ a$. However, if a is not in the domain of both R and S, then the split is undefined.

Due to such partiality, our notion of relational product is not a true categorical product in REL. The universal property for the split combinator can be obtained from $\vartriangle$-DEF by indirect equality:

$$X \subseteq R \vartriangle S \;\Leftrightarrow\; \pi_1 \circ X \subseteq R \;\wedge\; \pi_2 \circ X \subseteq S \qquad \times\text{-UNIV}$$

Nevertheless, it satisfies reflexivity, cancelation and fusion laws modulo additional side-conditions entailing that the relations are defined for the required domains, and the product combinator can still be defined in a similar way to SET (Appendix A).

Coproducts Unlike products, coproducts in REL are a simple generalization of coproducts in SET. Remembering the definition of the coproduct $A + B$ as a disjoint sum and the usual injection functions $i_1 : A \to A + B$ and $i_2 : B \to A + B$, the either combinator can be defined as follows:

$$R \triangledown S = R \circ i_1{}^{\circ} \ \cup \ S \circ i_2{}^{\circ} \qquad \triangledown\text{-DEF}$$

Relational coproducts are truly categorical and satisfy the usual laws for coproducts in SET, that can be derived from the following universal property:

$$X = R \triangledown S \ \Leftrightarrow \ X \circ i_1 = R \ \wedge \ X \circ i_2 = S \qquad +\text{-UNIQ}$$

Coreflexives A very special class of relations with useful properties (Appendix A) are *coreflexives*. A relation $\Phi : A \rightharpoonup A$ is said to be coreflexive if it is at most the identity ($\Phi \subseteq id_A$). Informally, we can see such coreflexive as a subset of A and by abuse of notation we will sometimes write $a \in \Phi$ to signify that a value a belongs to the set modeled by the coreflexive Φ. We will denote coreflexives by upper-case Greek letters ($\Psi, \Phi, \Omega, ...$).

In (Macedo et al., 2012), we introduced a *lift* combinator $[R] : A \times B \rightharpoonup A \times B$ that turns any relation $R : A \rightharpoonup B$ to a coreflexive on products. This is defined as follows:

$$[R] = \pi_2{}^{\circ} \circ R \circ \pi_1 \ \cap \ id \qquad [\cdot]\text{-DEF}$$

An alternative way to put it is to say that $[R]$ is the largest coreflexive Φ such that $\pi_2 \circ \Phi \subseteq R \circ \pi_1$, according to the following universal property:

$$\Phi \subseteq [R] \Leftrightarrow \pi_2 \circ \Phi \subseteq R \circ \pi_1$$

Conditionals In the relational calculus, coreflexives act as filters of data and can be used to model predicates, such that values a for which $a \ \Phi \ a$ satisfy the predicate Φ. For example, predicates that always return true or false can be defined as id or $\bot$, respectively. We can also trivially specify the predicate stating that both components of a pair are equal using the coreflexive $[id] : A \times A \rightharpoonup A \times A$.

Therefore, it is useful to overload the guard combinator to work over predicates modeled as coreflexives (instead of predicates modeled as functions): for a coreflexive $\Phi : A \rightharpoonup A$, we define the total function $\Phi? : A \rightharpoonup A + A$ that returns a left value if the input value belongs to the coreflexive or a right value otherwise:

$$\Phi? = i_1 \circ \Phi \cup i_2 \circ (id - \Phi) \qquad ?\text{-DEF}$$

By $\triangledown$-DEF, the property ?-DEF is equivalent to:

$$\Phi? = (\Phi \triangledown id - \Phi)^{\circ}$$

Catamorphisms The regular categorical notion of catamorphisms in a category of functions can be generalized into catamorphisms in the more general category of relations. Consider that an initial algebra $in_{\mathsf{F}} : \mathsf{F}\ A \to A$ in the subcategory of functions is also an initial algebra in the category of relations. Given a functor F, a relation $R : \mathsf{F}\ A \to A$ is an F-algebra if there exists an unique F-homomorphism $(\!|R|\!)_{\mathsf{F}} : \mu\mathsf{F} \to A$ from R to the initial F-algebra in_{F}, that is characterized by the following universal property:

$$X = (\!|R|\!)_{\mathsf{F}} \Leftrightarrow X \circ in_{\mathsf{F}} = R \circ \mathsf{F}\ X \qquad (\!|\cdot|\!)\text{-UNIQ}$$

The resulting morphism is called a *relational catamorphisms*. From the universal property, we can derive the usual laws like reflexivity, cancelation and fusion:

$$\begin{aligned}
&(\!|in_{\mathsf{F}}|\!)_{\mathsf{F}} = in_{\mathsf{F}} && (\!|\cdot|\!)\text{-REFLEX}\\
&(\!|R|\!)_{\mathsf{F}} \circ in_{\mathsf{F}} = R \circ \mathsf{F}\ (\!|R|\!)_{\mathsf{F}} && (\!|\cdot|\!)\text{-CANCEL}\\
&R \circ (\!|S|\!)_{\mathsf{F}} = (\!|T|\!)_{\mathsf{F}} \Leftarrow R \circ S = T \circ \mathsf{F}\ R && (\!|\cdot|\!)\text{-FUSION}
\end{aligned}$$

Anamorphisms In the relational calculus, the anamorphism is usually defined as the converse of the relational catamorphism (Backhouse and Doornbos, 2001)[3]. Given a functor F and a F-coalgebra $S : A \to \mathsf{F}\ A$, the *relational anamorphism* $[\!(S)\!]_{\mathsf{F}} : A \to \mu\mathsf{F}$ is defined as the expression:

$$[\!(S)\!]_{\mathsf{F}} = (\!|S^{\circ}|\!)_{\mathsf{F}}{}^{\circ}$$

[3] Although this definition is technically not the categorical dual of the catamorphism, since it is not the relational extension of the anamorphism on functions, for simplicity we will reason about them in similar ways, specifically regarding termination. The theoretical implications are discussed in more detail in (Hoogendijk, 1997).

Backhouse and Doornbos (2001) provide a nice general theorem stating that a relational anamorphism is terminating if and only if the coalgebra S is F-*reductive*. Based on this notion of F-reductivity, they propose a calculus of F-reductive relations to reason about the termination of recursive relational programs. If a coalgebra S is F-reductive, we call the resulting morphism a *recursive relational anamorphism*, denoted by $\llbracket\!(S)\!\rrbracket_{\mathsf{F}}$.

Hylomorphisms The *relational hylomorphism* is defined as the composition of a relational catamorphism after a relational anamorphism:

$$\llbracket R, S \rrbracket_{\mathsf{F}} = \llparenthesis R \rrparenthesis_{\mathsf{F}} \circ [\!(S)\!]_{\mathsf{F}}$$

The hylomorphism is the smallest solution of the inequation $R \circ \mathsf{F}\ X \circ S \subseteq X$, for all X, that is:

$$\llbracket R, S \rrbracket_{\mathsf{F}} \subseteq X \Leftarrow R \circ \mathsf{F}\ X \circ S \subseteq X$$

Since relational catamorphism are always terminating relations, it follows that the hylomorphism is a terminating program if and only if the relational anamorphism is terminating, i.e., the coalgebra S is reductive. If we consider the coalgebra to be reductive, the resulting *recursive relational hylomorphism*

$$\llbracket R, S \rrbracket_{\mathsf{F}} = \llparenthesis R \rrparenthesis_{\mathsf{F}} \circ \llbracket\!(S)\!\rrbracket_{\mathsf{F}} \tag{$\llbracket\cdot,\cdot\rrbracket$-Split}$$

is a terminating relation that satisfies the following equality:

$$\llbracket R, S \rrbracket_{\mathsf{F}} = X \Leftrightarrow R \circ \mathsf{F}\ X \circ S = X \tag{$\llbracket\cdot,\cdot\rrbracket$-Uniq}$$

2.2.2 Properties of Relations

Using the relational converse as an instrumental combinator, we can characterize various properties of relations. For example, the domain and range for which a relation $R : A \rightarrow B$ is defined are denoted by the coreflexives $\delta R = R^\circ \circ R\ \cap\ id$ and $\rho R = R \circ R^\circ\ \cap\ id$, respectively.

A relation R is said to be *simple* (formally, $R \circ R^\circ\ \subseteq\ id$) if it is single-valued, i.e., a partially defined function. The converse $R : A \rightarrow B$ of a simple relation is always *injective* ($R^\circ \circ R\ \subseteq\ id$), i.e., it maps distinct values in A to distinct values in B. On the other hand, a relation $R : A \rightarrow B$ is *entire* or *total* ($id \subseteq R^\circ \circ R$) if it is totally defined for its domain type A, meaning that $\delta R = id$. The converse $R : A \rightarrow B$ of an entire relation is always *surjective* ($id \subseteq R \circ R^\circ$), meaning that it is totally defined for its codomain type B with $\rho R = id$.

An injective (partial) function $f : A \rightarrow B$ is always *left-invertible* (or just *invertible*), i.e., there is a (partial) function $g : B \rightarrow A$ that undoes its behavior, such that for every $x \in \delta f$ we have $g\ (f\ x) = x$. Conversely, a surjective function $f : A \rightarrow B$ is always *right-invertible*, i.e., there is a function $g : B \rightarrow A$ that is undoable by f, such that for every $y \in \rho f$ we have $f\ (g\ x) = x$. A function $f : A \rightarrow B$ that is simultaneously injective, entire, simple and surjective is said to be *bijective* (also referred to as a *bijection* or an *isomorphism*).

In our notation, we define both a relation $R : A \rightarrow B$ (in REL) and a function $f : A \rightarrow B$ (in SET) using the same kind of arrow. In particular, total functions in SET arise as arrows in REL that are entire and simple. Henceforward, we will denote relations by upper-case identifiers $R, S, T, \ldots$ and use the lower-cases $f, g, h, \ldots$ to denote partial functions (simple relations) in REL or total functions in SET. When f is a partial function, we write $(f\ \ a){\downarrow}$ if f is defined for the input value a, i.e., $a \in \delta f$. When f and g are partial functions, $f \sqsubseteq g$ denotes inclusion of functions and is defined by $\forall a.\ (f\ \ a){\downarrow} \Rightarrow f\ \ a = g\ \ a$.

2.2.3 Proving the Termination of Anamorphisms

A more standard technique for proving the termination of an anamorphism $[\!(S)\!]_{\mathsf{F}}$, equivalent to stating that its coalgebra $S : A \rightarrow \mathsf{F}\ A$ is recursive, is to prove that its *accessibility relation* is *well-founded* (Bird and de Moor, 1997). The accessibility relation of a coalgebra $S : A \rightarrow \mathsf{F}\ A$ can be formulated as the relation $\in_{\mathsf{F}} \circ S : A \rightarrow A$. A relation $R : A \rightarrow A$ is well-founded if, for any relation $X : A \rightarrow B$:

$$X \subseteq X \circ R \quad \Rightarrow \quad X \subseteq \bot$$

Put in other words, the inequation $X \subseteq X \circ R$ only has one solution, $X = \bot$, for a well-founded R. This property corresponds to the set-theoretic notion that all descending chains of elements in A are finite and have a minimal element. Some interesting laws that allow reasoning about well-founded relations can be found in (Bird and de Moor, 1997; Doornbos and Karger, 1998).

The *membership relation* $\in_{\mathsf{F}} : \mathsf{F}\ A \rightarrow A$ lets through all the members of the functor F, such that $x \in_{\mathsf{F}} y$ means that x is a member of the F-structure y. It can be defined by induction over the structure of polynomial functors (Bird and de Moor, 1997; Barbosa and Oliveira, 2006) as follows:

$$\begin{aligned}
&\in_{\mathsf{F}} : \mathsf{F}\ A \rightarrow A \\
&\in_{\mathsf{Id}} \quad = id \\
&\in_{\underline{C}} = \bot \\
&\in_{\mathsf{F} \otimes \mathsf{G}} = \in_{\mathsf{F}} \circ \pi_1 \ \cup \ \in_{\mathsf{G}} \circ \pi_2 \\
&\in_{\mathsf{F} \oplus \mathsf{G}} = \in_{\mathsf{F}} \triangledown \in_{\mathsf{G}} \\
&\in_{\mathsf{F} \odot \mathsf{G}} = \in_{\mathsf{G}} \circ \in_{\mathsf{F}}
\end{aligned} \qquad \in\text{-DEF}$$

For type functors, a membership can always be defined but it requires the introduction of a more intricate notion of transitive closure (Bird and de Moor, 1997). For example, for lists, the membership relation is given by $(\pi_1 \circ cons^\circ) \circ (\pi_2 \circ cons^\circ)^*$, where the transitive closure of $\pi_2 \circ cons^\circ$ computes a suffix of the list at an arbitrary depth and $\pi_1 \circ cons^\circ$ selects the first element of the suffix. By defining two membership relations for bifunctors $\in_{1\mathsf{B}} : \mathsf{B}\ A\ B \rightarrow A$ and $\in_{2\mathsf{B}} : \mathsf{B}\ A\ B \rightarrow B$, Hoogendijk (1997) elegantly generalized this definition to arbitrary type functors T, with $T\ A = \mu(\mathsf{B}\ A)$, as $\in_{1\mathsf{B}} \circ out_{\mathsf{B}\ A} \circ (\in_{2\mathsf{B}} \circ out_{\mathsf{B}\ A})^*$. In this thesis, we reformulate his definition to an equivalent one that uses a relational catamorphism:

$$\in_T = i_1{}^\circ \circ (\!| i_1 \circ \in_{1\mathsf{B}} \ \cup \ \in_{2\mathsf{B}} \ \cup \ i_2 \circ ! |\!)_{\mathsf{B}\ A}$$

The fact that our catamorphism builds a result of type $A + 1$, that is later destructed by $i_1{}^\circ$, is due to a technical issue relating to the stop condition of the catamorphism (for an insight about this problem see (Bird and de Moor, 1997, Exercise 6.17)), that if $\bot$ would reduce the whole catamorphism to $\bot$. Therefore, we specify an explicit stop condition for whenever an element does not exist in the type functor, to avoid the bottom case, and select only left elements of type A in the resulting sum.

The membership relation, for any functor, satisfies the following naturality property:

$$\left.\begin{aligned}
\in_{\mathsf{F}} \circ \mathsf{F}\ f = f \circ \in_{\mathsf{F}} \\
\in_{\mathsf{F}} \circ \mathsf{F}\ R \subseteq R \circ \in_{\mathsf{F}}
\end{aligned}\right\} \qquad \in\text{-NAT}$$

2.3 Summary

This chapter has introduced the theoretical notation that will be thoroughly used across this thesis.

The point-free functional calculus constitutes a nice and concise formalism to specify transformations (i.e., functions), as the unidirectional components of bidirectional transformations (Chapter 3). Also, it will allow to specify bidirectional properties in an unifying way, while the underlying calculus provides an equational methodology to write the proofs of such properties for particular transformations. In Chapter 4, the same notation will be adopted to elegantly express bidirectional transformations and to develop an algebra of bidirectional transformations.

The point-free relational calculus gracefully captures the notion of partiality in the design of bidirectional transformations. It will also be used to abstractly represent the consistency relation (and related properties) of a bidirectional transformation in Chapter 3, and will provide an elegant and general algebra for building correspondence relations in Chapter 5. Moreover, some proofs of termination for recursive anamorphisms, found in Appendix A and Appendix B, can be performed at the relational level.

Chapter 3

State of the Art

The burgeoning interest in bidirectional transformations has led to a vast number of approaches in several computer science disciplines, inspired by different visions of the problem and motivated by the different contexts where the need for bidirectionality arises. Acknowledging the heterogeneity of the field, Czarnecki et al. (2009) survey the related literature on bidirectional transformations. They give an enumeration of existing work grouped by subcommunities, identifying some of the grand challenges of the field and providing a modest discussion on the terminology, key concepts and semantic properties adopted across the represented communities. A more detailed picture of the specific subarea of bidirectional model transformations is given by Stevens (2008), with a special emphasis on tool support and inherent open challenges.

Although these surveys enumerate such bidirectional challenges and point to some of the existing solutions, they do not attempt to compare these various solutions in a unifying setting, which is essential to understanding precisely their advantages and limitations and providing effective criteria for assessing progress in the field. In general, the design of a bidirectional language lies on a neat tension between the expressiveness allowed by its syntax and data domain, the robustness enforced by the totality and semantic properties of its transformations and the decidability of its type system.

One such unifying setting is presented by Antkiewicz and Czarnecki (2008), for classifying a wide spectrum of model synchronization axiomatizations and features, including systems that support parallel updates. Although they adopt a more formal perspective, they do not consider the algebraic underpinnings of each instance and only discuss the signatures of transformations, leaving aside the semantic laws which are essential to comprehending their behavior. Conversely, Diskin (2011) explores the mathematical foundations for building delta-based synchronization frameworks, stating the laws they should satisfy. The same author, in (Diskin, 2008a), proposes an algebraic state-based classifying system in which fewer bidirectional axiomatizations can be compared and analyzed in terms of their semantic laws, giving a few examples of frameworks that realize the different properties of the proposed systematization. Nevertheless, these two works only consider semantic and not design properties, and many of the existing approaches that can be found in related literature do not fit into one or both classification systems.

This chapter attempts to go further than previous work and proposes both: a general taxonomy that is sufficiently abstract to capture the most relevant features in the design of bidirectional languages, but precise enough to express the semantic properties that rule their behavior; and a deep multi-perspective classification of the state of the art of the field according to the given taxonomy. In this taxonomy, a bidirectional transformation is defined as a pair of transformations that translate independent updates between a pair of models in order to bring them into a consistent state. Although other more general notions of synchronization where both models can be simultaneously updated are not the focus of our work, we still survey some examples of synchronization systems that are constructed using bidirectional transformations.

3.1 Taxonomy

This section proposes a taxonomy for the classification of the defining features of bidirectional transformation approaches, namely their scheme (framework, updates), properties (round-tripping, consistency and totality laws) and deployment (data model, typing, specification, language and bidirectionalization approach). When introducing each feature and respective classification axes, we will also set (in parenthesis) a corresponding abbreviated symbol notation to be used later in Section 3.2.

3.1.1 Scheme

In contrast with general synchronization procedures that let users evolve models simultaneously, thus supporting parallel updates, this taxonomy focuses on a particular kind of bidirectional transformation frameworks tailored for single update propagation.

$$\begin{tikzcd} S \arrow[r, bend left, "to"] \arrow[r, phantom, "\rightleftharpoons"] & T \arrow[l, bend left, "from"] \end{tikzcd}
\qquad
\begin{tikzcd} S \arrow[rr, bend left, "get"] \arrow[rr, phantom, "\trianglerighteq"] & & T \\ & T \times S \arrow[ul, "put"] \arrow[ur, "\pi_1"'] & \end{tikzcd}
\qquad
\begin{tikzcd} & S \times T \arrow[dl, "\pi_1"'] \arrow[dr, "\triangleright"] & \\ S \arrow[rr, phantom, "\gtrless"] & & T \\ & S \times T \arrow[ul, "\triangleleft"] \arrow[ur, "\pi_2"'] & \end{tikzcd}$$

Figure 3.1: Bidirectional frameworks, diagrammatically.

Framework

A bidirectional transformation from S to T encompasses a (not always explicit) consistency relation $R \subseteq S \times T$ between both types, that acts as a partial specification of two forward and backward unidirectional transformations (partial functions), whose purpose is, respectively, to propagate S updates into T updates such that the resulting values are consistent, and vice-versa. There are three important and well-established bidirectional transformation frameworks (depicted in Figure 3.1):

Mappings ($\rightleftharpoons$) There is a forward transformation $\mathit{to} : S \to T$ and a backward transformation $\mathit{from} : T \to S$ that propagate source/target updates into target/source updates.

Lenses ($\trianglerighteq$) There is a forward transformation $\mathit{get} : S \to T$ that propagates source updates into target updates and a backward transformation $\mathit{put} : T \times S \to S$ that propagates target updates, with knowledge of the original consistent source, into source updates.

Maintainers ($\gtrless$) There is a forward transformation $\triangleright : S \times T \to T$ and a backward transformation $\triangleleft : S \times T \to S$ that propagate source/target updates into target/source updates, with knowledge of the original target/source.

Bidirectional *mappings* are the simplest of these three frameworks since they translate source updates into target updates (and vice-versa) without additional information about the previous target (or source) state. An information-symmetric instantiation of this framework are bijective languages (Yokoyama et al., 2008; Atanassow and Jeuring, 2007; Brabrand et al., 2008; Kennedy, 2004; Wadler, 1987), whose transformations establish a bijection between subsets of S and T and define structure-preserving mappings, in the sense that these subsets contain essentially the same information but just present it differently. Such languages promote the interoperability between different formats and are easy to reason about because bijectivity is preserved by composition and inversion.

Less restrictive reversible languages (Berdaguer et al., 2007; Mu et al., 2004; Terwilliger et al., 2007) define information-asymmetric mappings, by considering that transformations are only reversible in a particular direction, namely when T refines S: the forward transformation is an injective function and the backward transformation is a suitable inverse. Such reversibility entails that the forward transformation is (at least) information-preserving: it does not lose information, so that there always exists a backward function that undoes its behavior and recovers the initial model. Note that one particular approach that falls under this class is the 2LT framework (Cunha et al., 2006a; Berdaguer et al., 2007).

Other instantiations of mappings where the forward transformation is not injective also exist (Kawanaka and Hosoya, 2006; Wang et al., 2010), but due to information loss there may be many possible backward behaviors and the backward transformation must eventually cope with such ambiguous update translation.

Lenses are a popular asymmetric framework proposed as a solution to the classical view-update problem from database theory (Bancilhon and Spyratos, 1981). Since the forward transformation of a lens builds a more abstract view with less information than the original source, its backward transformation is non-deterministic in general as long as different sources can be abstracted to the same view. To help taming this non-determinism, the backward transformation of a lens considers additional *traceability* information: namely, put is "stateful" in the sense that it takes not only a target updated, but also some "trace" information about the original source that existed before the update, to be capable of restoring the information dropped by the forward transformation. To propagate view values for which there is no counterpart in the original source (for a typical example of this problem see the map combinator of Foster et al. (2007)), some approaches consider an additional "stateless" backward transformation $create : T \rightarrow S$ (Bohannon et al., 2008) that invents a default source from a view, or extend their framework such that put admits a missing or initial value as a source (Foster et al., 2007; Hofmann et al., 2011).

Nevertheless, the asymmetric treatment of lenses only works well for (essentially) surjective, lossy transformations, otherwise the propagation of updates from S to T is not able to restore T details not reflectable in S. For more general transformations without a dominant flow of information, where each of the source and target models may contain information not present in the other, we end up with the symmetric framework of *maintainers* (Meertens, 1998): the forward transformation explains how to modify a target model such that it relates to a source model; and the backward transformation defines how to modify a source to make it relate to a target modification. Unlike mappings and lenses, for which the consistency relation is usually implicitly defined by the transformations (as shown in Section 3.1.4), maintainers often restore conformity up to an explicitly defined consistency relation that represents the common subparts between source and target types. Maintainers are arguably the chosen framework for mainstream bidirectional model transformation approaches (Stevens, 2007), such as the relational language bundled with the Query/View/Transformation (QVT) standard proposed by the OMG and bidirectional approaches based on Triple graph Grammars (TGGs) (Schürr, 1995; Schürr and Klar, 2008). The downside is that, unlike the other bidirectional frameworks, they do not support compositional reasoning (sequential composition of maintainers is impossible in general, at least in a constructive way), as already noted by Meertens (1998).

Update representation

When talking about bidirectional transformations, the term "update" represents the effect of a change on a source of information. We consider four possible definitions of an update:

State (S) An update is represented only by the post-state.

Delta (D) An update is represented by the pre- and post-states and a *delta* or *sameness relation* stating which components of both are conceptually the same.

Edit (E) An update is represented as a sequence of edit operations that was performed over a particular pre-state.

Function (F) An update is represented by a semantic value modeling an endofunction that can be applied to the pre-state to produce a post-state.

Approaches that fall within the first category are usually known as *state-based*, as opposed to *operation-based* where some knowledge of the exact changes that led to the update result is also recorded. Due to its simplicity, many bidirectional approaches assume a state-based framework (Foster et al., 2007; Bohannon et al., 2008; Pacheco and Cunha, 2010). Most operation-based frameworks are edit-based (Liu et al., 2007; Hidaka et al., 2010; Hofmann et al., 2012), and within these some annotate the post-state with tags that represent the edit operations (Meertens, 1998; Mu et al., 2004). A few delta-based frameworks have been proposed (Diskin, 2008a; Diskin et al., 2011b; Barbosa et al., 2010) to support a more lightweight update representation. In function-based approaches like (Wang et al., 2011), updates are only conceptually represented.

Above, we have presented state-based definitions for mappings, lenses and maintainers. A more general formulation that allows for instantiating a wider range of frameworks and update representations is given in (Pacheco et al., 2012b).

Although the extra knowledge available for operation-based frameworks can lead to better results and to the preservation of more precise properties (see the discussions on "preservation" of updates and "minimality" of update translation in (Meertens, 1998; Hu et al., 2008; Foster, 2009)), its provision usually demands a tight coupling with applications, so that they can track such changes. Transforming updates onto updates also makes them more natural with incrementality (Giese and Wagner, 2006) (rather than recomputing a new model when the correlated model changes, only a small "delta" is propagated), and allows bidirectional properties to be formulated in terms of updates. On the other hand, state-based frameworks are more flexible and support more usage scenarios such as integration with off-the-shelf applications that have not been designed with bidirectionality in mind, and are moreover less sensitive to "noise" in the updates.

The distinction between state- and operation-based approaches is not always obvious. Some hybrid approaches build a state-based system with a richer operation-based core (Xiong et al., 2007). As they discard all update information, some model differencing procedure is required to infer new hypothetical update operations. Similarly, an incremental system can have a simple state-base core, but keep track of updates merely as an optimization, to exploit the locality of updates (Wang et al., 2011). Also, some asymmetric approaches represent source and target updates differently (Mu et al., 2004; Liu et al., 2007; Fegaras, 2010).

3.1.2 Properties

By itself, the scheme/signature of the transformations gives some hints about the expressivity of the system. However, it says nothing regarding the semantic properties expected of the transformations. Such properties are extremely relevant for the end-user because they force some predictability on the behavior of the transformations, namely concerning bidirectionality. Naturally, some schemes are more suited for particular classes of transformations and are more likely to exhibit specific kinds of properties. We identify several general properties that, independently of the scheme, might be

Property \ Framework	Mappings	
	$from$	to
Stability	to-Invertibility	$from$-Invertibility
Invertibility	$to \circ from \sqsubseteq id$	$from \circ to \sqsubseteq id$
Undoability	to-Invertibility	$from$-Invertibility
History Ignorance	trivial	trivial
...	Lenses	
	put	get
Stability	$put \circ (get \vartriangle id) \sqsubseteq id$	put-Invertibility
Invertibility	$get \circ put \sqsubseteq \pi_1$	$put \circ (get \times id) \sqsubseteq \pi_1$
Undoability	$put \circ (get \circ \pi_2 \vartriangle put) \sqsubseteq \pi_2$	put-Invertibility
History Ignorance	$put \circ (id \times put) \sqsubseteq put \circ (id \times \pi_2)$	trivial
...	Maintainers	
	$\lhd$	$\rhd$
Stability	$\lhd \circ (\pi_1 \vartriangle \rhd) \sqsubseteq \pi_1$	$\rhd \circ (\lhd \vartriangle \pi_2) \sqsubseteq \pi_2$
Invertibility	$\rhd \circ (\lhd \times id) \sqsubseteq \pi_2 \circ \pi_1$	$\lhd \circ (id \times \rhd) \sqsubseteq \pi_1 \circ \pi_2$
Undoability	$\lhd \circ (\lhd \vartriangle \rhd) \sqsubseteq \pi_1$	$\rhd \circ (\lhd \vartriangle \rhd) \sqsubseteq \pi_2$
History Ignorance	$\lhd \circ (\lhd \times id) \sqsubseteq \lhd \circ (\pi_1 \times id)$	$\rhd \circ (id \times \rhd) \sqsubseteq \rhd \circ (id \times \pi_2)$

Table 3.1: Round-tripping properties for bidirectional transformations.

desirable from an end-user perspective. Nevertheless, when introducing these properties we instantiate them for our three particular state-based frameworks. A more global perspective is given in (Pacheco et al., 2012b). Each of the properties described in this subsection has a forward and a backward version and can be classified in a three-level scale. In the graphical notation later used in Table 3.4, a normal arrow ($\longrightarrow$) denotes that a property is satisfied in the respective direction, while a dashed arrow ($\dashrightarrow$) signals that only a weaker version holds. The absence of an arrow means that the property is either not ensured, ignored by the authors, trivially satisfiable in the framework or implied by other properties.

Round-tripping

Typically, a bidirectional transformation must satisfy round-tripping laws entailing that its forward and backward transformations are somehow compatible (Table 3.1):

Stability The transformations translate null updates to null updates.

Invertibility It is possible to undo the translation of an update by applying the opposite transformation.

Undoability It is possible to undo the translation of an update by translating the converse update.

History Ignorance Update translation does not depend on the update history, i.e., consecutive update translations can be merged into a single one.

For each framework, the notion of what is a *well-behaved* bidirectional transformation, i.e., the minimum set of properties that it must satisfy to be considered reasonable, usually requires at least one of the transformations to be *stable* or *invertible*. If a framework also satisfies history ignorance, then it is usually considered *very well-behaved*.

When instantiating these laws for a concrete framework, one must consider how to model the converse of an update, how to compose updates and how to represent null updates. While these notions are natural for operation-based approaches, they often degenerate into slightly different concepts in state-based approaches. To give a hint about how the above informal round-tripping laws are instantiated to the specific laws from Table 3.1 for our three state-based frameworks, we make the following considerations: (1) since state-based updates are represented by only the post-state, composition of updates can be defined directly as $u_1 \circ u_2 = u_1$; (2) update converse can be trivially defined if we have access to the pre-state of the update; and (3) a null update happens when the pre- and post-states of the update are the same. Due to this forgetting of edit information (by representing updates as states), state-based laws tend to be stronger than their operation-based counterparts, as they must hold modulo all updates that generate the same result. A typical example is the state-based *history ignorance* law, that requires the translation of two consecutive updates to be the equivalent to the last issued update alone (since $u_1 \circ u_2 = u_1$), as discussed in (Diskin et al., 2011b).

For mappings, assuming that the pre-state of a view update is modeled by $to\ s$ (for an original source s) and the pre-state of a source update is modeled by $\mathit{from}\ t$ (for an original target t), forward and backward *stability* are reduced to backward and forward invertibility, respectively. Since no traceability information is passed to the transformations, *undoability* follows the same rationale. *History ignorance* becomes trivial, since only the post-states are used for update translation. We consider two dual asymmetric well-behaved mapping scenarios: if to is invertible, then T is a *refinement* of S ($S \leqslant T$), meaning that it contains more information; conversely, if $from$ is invertible, then T is an abstraction of S ($S \geqslant T$), meaning that it contains less information. In a well-behaved symmetric mapping, both transformations must be invertible and the bidirectional transformation must form an isomorphism ($S \cong T$), such that *to* and *from* are bijections (at least when restricted to the respective domains).

For lenses, put-Stability can be encoded by passing both the view pre-state $get\ s$ and the original source s to put, while get-Stability is similar to the one for mapppings. Lenses are undoable (Matsuda et al., 2007) if, for an updated view v' and an original source s, the view update v' can be cancelled by translating the original view $get\ s$ using as traceability the updated source $put\ (v', s)$ (since put-Invertibility implies that all source updates are undoable). Lenses are history ignorant (called PUTPUT by Foster et al. (2007)) if the translation of a (composite) view update does not depend on the intermediate source. A lens is well-behaved if put is stable and invertible, and these laws are usually known as GETPUT and PUTGET (Foster et al., 2007), respectively. Likewise mappings, put-Invertibility implies that a lens is an abstraction (type T is a view of type S), meaning that it cannot ignore view updates and must translate them exactly, such that a view-to-view round-trip always yields the same view. As an asymmetric framework, get-Invertibility is usually not postulated for lenses since it would imply the transformations to be isomorphisms. Nevertheless, put-Stability entails that get is invertible in a much weaker sense, only for a subclass of consistent view-source pairs: abstracting a source update and immediately putting it back under the same source yields the same updated source. Lenses for view update translation under a constant complement (Bancilhon and Spyratos, 1981; Matsuda et al., 2007), such that they preserve all the "hidden" data in the source that is not related by a forward transformation, are also history ignorant.

For maintainers, the laws of $\lhd$ are similar to the laws of put for lenses, and the laws of $\rhd$ are symmetric. A maintainer is well-behaved if it is correct and stable: correctness will be introduced later; $\lhd$-Stability entails that if the target does not change after forward transformation, then backward transformation shall return the original source, and vice-versa for $\rhd$-Stability. Since in a maintainer both S and T may contain information not present in the other, invertibility is often deemed too strong. Diskin (2008a) shows that stable and history ignorant lenses and maintainers are undoable.

Sometimes, weaker versions of the laws may be satisfied instead. For example, we can have *weak* variants of the round-tripping laws, namely whenever they are

Property \ Framework	Maintainers	
	$\lhd$	$\rhd$
Correctness	$\lhd \subseteq R^\circ \circ \pi_2$	$\rhd \subseteq R \circ \pi_1$
Hippocraticness	$[R] \subseteq {\pi_1}^\circ \circ \lhd$	$[R] \subseteq {\pi_2}^\circ \circ \rhd$

Table 3.2: Consistency properties for bidirectional transformations.

only satisfied modulo details that are inessential for the application scenario, like ordering (Bohannon et al., 2008; Ennals and Gay, 2007), whitespaces (Brabrand et al., 2008; Foster et al., 2008) or structure sharing (Kennedy, 2004). For some specific approaches that allow side-effects, such weak laws may even amount to applying another forward or backward transformation when comparing states. In (Hu et al., 2008), such a kind of weak stability (dubbed *bi-idempotence*) is used to guarantee that propagating a null view update followed by applying a forward transformation yields again a null view update, and dually for source updates. Diskin et al. (2011b) consider weak "one-and-a-half" and "undo-then-transform" variants of invertibility and undoability laws, respectively. For operation-based approaches, normal arrows denote lifted laws modulo updates and weak arrows may also mean that round-tripping does not preserve the full update, but only its post-state. For example, stability and invertibility for operation-based lens approaches are typically weakened in this sense (Barbosa et al., 2010; Hu et al., 2008; Mu et al., 2004).

Consistency

We can also formulate consistency properties that guarantee that the work of the transformations is coherent with the consistency relation (Table 3.2):

Correctness The transformations restore consistency.

Hippocraticness If an update does not break consistency, then it should be ignored by the transformation.

Correctness (called *range correctness* by Diskin (2008a)) and *hippocraticness* (coined by (Stevens, 2007)) properties are usually only considered in approaches where the consistency relation is explicitly defined (Ehrig et al., 2007; Hermann et al., 2011; Kawanaka and Hosoya, 2006; Meertens, 1998; Stevens, 2007). Thus, we only formulate these properties for maintainers, despite the fact that the laws for mappings and lenses (for which the consistency relation is usually the forward transformation) can be trivially derived by ignoring the unused traceability information, and often coincide with the round-tripping laws. For example, assuming that the implicit consistency relation in (asymmetric) lenses is $R = get$, then put-Correctness is the same as put-Invertibility. Although a well-behaved maintainer is required to be correct, hippocraticness is by itself particularly strong and is not always desired (Diskin, 2008a).

Our definitions of correctness for maintainers entail that, if a backward transformation is defined for a target update, then it returns an updated source consistent with the target update, and vice-versa. Written point-wise:

$$\forall s, t.\ (s \lhd t){\downarrow} \Rightarrow tR(s \lhd t) \qquad \lhd\text{-Correctness}$$

$$\forall s, t.\ (s \rhd t){\downarrow} \Rightarrow (s \rhd t)Rs \qquad \rhd\text{-Correctness}$$

Hippocraticness says that if a target update is consistent with the original source, then backward transformation returns the same original source, and vice-versa:

$$\forall s, t.\ tRs \Rightarrow s \lhd t = s \qquad \lhd\text{-Hippocraticness}$$

$$\forall s, t.\ tRs \Rightarrow s \rhd t = t \qquad \rhd\text{-Hippocraticness}$$

In our point-free encoding, we use the combinator $[R]$ to check if two source and target values are consistent according to the consistency relation R.

Totality

So far, we have defined the semantic laws modulo undefinedness of the unidirectional transformations. This is because totality requirements are by themselves an important feature in the design of a bidirectional transformation framework. In practice, it is often convenient to acknowledge that the type system might not be expressive enough to capture all the constraints induced by the transformations, and allow the source and target types to be larger than the actual domains of the transformations, leading to partially defined transformations. For example, non-surjectivity of get for a well-behaved lens (its range is smaller than the target type) implies partiality of put, otherwise for an updated target $t \notin \rho(get)$ and any source s we would find a different $t' = get\,(put\,(t, s))$ violating well-behavedness: there exists no corresponding source update for a target update outside the range of the query (a paradigmatic example is the constant combinator from (Liu et al., 2007)). While this might be a “show stopper”

Property \ Framework		Mappings	
		from	*to*
Totality	Safe	$\rho(to) \subseteq \delta(from)$	$\rho(from) \subseteq \delta(to)$
	Total	*from* total	*to* total

...		Lenses	
		put	get
Totality	Safe	$\rho(get \times id) \subseteq \delta(put)$	$\rho(put) \subseteq \delta(get)$
	Total	put total	get total

...		Maintainers	
		$\lhd$	$\rhd$
Totality	Safe	$\delta(R^\circ \circ \pi_2) \subseteq \delta(\lhd)$	$\delta(R \circ \pi_1) \subseteq \delta(\rhd)$
	Total	$\lhd$ total	$\rhd$ total

Table 3.3: Totality properties for bidirectional transformations.

for (usually state-based) batch applications that are expected to always produce results, it is acceptable for (usually operation-based) interactive applications that have direct control over the data manipulation interface: an editor does not need to handle every update and can signal an error to the user disallowing a specific modification. Partiality is though not admissible for security applications (Foster et al., 2009), since users might extract information about hidden data from the cases for which the transformations fail.

We distinguish three totality requirements that transformations must satisfy in increasing order of definedness (Table 3.3):

Partial The transformation is partial.

Safe The transformation is at least defined for the range of the opposite transformation.

Total The transformation is total.

In our graphical notation (Table 3.4), the absence of an arrow means that a transformation is partial. Safe transformations are denoted by weak dashed arrows, and total transformations by normal arrows.

Some approaches do not enforce any totality requirements (Mu et al., 2004). However, this can easily be abused: a *partial* bidirectional transformation can be trivially well-behaved if both transformations are always undefined.

A more reasonable statement is to require that transformations are *safe* (also known as *domain correct* in (Diskin, 2008a)). For asymmetric frameworks, one transformation generally dominates the data flow and has stronger totality requirements, and thus mapping and lens approaches usually assume the forward transformation to be total and the backward transformation to be either partial (Hidaka et al., 2010; Hu et al., 2008; Liu et al., 2007; Matsuda et al., 2007; Voigtländer, 2009) or safe (Berdaguer et al., 2007; Diskin et al., 2011a; Terwilliger et al., 2007). For lenses, the latter implies that put must be defined for any view in the range of get, independently of the original source value. For symmetric frameworks like maintainers, there is not generally a dominant data flow (for example, not every Java feature can be represented in a relational database schema, and vice-versa), and both transformations may be plausibly partial. Nevertheless, we can give a reasonable definition for safety: if there exists at least a consistent state for a source value, then forward transformation departing from that source must be defined (and reach such state due to correctness), and similarly for target values.

For *total* bidirectional transformations (Foster et al., 2007; Kawanaka and Hosoya, 2006; Meertens, 1998; Pacheco and Cunha, 2010), the types capture the exact domains over which the transformation is defined and guaranteed to behave well, ensuring that update translation cannot fail at run time.

3.1.3 Deployment

The remaining features of our taxonomy characterize how the different approaches operationalize an abstract bidirectional scheme into a concrete bidirectional transformation language or system.

Data domain

Bidirectional languages are usually conceived for the manipulation of specific kinds of data. The choice of a data domain helps in deciding on appropriate language and syntax, and on a type system that captures the necessary language constraints that entail well-behavedness. In our taxonomy, approaches are grouped into four broad classes of data models:

Strings (S) The transformations process textual data.

Trees (T) The transformations process tree-based data.

Relations (R) The transformations process relational data.

Graphs (G) The transformations process graph-based data.

String data (Bohannon et al., 2008; Brabrand et al., 2008; Foster et al., 2008; Kawanaka and Hosoya, 2006) abound in computer science, either as textual data formats, such as BibTeX, or plain representations of more structured formats like XML schemas. In some cases, manipulating the string data directly without a parsing phase is advantageous for programmers. Typical string-processing languages include regular string transducers (concatenation, iteration), and depend on machinery such as context-free grammars and regular expressions to define the formation rules for strings.

Trees (Foster et al., 2007; Hofmann et al., 2011; Hu et al., 2008; Mu et al., 2004; Voigtländer, 2009) are a natural way of representing more structured hierarchical formats, such as XML schemas, and many bidirectional functional languages handle essentially tree data. Typical bidirectional functional languages work with algebraic data types (that are essentially tree structures) and explore features such as pattern matching or polymorphism.

Much of the work on bidirectionalization and view update propagation has been developed over *relational* data (Dayal and Bernstein, 1982; Bohannon et al., 2006; Melnik et al., 2007; Terwilliger et al., 2007), concentrating on the correct handling of relational constraints such as functional and inclusion dependencies and on exploiting the algebraic properties of standard relational algebra operations such as joins and unions to identify subsets of transformations that satisfy particular bidirectional properties.

A more general data domain considers *graphs* (Diskin et al., 2011b; Ehrig et al., 2007; Hidaka et al., 2010), that natively support shared nodes and cycles and require careful algorithmic design to avoid possibly infinite traversals. Particularly in model-driven engineering, models are often formalized as graphs and thus model transformations are graph transformations. Examples of famous graph-based languages are the UnCAL graph algebra for querying graph databases (Buneman et al., 2000), triple graph grammars for graph-based model transformations (Schürr, 1995), and the ATLAS model transformation language (ATL) (Jouault and Kurtev, 2006).

Some asymmetric frameworks transform between different data domains, for instance from trees to strings (Kennedy, 2004) or from relations to trees (Fegaras, 2010). Other frameworks enrich their data domains with extensions so as to represent models that are more natural in other data domains. For example, the 2LT framework extends a tree domain with representations of relational models supporting referential constraints (Berdaguer et al., 2007) and spreadsheet models (Cunha et al., 2009).

Typing

Another relevant aspect of a bidirectional language is the precision of the type system that validates the construction of bidirectional programs and captures the domains for which the transformations are well-defined. If done statically at compile-time (in opposition to dynamically typed languages), type checking is important for catching errors in an early development stage and to ensure logical or modular properties of programs. For bidirectional languages, type systems may also be used to track elaborate properties about programs including well-behavedness, as conjectured by Foster et al. (2007). A type system also provides important structural information that can be used to narrow non-determinism during bidirectionalization (Liu et al., 2007) and to guide type-dependent optimizations (Melnik et al., 2007).

In our classification, we consider three degrees of typing:

Untyped (U) The domains of the transformations are not defined.

Typed (T) Types define the domains of the transformations.

Decidable ($\mathbb{T}$) Types define the domains of the transformations and type checking is decidable.

A transformation language may be *untyped*. For example, an untyped XML transformation language like XSLT 1.0 (Clark, 1999) transforms XML documents, but with no consideration for the classes of valid XML documents that are actually supported. However, this partiality of the transformations has a direct impact on their robustness, since users cannot have any reasonable expectations about which documents can be processed by a particular transformation.

Therefore, most bidirectional languages consider some degree of typing. For example, XSLT 2.0 (Kay, 2007) considers a *typed* approach and allows to define transformations between XML documents pertaining to particular XML Schemas. This time, even if transformations remain partial, the schemas give users a more realistic upper bound on the kinds of supported documents.

For transformations to be totally defined, the types must capture the exact domains, eventually demanding type systems for which type checking is undecidable, i.e., for some value or function, it is not always possible to give a correct answer saying whether or not it belongs to a given type. To remain *decidable*, some approaches impose additional type restrictions or annotate the domains of the transformations (Bohannon et al., 2008; Foster et al., 2008).

Specification

Bidirectional approaches vary depending on which artifacts need to be specified. We consider two particular cases:

Consistency relation (R) A bidirectional transformation is specified by the consistency relation.

Transformation (T) A bidirectional transformation is specified by its unidirectional transformations.

In some approaches, the consistency relation is explicitly specified, from which a bidirectionalization procedure derives a forward and a backward transformation. This is the case of some mainstream bidirectional model transformation frameworks based on OMG's QVT standard and TGGs (Schürr, 1995), such as (Stevens, 2007) and (Ehrig et al., 2007; Hermann et al., 2011), respectively.

Another popular approach is to specify one of the unidirectional transformations, from which the other can be derived, or similarly, to specify the bidirectional transformation in a particular language from which both transformations can be derived. For these cases (Foster et al., 2007; Hofmann et al., 2011; Hu et al., 2008; Meertens, 1998; Voigtländer, 2009), the consistency relation is implicitly defined.

Language

Bidirectional systems can be constructed using either a *domain-specific language* (van Deursen et al., 2000) or a *general-purpose language*:

Domain-specific language (D) A new bidirectional language is developed.

General-purpose langauge (G) An existing language is bidirectionalized.

Languages in the first class impose a more restricted specification style (Foster et al., 2007; Hu et al., 2008; Kawanaka and Hosoya, 2006; Meertens, 1998; Mu et al., 2004), and are often designed to contain special-purpose bidirectional constructs allowing programs to be read in both directions according to particular properties. The second class (Atanassow and Jeuring, 2007; Liu et al., 2007; Matsuda et al., 2007; Takeichi, 2009; Voigtländer, 2009) is more permissive, since it amounts to interpreting or extending unidirectional programs written in an existing standard language as/to bidirectional programs. Nevertheless, it is able to collect less information about the transformations, making certain behaviors or validations more difficult to specify.

Bidirectionalization Approach

Bidirectional systems can also be classified according to the techniques that are used to convert a specification into a bidirectional program (the process that we refer to as "bidirectionalization"[1]), while ensuring the semantic correctness of the resulting bidirectional transformations:

Ad hoc (A) The transformations are independently specified and their semantics must be proven correct by the developer.

Combinatorial (C) Each construct in the language denotes a bidirectional transformation with built-in semantics.

Syntactic (S) The transformations are computed at compile-time according to a syntactic analysis algorithm.

Semantic ($\mathbb{S}$) The transformations are interpreted at run-time for specific executions of a program.

An *ad hoc* solution like (Ennals and Gay, 2007) permits full freedom in the linguistic technologies used to code the transformations, but has serious disadvantages: it is doubly expensive because we have to write two transformations, error-prone, and likely to cause a maintenance problem since any change in a data format or program requires a redefinition of both transformations, and a new correctness proof.

A better approach is to design (or redesign) a *combinatorial* language in which every expression is a correct-by-construction bidirectional program (Foster et al., 2007; Hofmann et al., 2011; Hu et al., 2008; Mu et al., 2004; Pacheco and Cunha, 2010), denoting both transformations and (when applicable) a consistency relation. Since the combinators are natively bidirectional, bidirectionalization is trivial. An instrumental feature of these languages is compositionality, as it allows the construction of complex transformations via the composition of smaller ones and the reasoning about language properties by purely compositional means.

On the other hand, a more explicitly *syntactic* approach can be used for cases where a deeper whole-program analysis is necessary or for the bidirectionalization of

[1] In related literature (Hidaka et al., 2010; Matsuda et al., 2007; Voigtländer, 2009), sometimes the term "bidirectionalization" is used to characterize syntactic or semantic bidirectional approaches, in opposition to inherently combinatorial languages. Our nomenclature is more broad and considers all as possible bidirectionalization techniques.

a (syntactically restricted) general-purpose language according to program transformation techniques (Matsuda et al., 2007; Melnik et al., 2007; Atanassow and Jeuring, 2007; Brabrand et al., 2008; Hermann et al., 2011). Many of these approaches do not support composition and users are often required to specify a consistency relation, from which the system derives suitable unidirectional transformations by syntactic means. Despite giving hand of compositional reasoning, most syntactic approaches employ program transformation techniques to prove that their algorithms derive well-behaved transformations by construction.

An alternative *semantic* approach is to consider the semantic values of programs instead of their syntax. Thus, programs are bidirectionalized on-the-fly, by encoding the unidirectional transformations as algorithms that observe the inputs/outputs for specific run-time executions. Such a syntax-agnostic style allows the bidirectionalization of arbitrary programs, including those whose source code cannot be analyzed, but requires possibly strong static restrictions (Fegaras, 2010; Voigtländer, 2009).

Hybrid approaches can also be considered. For instance, some combinatorial approaches (Kennedy, 2004; Terwilliger et al., 2007) make crucial use of user-defined functions, for which additional proof obligations are required; or are essentially ad hoc (Wadler, 1987) with some correct-by-construction cases. A mixed combinatorial and semantic approach is presented in (Hidaka et al., 2010), while (Voigtländer et al., 2010) resorts to both syntactic and semantic manipulation. Different but related, an approach may resort to additional syntactic or semantic checks to rule out non-supported transformations (Fegaras, 2010; Wang et al., 2010).

3.1.4 Exploring the Design Space

As presented thus far, any abstract bidirectional framework encompasses a consistency relation $R \subseteq S \times T$ between source and target types, such that the unidirectional transformations propagate updates in order to bring inconsistent values into a consistent state. However, we have not presented how such consistency is instantiated for bidirectional frameworks other than maintainers, in particular those that consider an explicit data flow. We now unveil the formal properties required of such consistency relations in order to witness well-behaved transformations, and study the connections among the different frameworks.

For refinements, the relation R must ensure that T contains more information than S, that is, each source in S can be losslessly represented in T. Hence, it must be total (all sources are representable) and injective (each source is distinctly represented), as identified in the following round-tripping theorem from (Melnik et al., 2007):

Theorem 1. *The relation R is total and injective if and only if there exists a total refinement*

$$\begin{tikzcd} S \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & T \arrow[l, bend left, "from"] \end{tikzcd}$$

, such that $to \subseteq R \subseteq from^\circ$.

Note that the relation R must be injective (but not necessarily simple), meaning that each source may have more than one distinct target representation. For any total refinement, to is an injective function choosing one such representation for each source, and $from$ is a surjective function capable of restoring each transformed source.

Dually for abstractions, with S larger than T, the relation R must be simple (each target is uniquely determined) and surjective (all targets are views of sources), as the converses of total and injective relations always are (Oliveira, 2008):

Corollary 1. *The relation R is simple and surjective if and only if there exists a total abstraction*

$$\begin{tikzcd} S \arrow[r, bend left, "to"] \arrow[r, phantom, "\geqslant"] & T \arrow[l, bend left, "from"] \end{tikzcd}$$

, such that $from^\circ \subseteq R \subseteq to$.

Conversely to refinements, to and $from$ must now be surjective and injective functions, respectively.

Whenever the relation forms both a refinement and an abstraction, it determines a unique isomorphism:

Corollary 2. *The relation R is a bijection if and only if there exists a total isomorphism*

$$\begin{tikzcd} S \arrow[r, bend left, "to"] \arrow[r, phantom, "\cong"] & T \arrow[l, bend left, "from"] \end{tikzcd}$$

, such that $to = R = from^\circ$.

For isomorphisms, functions to and $from$ are both bijective and the inverse of each other. Note that a (total) function is an isomorphism iff its converse is a (total) function (Oliveira, 2008, Exercise 14).

Lenses have subtly more structure than abstractions, and their totality requires the relation to be additionally total, as evidenced by the following theorem:

Theorem 2. *The relation R is simple, total and surjective if and only if there exists a total well-behaved lens*

$$\begin{tikzcd} S \arrow[rr, bend left, "get"] \arrow[rr, phantom, "\rhd"] & & T \\ & T \times S \arrow[ul, "put"] \arrow[ur, "\pi_1"'] & \end{tikzcd}$$

, such that $\pi_1 \circ put^\circ \subseteq R = get$.

Proof. The forward implication can be shown by taking $get = R$ and defining put as any total function satisfying the equation $put \subseteq (\pi_2 \triangledown get^\circ \circ \pi_1) \circ [R^\circ]?$. The backward implication is straightforward from the properties of total well-behaved lenses. □

The fact that $R = get$, making R necessarily total, is a consequence of GETPUT: $\pi_1 \circ put^\circ \subseteq R$ tells us that the updated source returned by put and the updated target passed to put must form a consistent pair in R; if for any source s we have $put\ (get\ s, s) = s$, then s must be consistent with $get\ s$, or alternatively $get \subseteq R$.

Indeed, Oliveira (2008) showed that lenses meet the connectivity requirements of two $\geqslant$-diagrams:

Corollary 3. *A total lens l is well-behaved if and only if there exist two total abstractions*

$$\begin{tikzcd} S \arrow[r, bend left, "get_l"] \arrow[r, phantom, "\geqslant"] & T \arrow[l, bend left, "put_l \circ {\pi_1}^\circ"] \end{tikzcd} \text{ and } \begin{tikzcd} T \times S \arrow[r, bend left, "put_l"] \arrow[r, phantom, "\geqslant"] & S \arrow[l, bend left, "get_l \vartriangle id"] \end{tikzcd} .$$

Therefore, for a total well-behaved lens, get is surjective and put is surjective and *semi-injective* (Foster et al., 2007), in the sense that it is injective regarding only its first argument (i.e., $put \circ {\pi_1}^\circ$ is injective).

A lens is *oblivious* (Foster et al., 2007) when put is agnostic to its source argument, that is, for any endofunction $s : S \to S$ the following property holds:

$$put = put \circ (id \times s) \qquad put\text{-Oblivious}$$

Oblivious lenses are simply bijective mappings satisfying both get-Invertibility and put-Invertibility:

Corollary 4. *A total well-behaved lens l is oblivious if and only if there exists a total isomorphism*

$$\begin{tikzcd} S \arrow[r, bend left, "get_l"] \arrow[r, phantom, "\cong"] & T \arrow[l, bend left, "put_l \circ {\pi_1}^\circ"] \end{tikzcd} .$$

In general, the consistency relation may be neither total nor surjective, and especially not simple in either of the directions. Meertens (1998) showed that any relation that is total in both directions has a total maintainer:

Theorem 3. *If $S \neq 0$ and $T \neq 0$ (where 0 denotes the empty type with no values), the relation R is total and surjective if and only if there exists a total, well-behaved and hippocratic maintainer for R.*

Meertens does not consider an arbitrary relation because his development is based on total maintainers, but this restriction is too strong in practice, since the source and target schemas may contain non-reflectable information that should not be related by the consistency relation.

We can relax this constraint by considering only safe maintainers:

Theorem 4. *If $S \neq 0$ and $T \neq 0$, there always exists a safe, well-behaved and hippocratic maintainer for any relation R.*

Proof. This proof can be conducted in a similar way to the proof done by Meertens, assuming a *biased selector* $\oint : A \times \mathcal{P}_+A \to A$ that receives a value $x \in A$ and a non-empty set of values $s \in \mathcal{P}_+A$ and selects x if $x \in s$ or the "closest" value otherwise. We define:

$$m \rhd n = \begin{cases} n \oint (mR) & \text{if } mR \neq \emptyset \\ \bot & \text{otherwise} \end{cases} \qquad m \lhd n = \begin{cases} m \oint (Rn) & \text{if } Rn \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

□

An interesting result proven by Diskin (2008a) is that, when the consistency relation is simple, a maintainer degenerates into a lens[2]. Totality and surjectivity of the consistency relation are on par with safety and totality of the put function of the lens:

Lemma 1. *Given a maintainer m with a simple consistency relation R there exists a lens $l(m)$. Conversely, given a lens l there exists a maintainer $m(l)$ with a consistency relation $get_{l(m)}$, such that:*

$$\begin{tikzcd}
l(m) = S \arrow[rr, bend left, "\rhd_m \circ \pi_1{}^\circ"] \arrow[rr, phantom, "\unrhd"] & & T \\
& T \times S \arrow[ul, "\lhd_m \circ swap"] \arrow[ur, "\pi_1"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
& S \times T \arrow[dl, "\pi_1"'] \arrow[dr, "get_l \circ \pi_1"] & \\
m(l) = S \arrow[rr, phantom, "\gtrless"] & & T \\
& S \times T \arrow[ul, "put_l \circ swap"] \arrow[ur, "\pi_2"'] &
\end{tikzcd}$$

Theorem 5. *Given a safe/total well-behaved maintainer m with a simple and total consistency relation, $l(m)$ is a well-behaved lens with a total get and a safe/total put. Also, $l(m)$ is undoable/history ignorant if m is such. Dually, given a well-behaved lens l with a total get and a safe/total put, $m(l)$ is a safe/total well-behaved maintainer. Additionally, $m(l)$ is undoable/history ignorant if l is such.*

[2] This is not entirely new, as lenses roughly correspond to the functional maintainers of Meertens.

An interesting way to define composite maintainers is through the non-sequential composition of lenses, giving rise to maintainers whose consistency relations are not simple (and thus are not definable as lenses). One usual scenario in model-driven development, recognized by both Meertens (1998) and Diskin (2008a), is when source and target types are transformed into a common abstract view that represents the features present in both types. Using lenses, this construction can be defined as follows:

Definition 1. *Let $f : S \trianglerighteq A$ and $g : T \trianglerighteq A$ be two lenses. Their* co-targetial composition *is a maintainer $f \bowtie g$:*

$$\begin{aligned} R_{f \bowtie g} &= get_g^\circ \circ get_f \\ \rhd_{f \bowtie g} &= put_g \circ (get_f \times id) \\ \lhd_{f \bowtie g} &= put_f \circ (get_g \times id) \circ swap \end{aligned}$$

The consistency relation binds source and target values that have the same abstract component, and would be defined in point-wise by $tRs \equiv get_f\ s = get_g\ t$. The forward function generates a view from the source, and propagates the view modifications to a new target. The backward function proceeds similarly. The co-targetial composition of lenses has the following properties:

Theorem 6. *Given two well-behaved lenses f, g with total gets and safe/total puts, their co-targetial composition $f \bowtie g$ is a safe/total, well-behaved and hippocratic maintainer. Also, $f \bowtie g$ is undoable/history ignorant if f, g are such.*

Diskin (2008b) also conjectures the assumedly less usual and dual scenario of composing lenses from a common concrete complement that is a superset of both the source and the target types. However, he neither gives a complete definition nor explores the properties of such construction. We complete his formulation with the following definition:

Definition 2. *Let $f : C \trianglerighteq S$ and $g : C \trianglerighteq T$ be two lenses. Their* co-sourcial composition *is a maintainer $f \diamond g$, being $create_f : S \to C$ and $create_g : T \to C$ any two functions such that $create_f \subseteq put_f \circ {\pi_1}^\circ$ and $create_g \subseteq put_g \circ {\pi_1}^\circ$:*

$$\begin{aligned} R_{f \diamond g} &= get_g \circ get_f^\circ \\ \rhd_{f \diamond g} &= get_g \circ put_f \circ (id \times create_g) \\ \lhd_{f \diamond g} &= get_f \circ put_g \circ (id \times create_f) \circ swap \end{aligned}$$

The consistency relation now implies an existential quantification: for source s and target t values to be related, it must be possible to find a mediating concrete value of which they are two valid projections, i.e., $t\ R\ s\ \equiv\ \exists c \in C.\ get_f\ c = s\ \wedge\ get_g\ c = t$. Due to the signature of the lenses' put functions, the translations have to consider additional $create$ functions that invent default complements from the existing values. The forward function creates a complement from a target to propagate the source changes into a new complement, and computes a new target projection. The backward function proceeds similarly.

Unfortunately, to ensure that the maintainer is well-behaved, we need to guarantee that the additional information introduced in the complement but not present in either the source or target types does not influence the propagation of source and target updates. For instance, if put_f does not preserve all the target information in the complement that is not overridden by the source update, such side-effects will break the stability of the maintainer. We can characterize pairs of *complementable* lenses that are free of complement side-effects as follows:

Definition 3. *For two lenses $f : C \trianglerighteq S, g : C \trianglerighteq T$ from a common complement C, we call them* complementable *if and only if the following equations are satisfied, for any two endofunctions $c_1, c_2 : C \to C$:*

$$get_f \circ c_1 = get_f \circ c_2 \Rightarrow get_f \circ put_g \circ (id \times c_1) = get_f \circ put_g \circ (id \times c_2)$$
$$get_g \circ c_1 = get_g \circ c_2 \Rightarrow get_g \circ put_f \circ (id \times c_1) = get_g \circ put_f \circ (id \times c_2)$$

This property entails that, for a source update, any two complements possessing the same target projection induce the same target update, and vice-versa. For an example of two complementable lenses, define $f = \pi_1$ and $g = \pi_2$ using our lens combinators from Chapter 4. A counter-example is to define $f = ! + !$ and $g = id \triangledown id$. Assuming that lenses are complementable, we can prove that their co-sourcial composition preserves all the desired properties.

Theorem 7. *Consider two complementable well-behaved lenses f, g and their co-sourcial composition $f \diamond g$. If the functions $get_f, get_g, create_f, create_g$ are total and put_f, put_g are safe, then $f \diamond g$ is a safe, well-behaved and hippocratic maintainer. If f, g are total and $C \neq 0$, then $f \diamond g$ is a total, well-behaved and hippocratic maintainer. Also, $f \bowtie g$ is undoable/history ignorant if f, g are such.*

We believe that these two constructions are important to leverage the building of maintainers to purely compositional means. Essential for such an approach is the ability to track or statically check when complementability does hold for many specific cases.

On a more applicational side, Stevens (2007) discusses the application of maintainers to formalize bidirectional model transformations written in the QVT relations (QVT-R) language. In QVT-R, users specify a relation between source and target models according to a set of rules establishing the commonalities between them, from which an implementation must derive forward and backward transformations. Although the QVT standard contains some informal statements regarding the intended behavior of QVT implementations, its design mainly concerns the expressiveness and conformance with model transformation practices, in detriment of the properties of the transformations. Indeed, Stevens stresses that these postulates are ambiguous and do not lead to clear formal definitions. To fill this gap, she points out that the "check-then-enforce" semantics of QVT-R entails consistent and hippocratic maintainers and advocates that valid transformations should also be total and obey a stronger notion of undoability:

$$\pi_1 \circ [R] \subseteq \lhd \circ (\lhd \circ {\pi_1}^\circ \times id) \qquad \lhd\text{-UNDO}$$

$$\pi_2 \circ [R] \subseteq \rhd \circ (id \times \rhd \circ {\pi_2}^\circ) \qquad \rhd\text{-UNDO}$$

3.2 Survey

In the previous section, we have proposed a taxonomy for the classification of the defining features of bidirectional transformation approaches. With the taxonomy in place, this section contributes with a survey of the most influential work in the field, organized by frameworks and discussed case-by-case within each framework. Table 3.4 takes a global picture of the state of the art of the field. It must be read with some caution, though, since it does not capture specific intricacies of particular approaches that are not representable in our taxonomy and since some approaches are omissive or ambiguous regarding particular features, what leads to some subjectivity in their classification. Despite not being essential for a global perspective on the field, these idiosyncrasies are noted in each separate discussion.

3.2.1 Mapping Frameworks

A famous language based on bidirectional mappings is XSugar (Brabrand et al., 2008), that translates between XML documents and ASCII textual representations. In XSugar, bidirectional transformations are specified using pairs of intertwined grammars describing both the formats of the XML and non-XML data, from which a forward transformation is obtained by parsing according to the rules in one grammar and a backward transformation by parsing according to the rules in the other. XSugar transformations are not purely bijective: performing a round-trip should yield the exact same document modulo an equivalence relation that captures the loss of information related to whitespaces, canonization/normalization of XML documents, renaming of unnamed items and reordering of unordered XML elements. To identify such essentially bijective dual grammars, the authors describe a static analysis for checking XSugar specifications in order to rule out non-reversible or ambiguous grammars.

A similar language biXid (Kawanaka and Hosoya, 2006) describes XML to XML mappings using pairs of intertwined grammars with native support for ambiguity and non-linear use of variables. A biXid grammar specifies a consistency relation between documents from which the system infers the unidirectional transformations. The source and target formats specified in the dual grammar typically have quite similar structures but with various discrepancies in detail, from which ambiguity may arise, involving choices of multiple representations, freedom of order or non-reflectable data. Though the consistency relation is non-deterministic, their algorithms deterministically select one of multiple possibilities and create default data when necessary. The language provides no explicit bidirectional properties, but its authors intend it to "morally" preserve the similarities between the formats in such a way that resulting documents are consistent.

Janus (Yokoyama et al., 2008) is a high-level imperative language for the specification of reversible lossless computations in a C-like syntax. Each construct in the language is carefully designed to be purely reversible and possesses forward and backward deterministic semantics, meaning that program inversion can be done in a simple compositional fashion. As data structures, Janus supports integers, stacks and arrays. The language includes assignments (for constants and reversible arithmetic and logic operations), conditionals and loops with explicit post-conditions, stack operations, call-by-reference procedures and local variable allocation/deallocation.

Kennedy (2004) proposes a language of combinators for serialization of internal data representations (data types) into proper persistent formats (streams of bytes or characters) and vice-versa, to serve storage or transmission needs. These combinators are encoded in Haskell, and include a set of primitives over base types and combinators for products and sums that allow the pickling of arbitrary algebraic types into strings, by exploiting their "sums-of-products" structure. To avoid the repetition of similar patterns and produce more efficient serialized data, the combinators also take into account structure sharing. The resulting picklers are mostly bijective by construction (with additional proof obligations for some cases) modulo structure sharing: pickling followed by unpickling returns the original data if successful; but unpickling followed by pickling can produce a string with a different sharing. They are also partial in essence: some picklers have specific domains (for example, the picking transformation for the $zeroTo\ \ n$ combinator is only defined for integers up to n); and the reverse unpickler is only defined for particular strings.

Wadler (1987)'s *views* propose extending functional languages with abstract types that describe alternative representations of data types and are amenable to equational reasoning and pattern matching. To define a view, users must provide a type definition together with explicit in and out functions to map between the abstract type and the viewed type. A view is considered well-defined when in and out form a bijection between a subset of the viewed type and a subset of the abstract type. Although defining such functions and proving their accompanying property are left to users, a special case is identified when they can be defined at once via an $inout$ clause for which bijectivity is guaranteed.

Atanassow and Jeuring (2007) propose UUXML, an automatic mechanism for inferring Haskell data types and respective data bindings from a class of XML Schemas, that provides a type-safe embedding for XML-processing applications. Since the resulting types are very verbose and unnatural for users, they identify a set of primitive isomorphisms within Haskell data types, and devise an additional step that automatically infers type-preserving coercions between the machine-derivable types and more natural user-defined data types, by normalizing both to a common algebraic representation of sums-of-products.

Guava (Terwilliger et al., 2007) allows developers to build bridges between user interfaces and the underlying databases in order to be able to issue updates and answer queries directly against the user interface. The framework maps the user interface into a tree structure from which it infers a *natural database schema*, that developers can refine into a proper *physical database schema* using a language of typed invertible algebraic operations. These operations take parameters for which additional constraints are assumed, such as the injectivity of functions, or checked by the type system, such as the existence of functional dependencies. Although they only postulate forward invertibility, the core operations are essentially bijective, as they are defined as inverses of each others for specific subsets. This design intention is corroborated by the contrast of such operations with other supported refinements, such as tuple augmentation, whose backward transformations are not invertible. Since in practice it is unreasonable to store a materialized instance of the natural schema, their combinators support incremental updating and translate user interface updates in the form of statements in a *Data Manipulation Language* (DML), like INSERT statements in SQL, into respective database update statements.

In (Cunha et al., 2012), we and others present a bidirectional transformation environment to maintain spreadsheet models (modeling the business logic of spreadsheets) and instances synchronized. The core of our environment is built as a mapping that transforms edits on spreadsheet models into edits on spreadsheet instances, and vice-versa, such that its transformations are total for a restricted language of distinct operations on both sides. Spreadsheet instances refine spreadsheet models (to-Invertibility), such that we can undo the translation of an operation on models with an application of $from$ that yields the original model operation. Our transformations are correct in respect to a conformity relation ::, where $s :: m$ means that a spreadsheet instance s conforms to a spreadsheet model m. Moreover, we postulate a $from$-Hippocraticness law entailing that any model operation such that the original instance still conforms to the modified model is translated to a null update, together with edit-based stability and history ignorance laws.

On a more algebraic tone, Mu et al. (2004) propose a point-free language of injective and left-invertible functions called Inv. For an injective function to, its converse function $from$ is only partially defined for the range of to. In order to deal with duplication and structural changes, they admit edit tags (insertion, deletion and in-place modification) to model target updates and define an extended semantics that enlarges backward transformation for such tags. Considering the extended semantics, to may fail for outputs of $from$. On top of to-Invertibility, the authors also formulate a $from$-Weak Invertibility property modulo edit tags to highlight that Inv transformations are capable of repairing target invariants:

$$\mathit{from} \circ \mathit{to} \circ \mathit{from} \sqsubseteq \mathit{from} \qquad \textsc{FromToFrom}$$

For the case of duplication, if only one element of a target pair is updated (and thus marked with an edit tag), a target-to-target round-trip is then able to propagate the modifications to the other element and return a consistent duplicated pair (without edit tags), since we know that another *from* would not modify the source value.

We and others have proposed the 2LT system (Berdaguer et al., 2007) for the two-level transformation of XML schemas and SQL databases. After translating the schemas to internal Haskell type representations, the core of the system consists of a library of point-free combinators modeling schema refinement steps that preserve structural information and referential constraints. The type-safety of the coupled value-level transformations is ensured with a deep embedding in Haskell.

Wang et al. (2010) propose RInv, a language of surjective and right-invertible point-free combinators for defining total abstractions. Their design intends to liberate the original view mechanism of Wadler from the over-restrictive isomorphism requirement and the undesired manual proofs, while preserving sound equational reasoning and pattern matching at the view level. As long as the schema of the language is based on non-injective mappings, the forward transformation might loose concrete details that are not recoverable by the backward transformation. This problem is, however, not crucial for their design because they only consider views that discard redundant information (justifying the omission of general projection functions), whose concrete implementations are simply more efficient representations such as join lists or double-list queues. RInv also supports non-surjective data type constructors as primitives, what implies an additional compile-time check to test the joint surjectivity of programs involving such constructors.

3.2.2 Lens Frameworks

Work on operation-based view update translation has a long tradition in the database community going back to the late ’70s and ’80s, when different authors explored the existing design space between the expressivity of the supported relational operations, the totality of the update functions and the strength of the semantic properties. Bancilhon and Spyratos (1981) remarked that if one considers translation under a *constant complement* that keeps (at least) all the information in the view, so that any changes to the information that the complement has kept are forbidden, then there is at most one database update that reflects a given view update. More permissive approaches that reject fewer view updates, but permit several reasonable translations, were studied by Gottlob et al. (1988) and Dayal and Bernstein (1982). Keller (1986) studied reasonable criteria for disambiguating view updates for views defined using selections, joins and projections. In particular, he proposed an interactive algorithm that, based on the view definition and on the schema information, runs a dialog with the user to choose a sensible view update policy. As of now, we will focus on a more recent trend of bidirectional approaches developed mainly by the programming languages and model-driven engineering communities. A detailed review of related database literature can be found in (Foster, 2009).

One of the first linguistic approaches to the view-update problem is the Focal tree transformation language (Foster et al., 2007), that introduced the framework of lenses (and well-known round-tripping laws) as the main constructs of a data synchronization framework called Harmony. Focal provides a rich set of lens combinators, from general functional programming features (composition, mapping, recursion) to tree-specific operations (splitting, pruning, merging), whose backward behavior is inferred from the compositional structure of the transformations. Each combinator is proven total and well-behaved against a complex but precise type system based on record types. Using such *semantic types* (Frisch et al., 2008), they are able to define the exact domains for which their lens combinators are well-behaved, allowing the definition of constant, duplication and conditional combinators as total well-behaved lenses. For instance, for the constant combinator in Focal, if the forward function introduces a constant value, then it cannot be modified at the cost violating well-behavedness – the range of the lens is a singleton set with a sole constant value. To deal with creation of source information, the authors extended the universe of source values with a placeholder Ω for missing information, that is recognized and propagated by put and handled judiciously by the constant lens: if the source value is missing, then it creates a new source value as a result. They also demonstrated that well-behaved lenses are equivalent to the *dynamic views* by Gottlob et al. (1988), and very well-behaved lenses are isomorphic to the stronger notion of *closed* view-updating by Bancilhon and Spyratos (1981).

More languages have been proposed by the same group, including a new bidirectional language with a set of relational algebra primitives (selection, join, projection) and a complex type system with value-level predicates and functional dependencies targeted at relational data (Bohannon et al., 2006). To guarantee that their relational lenses are total and well-behaved, they impose several restrictions on the typing rules for each combinator, like assuming that functional dependencies are representable in a *tree form*. The authors estimate that type-checking should be decidable for a sensible class of predicates, but do not pursue this argument.

A third language Boomerang (Bohannon et al., 2008) tackles the bidirectional transformation of string data. Boomerang is built using a set of regular operations (union, concatenation, iteration) and a type system based on regular expressions. To ensure decidability and well-behavedness, the typing rules for some combinators impose particular restrictions on the regular expressions, such as disjoint or unambiguously concatenable domains. This time, a create function is used for the cases when the original concrete model is unavailable, what avoids extending the type universe with a missing value and propagating it through the transformations. To overcome issues with order, they allow users to annotate lens expressions with key mechanisms – termed *dictionary lenses* – and extend put with a pre-processing step that parses source strings into a dictionary of chunks indexed by keys that is used to match source and view elements by keys rather than by positions. By design, dictionary lenses are *quasi-oblivious*, a property that obliges lenses to ignore concrete details in respect to an equivalence relation $\sim_S$ (the operation that parses a dictionary satisfies the equivalence relation that ignores the order of the concrete elements):

$$s \sim_S s' \quad \Rightarrow \quad \mathit{put}\ (v, s) = \mathit{put}\ (v, s') \qquad \text{QUASIEQUIVPUT}$$

In later work, Foster et al. (2008) recognized that, especially for string transformation languages like Boomerang (that manipulate the data directly without parsing/pretty-printing it to/from an intermediate abstract syntax tree), it is reasonable to assume that transformations only satisfy the round-tripping laws modulo specific details that are behaviorally inessential, and formalize such *quotient lenses* by considering lens domains and round-tripping laws modulo general equivalent relations. For that purpose, they consider a quotient lens $S_{\sim_S} \trianglerighteq V_{\sim_V}$, between sources of type S and views of type V coarsened by the equivalences $\sim_S$ and $\sim_V$, to be well-behaved if it satisfies loosened round-tripping laws

$$\mathit{put}\ (\mathit{get}\ s)\ s \sim_S s \qquad \text{GETPUTEQUIV}$$
$$\mathit{get}\ (\mathit{put}\ (v, s)) \sim_V v \qquad \text{PUTGETEQUIV}$$

and additional laws ensuring that it is faithfully oblivious to the equivalences:

$$s \sim_S s' \quad \Rightarrow \quad \mathit{get}\ s \sim_V \mathit{get}\ s' \qquad \text{EQUIVGET}$$

$$v \sim_V v' \wedge s \sim_S s' \quad \Rightarrow \quad put\ (v, s) \sim_S put\ (v', s') \qquad \text{EQUIVPUT}$$

They also consider the composition of lenses with pure abstractions – dubbed *canonizers* – to build quotient lenses that ignore the abstracted details from more concrete source or target domains. Because composition requires testing the equality of equivalence relations, only those with an intermediate equality relation are accepted by the type checker. The authors advocate that even such a simple technique supports interesting scenarios, namely lenses whose quotienting is biased to one side of the lens. Inspired by quotient lenses, *secure lenses* (Foster et al., 2009) extend Boomerang with a simple declarative way to specify view update policies determining which parts of the view can be updated that is formally expressed with equivalence relations and entails corresponding coarsened laws.

Matching lenses (Barbosa et al., 2010) generalize the key-based dictionary lenses to consider arbitrary alignment heuristics. Operationally, they assume an explicit separation of values into a rigid structure or shape, that is treated positionally by the lenses, from a list of chunks that populate the structure and can be freely rearranged according to the chosen alignment directive. They retool each Boomerang combinator with an auxiliary function that computes a source-to-view delta denoting the forward traceability, and with put functions that process view deltas (a canonical set of correspondences denoting the common chunks in the pre- and post-states) instead of mere states. Matching lenses obey particularly restrictive laws enforcing the propagation of all source chunks to the view (GETCHUNKS law) and that shape alignment is kept positional (SKELPUT law), together with delta-based round-tripping laws ensuring that deltas are correctly propagated. For instance, they postulate a law stating that put translates view reorderings into corresponding source reorderings, yielding delta-based put-Invertibility.

With the upsurging of lenses, other authors have proposed different visions of related bidirectional problems. A line of work regarding the design of bidirectional languages supporting duplication has been developed by a group of researchers from Tokyo. Hu et al. (2008) design a programmable editor for the interactive development of XML documents, so that users start from a view to which they can gradually apply modifications, while maintaining a source document and a linking transformation. The editor is built upon a lens language named X that provides typical functional combinators (composition, mapping, folding) and three primitives for Galois-connected pairs of functions (that are bi-idempotent), arbitrary constant functions (that disallow all view modifications) and duplication. Each combinator $l : S \trianglerighteq V$ in X is bidirectionalized through an embedding into the Inv language (Mu et al., 2004): after identifying an Inv refinement $r : S \leqslant V \times S$ that computes a view together with a copy of the input, they define $get_l = \pi_1 \circ to_r$ and $put_l = from_r$, considering the extended semantics with edit tags. Since X supports duplication and other view inter-dependencies, they only require transformations to satisfy a weak stability property modulo edit tags dubbed *bi-idempotence*, stating that source-to-source and view-to-view round-trips preserve null updates[3]:

$$put \circ (get \vartriangle id) \circ put \sqsubseteq put \qquad \textsc{PutGetPut}$$

$$get \circ put \circ (get \vartriangle id) \sqsubseteq get \qquad \textsc{GetPutGet}$$

This guarantees that the system "converges" into a final state, in the sense that after a round-trip further transformation does not change the state. For example, using their lenses, when the view is modified, a bidirectional system may apply put to calculate a new source followed by get to compute the view side-effects, knowing that a further put would not change the computed source. However, they acknowledge that a valid $put = \pi_2$ would not update the source at all, and conjecture a new *update preservation* property ensuring that an edited view should not fall back to a less updated one after a round-trip:

$$get \circ put \succeq \pi_1 \qquad \succeq\text{-}\textsc{PutGet}$$

Above, the ordering $\succeq$ on views determines if a view is more edited than other. We classify their lenses as partial, since put may fail for views within the range of get and (according to the Inv semantics) get may also fail for sources within the range of put. Anyhow, the transformations are used in an *online* setting, such that the editor reacts immediately after each update, forbidding certain unsupported cases like editing both sides of a duplicated pair.

Liu et al. (2007) propose Bi-X, a functional lens language closely resembling the XQuery Core language that can serve as the host language for the bidirectionalization of XQuery. The main feature of Bi-X is its support for variable binding, allowing lenses

[3] Although to-Invertibility of the embedded Inv expressions implies put-Stability of lenses, Galois-connected functions for instance do not satisfy such property.

that perform implicit duplication. Operationally, they annotate XML tree values with edit tags and establish an edit-based round-tripping property stating that put propagates all and only the view update tags to the source. Although their development is done in an untyped setting, they define a type system of regular expressions that is used to refine backward behavior for insertion tags. A more practical application of Bi-X is the Vu-X system (Nakano et al., 2009), that can be used to describe a bidirectional connection between a XML document and a set of HTML web pages. It provides a WYSIWYG interface for editing the HTML sources while reflecting the updates on the XML document (content updating), or the Bi-X transformation that describes the layouting of the database (code editing).

Following a similar approach, Hidaka et al. (2010) provide a bidirectionalization for the UnCAL graph algebra (Buneman et al., 2000), supporting structural recursion, conditionals and variable binding for integration in the GRoundTram system for bidirectional graph transformations. For the forward transformation, they consider the bulk semantics of UnCAL, that computes the result while preserving the shape of the original graph, and enrich it with additional traceability information to aid backward transformation. Each combinator is then given a corresponding backward semantics that uses the available traces for propagating in-place view updates. Since the language supports conditionals and implicit duplication, the authors postulate a put-Weak Invertibility property allowing view side-effects:

$$put \circ (get \circ put \vartriangle \pi_2) \sqsubseteq put \qquad \text{WPUTGET}$$

Despite the language is combinatorial, deletion and insertion updates are translated in a semantic way: for deletion, they compute the set of nodes in the source that corresponds to the deletion of a set of nodes in the view, and fail if a node is not correlated or if further forward transformation does not return the modified view graph, what would violate WPUTGET; for insertion, they extract the inserted view graph, and compute a corresponding inserted source graph using a universal resolving algorithm that explores all possible right inverses for the forward transformation and satisfies even the stronger PUTGET. Their approach also supports optimization. The key construct of their language is the structural recursion operation on graphs, that enjoys a fusion law on the underlying unidirectional graph algebra: two consecutive structural recursions $rec\ \ e_2 \circ rec\ \ e_1$ can be fused into a single structural recursion $rec\ \ (rec\ \ e_2 \circ e_1)$ that avoids computing an intermediate result, if the expression e_2 does not depend on its argument graph. This calculational law is applied to optimize get before bidirectionalization, but since it is not stated bidirectionally it may lead to different behaviors in the backward transformation. Although the bidirectionalization approach is completely untyped, Inaba et al. (2011) propose a static verification algorithm that checks if UnCAL transformations annotated with schema annotations consume and produce graphs obeying to source and target graphs schemas, respectively. Sasano et al. (2011) report a first effort towards the bidirectionalization of a restricted class of ATL transformations by converting them into UnCAL bidirectional programs.

A more pragmatic approach, described as *bidirectionality by programming*, is proposed by Takeichi (2009). He advocates that it is more flexible for language developers to only assume put-Stability as a basilar property, with other particular laws being only sensible for specific lens combinators. He exercises this methodology by bidirectionalizing the HaXML Haskell library, to serve as a back-end to a graphical interface for view-updating of XML documents. In the lifted HaXML, lens combinators are implemented as a higher-order bidirectional function of type $S \to V \times (V \to S)$, that given a source value returns a view and a backward function for that particular source. This partial evaluation of the backward function avoids inspecting the original source twice during forward and backward updating.

Other lens approaches without an emphasis on totality or duplication have also been proposed. Matsuda et al. (2007) studied the automatic derivation of backward transformations under a constant complement (Bancilhon and Spyratos, 1981), for functions defined in a restricted first-order functional language forbidding repeated variable occurrences (*affine*) and nested function calls (*treeless*). Given a $get : S \to V$ function, they derive an explicit complement function $cpl : S \to C$, such that the tupled function $get \vartriangle cpl$ is injective and $put_{\langle get,cpl \rangle} = (get \vartriangle cpl)^{-1} \circ (id \times cpl)$ is unique. However, there are many complement functions that preserve at least the information dropped by the view and make view propagation unique. The corresponding put functions have different updatability according to the complement, that needs to be kept constant as stated by the following law:

$$cpl \circ put \sqsubseteq cpl \circ \pi_2 \qquad \text{PUTCPL}$$

The more information preserved by the complement, the less modifications are permitted in the view, and thus the optimal complement is one such that $get \vartriangle cpl$ is bijective, meaning that the complement keeps exactly the information dropped by the view. To compare complements, Bancilhon and Spyratos define a *collapsing order* $\precsim$ between two functions g_1, g_2 as:

$$g_1 \precsim g_2 \;\Leftrightarrow\; \forall s, s'.\; g_2\ s = g_2\ s' \Rightarrow g_1\ s = g_1\ s' \qquad \precsim\text{-DEF}$$

The preorder $g_1 \precsim g_2$ means that g_1 will collapse more inputs than g_2. Minimal complements are constant functions that collapse all inputs into the same value, and maximal complements are injective functions, that do not collapse any inputs. A smaller complement function discards more data, producing a better backward transformation. Formally, Bancilhon and Spyratos establish that for the same view function f a backward function created from a smaller complement is always more defined:

$$g_1 \precsim g_2 \;\Rightarrow\; put_{\langle f, g_1\rangle} \sqsubseteq put_{\langle f, g_2\rangle} \qquad \precsim\text{-PUT}$$

The algorithm devised by Matsuda et al. employs injectivity and range analysis techniques to find smaller complement functions. They also provide a static procedure that checks if a view update is in the range of $get \vartriangle cpl$, and thus in the domain of put. The bidirectional properties follow those of closed view-updating and yield partial (undoable) very well-behaved lenses, since put should forbid any changes to the information that the complement has kept. For instance, inserting and removing elements are forbidden updates in their running example of a filtering lens.

To avoid restricting the syntax of the forward transformations, Voigtländer (2009) allows regular Haskell functions to be used in lens definitions. He instead considers parametrically polymorphic $get : \forall \alpha.\ [\alpha] \rightarrow [\alpha]$ functions over lists, that work on the "shape" of the data and are oblivious to the actual values of the polymorphic type α. Although this parametricity forbids many non-polymorphic lenses, such as mapping and content filtering, it allows the data values to be replaced by values of a type of positions, what permits observing the runtime behavior of the forward transformations. Based on such traceability information, he defines a put function as a higher-order bidirectionalizer $\mathit{bff} : (\forall \alpha.\ [\alpha] \rightarrow [\alpha]) \rightarrow (\forall \alpha.\ [\alpha] \times [\alpha] \rightarrow [\alpha])$, and exercises how to generalize bff to support arbitrary polymorphic functions. This semantic bidirectionalization yields very well-behaved lenses, but with a severe updatability limitation: put is only defined for updates that do not change the shape of the view. For example, the put function for a $halve$ function that computes the first half of a list would only be defined for the cases when the target list remains with exactly half of the size of the original list.

To overcome the updatability of the two previous approaches, Voigtländer et al. (2010) explore an hybrid syntactic and semantic approach that is applicable to the intersection of their domains, i.e., affine, treeless and polymorphic Haskell functions over lists. Operationally, the backward transformation for the combined technique extracts the shape of the source and target types (by replacing polymorphic type variables with the unit type, inducing a move from $[\alpha]$ to the type Nat of natural numbers isomorphic to $[1]$), invokes the syntactic approach to compute a $\mathit{put} : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat}$ function on shapes, and applies an adapted semantic approach to estimate an updated source list with the desired shape from the updated view data values and the original source list. Although the gain in updatability is quite significant for their examples – for instance, sieve that takes every second element of a list and $\mathit{reverse}$ that reverses a list become total lenses using their approach – it is not guaranteed in general that the resulting lenses are safe or total. The combined approach is proven to produce well-behaved (and no longer history ignorant) lenses, justifying the improved updatability.

Another mixed syntactic and semantic approach is used by Fegaras (2010) to translate view updates written in the XQuery Update facility view to embedded SQL updates, without view side-effects (denoting state-based put-Invertibility). Although no particular restrictions are placed in the XQuery syntax, he restricts the way relational tables are joined to form a view. The main idea of this approach is to use polymorphic type inference to statically infer trace information for the view expression and to identify which columns in the database table are *exclusive data sources* (EDS) that appear at most once in the view. Using such information, view updates reflectable to EDS sources and non-reflectable view updates can be translated at compile-time, while other updates must be checked for view side-effects at run-time. Unlike (Voigtländer, 2009), where view functions must be polymorphic, Fegaras only requires the updatable view components appearing in the view to be polymorphic, whereas non-updatable ones may be defined in any other way.

In Microsoft's Entity Framework, developers specify declarative mappings (using the general-purpose Entity SQL query language) between entities (objects) in an application model and tables in a relational database, and the system tries to infer a (reverse) lens that handles the data access layer and reflects updates to the client-side model as updates to the persistent data (Melnik et al., 2007). For each mapping that is a total and injective relation (Theorem 1), they infer a refinement that handles the reshaping of the data (containing a *query view* q that expresses entities in terms of tables and an *update view* u that expresses tables in terms of entities), and a merge function m, based on view complements, that captures the store-side state transition behavior (taking an old and a new database states and returns an updated state where the unexposed information in the old state is kept intact); and construct the lens given by $get = q$ and $put = m \circ (u \times id)$. Compilation guarantees that the resulting lens is total and well-behaved by construction for all client models but only for database states in the range of the mapping.

Diskin et al. (2011a) discuss the inherent limitations of state-based lens approaches and conceive an abstract framework of *delta lenses* that generalizes matching lenses (Barbosa et al., 2010) and separates lenses into two distinct phases: a differencing or alignment operation that computes a delta between two view states; and delta-based transformations that transform view deltas into source deltas, and vice-versa. A well-behaved delta lens satisfies lifted well-behavedness laws stating that put preserves identity updates and that a view-to-view round-trip preserves deltas, and a backward totality law stating that put processes any update to the original view. Their framework also gives a more refined account of history ignorance, by showing that a state-based instantiation of the PUTPUT law subsumes both a delta-based history ignorance law and a very well-behaved alignment property requiring that computing the alignment between two states is the same as composing the alignments between those states and an intermediate state. They also argue that, in practice, the component that makes history ignorance an overly restrictive premise is very well-behaved alignment, while preservation of update composition is manageable for delta-based applications. Note that since they only formulate the framework and do not propose a concrete language of delta lenses, none of the deployment axes is instantiated in Table 3.4.

An abstract framework of *change-based* lenses is proposed by Wang et al. (2011), in order to capitalize any locality of view updates into incrementality of update propagation. In their setting, view updates are seen as total endofunctions on views and view update propagation consists in identifying the location (and respective subterm) in the view where a modification occurs, applying a state-based put function to produce an updated source subterm, and replacing the old source subterm at the original location with the updated one. The authors discuss sufficient semantic conditions of the types and the transformations so that the transformed structures enjoy good locality properties, and focus particularly on polymorphic get functions specified as folds between parametrically polymorphic algebraic data types, with locations pointing to recursive subcomponents of source and view values. As long as their approach is not dependent on any particular bidirectional language, all the deployment axes besides the data domain are not instantiated in Table 3.4.

3.2.3 Maintainer Frameworks

In pioneering work, Meertens (1998) studies *(constraint) maintainers* in the context of graphical user interfaces. The idea is that a maintainer preserves a connection between two graphical objects, so that its unidirectional transformations propagate updates while restoring some desired constraint or relationship between them. For Meertens, a well-defined maintainer must be correct and total. He also postulates a *principle of least change* expressing that the action taken by a maintenance operation to restore the violation of a constraint should be minimal up to some sense. This implies that: if the constraint is not violated, then the maintainer shall introduce no change (hippocraticness); and harder to specify, changes shall be as small as possible when given many possible choices. To formalize this notion of distance between values, he treats domains as well-ordered sets that establish a preorder on values and builds maintainers that choose the closest values within such preorder for specific constraints over sets based on selectors. Moreover, Meertens develops a number of edit-based maintainers over algebraic structures such as lists by extending the algebra of types with additional constructors denoting edit operations. By considering explicit edit operations, these maintainers are able to identify precisely which parts of the models are not involved in a change and must be preserved. Meertens also explores the construction of maintainers by combinatorial means. In particular, he acknowledges that composition of maintainers is impossible in the general case, but identifies certain restrictions that make these constructions possible.

Hofmann et al. (2011) propose a different symmetric formulation and explore an algebraic category of *symmetric lenses* that generalize traditional asymmetric lenses and support composition unlike maintainers. Many of their results corroborate our own conclusions for asymmetric lenses (presented later in Chapter 4), such as the existence of tensor products and sums (but not categorical sums and products), and the ability to define recursive lenses with folds and unfolds that satisfy uniqueness. In comparison to a maintainer, a symmetric lens from S to T defines two transformations $putr : S \times C \to T \times C$ and $putl : T \times C \to S \times C$ that use as traceability a complement of type C to recover and store the information not present in S or T, instead of whole states. Well-behaved symmetric lenses are said to be total and stable in their own sense:

$$\pi_1 \circ putl \circ putr = \pi_1 \qquad \text{PUTRL}$$

$$\pi_1 \circ putr \circ putl = \pi_1 \qquad \text{PUTLR}$$

The added flexibility over maintainers comes from treating complements as internal components, not visible to users, that are constructed through composition. For instance, when composing two symmetric lenses, the internal complement of the composition lens (that is available as inputs to the transformations) is the product of the complements of the two lenses. A total stable maintainer with an arbitrary consistency relation can be converted into a total well-behaved symmetric lens with $putr\ (t', (s, t)) = (s \rhd t', (s \rhd t', t'))$ and a dual $putl$. Using this conversion, the composition of two maintainers (as symmetric lenses) is possible but not hippocratic. Another nuance is that due to the nature of the complements, standard syntactic equivalence of symmetric lenses is not possible. Instead, two symmetric lenses are said to be *observationally equivalent* if the behavior of their unidirectional transformations is indistinguishable, even though their complements may be structurally different. This entails that lens laws do not hold "on the nose" but only up to equivalence.

Hofmann et al. (2012) show how state-based symmetric lenses can be lifted to an operation-based space of *edit lenses* that handle updates in the form of the edits describing only the changes rather than whole annotated states. A well-behaved symmetric lens must be total (if an edit can be applied to a value, then propagation must be defined for such edit), stable and correct up to an explicit consistency relation considering the internal complement. Their language of edit lenses includes combinators for inductive products, sums, lists and two particular combinators over container structures, namely mapping (that only modifies the data) and restructuring (that only modifies the shape). Nevertheless, their restructuring combinator requires the positions of the transformed containers to be in bijective correspondence, meaning that it can not change the number of data placeholders of the shapes. Additionally, their language of edits over containers considers insertion and deletion at the rear positions of containers and rearrangement of the elements of a container without changing its shape. This entails that shape alignment is kept positional, as insertions and deletions at arbitrary positions are always reflected at the end positions of the shape.

In the follow-up of their previous delta-based framework, (Diskin et al., 2011b) conceive an abstract framework of *symmetric delta-based lenses*, where transformations consume and produce both a vertical delta representing the update and an horizontal delta representing the traceability information. They formulate a delta-based Weak Invertibility law stating that the result of a source-to-source round-trip is equal to the original source modulo another forward transformation (and vice-versa); and another Weak Undoability law stating that the propagation of a converse source update produces the converse target update modulo another backward transformation (and vice-versa). Because they only discuss the conceptual aspects of such a symmetric framework, the classification of their work in Table 3.4 does not include any deployment axes.

Most bidirectional transformation approaches in the context of model-driven development are based on graph grammar formalisms (Ehrig et al., 2006), especially triple graph grammars (Schürr, 1995; Schürr and Klar, 2008). A TGG consists of three source, correspondence and target graphs (conforming to three additional graph schemas if the TGG is typed), together with two graph morphisms from the correspondence graph to the source and to the target graphs. The composition of these two mappings defines a consistency relation between source and target elements, and is specified declaratively by a set of *graph rules* that match specific sub triple graphs and describe how any graph that conforms to the graph grammar can be created. Thus, the application of triple graph rules formalizes the simultaneous evolution of source and target graphs that are always consistent, using the third graph to track the correspondences between them. However, a TGG can only handle simultaneous updates, as the rules force graphs to evolve simultaneously, and is not able to restore inconsistent models, since the graphs must be consistent at all times. To derive an actual bidirectional transformation that processes independent updates and restores consistency, a set of *operational rules* can be generated automatically from a TGG according to some syntactic procedure. These rules can be seen as maintainer forward and backward transformations that use the correspondence graph, an instance of the consistency relation linking the source and target graphs, to provide important traceability information. The FUJABA graphical consistency management tool provides the most serious application support based on TGGs, and proposes to answer some of the design goals of the QVT proposal (Königs and Schürr, 2006).

As one of the first results regarding round-tripping behavior for TGGs, Ehrig et al. (2007) study in which particular circumstances a TGG realizes invertible forward and backward transformations.

Hermann et al. (2011) propose a carefully designed algorithm that derives a symmetric delta lens (Diskin et al., 2011b) from a TGG, but that does not even guarantee by construction that the derived transformations are deterministic, i.e., functions. Hence, they identify sufficient static conditions on the TGGs that can be automatically checked and ensure that the resulting transformations are indeed functions that are safe, correct and weakly invertible. In their context, update deltas are specified as graph morphisms denoting the elements that are preserved by each modification.

Cicchetti et al. (2011) propose JTL, a declarative bidirectional model transformation language inspired by a QVT-like syntax. A JTL transformation specifies a consistency relation between source and target schemas in the same way as a QVT relations program, that is compiled into a set of logic predicates. Then, they implement non-deterministic maintainers whose forward and backward transformations invoke a logic programming solver to find all possible models in the following sense: if a modified target model has a trace to a source model, such source model is modified, otherwise all consistent source models are returned. In their context, a bidirectional transformation must be safe and total, even if the consistency relation is non-total. Whenever no source model exists that is consistent with a modified target model, their transformations employ a model approximation technique to infer the closest non-consistent source model. As a result, transformations are not correct since they may generate inconsistent models.

An example of an ad-hoc symmetric approach is JT (Ennals and Gay, 2007), a bidirectional system for maintaining source code written in two specific programming languages, namely C and Jekyll, a superset of C with high-level features of functional languages. Since code translation is non-deterministic in general (the same behavior in C may be encoded in different flavors in Jekyll, and a Jekyll feature may be implemented in several ways in C), JT employs a notion of textual difference between each possible update and the original files in order to choose the closest match, allowing the detection of non-local edits such as reorderings. Although they don't provide any formal evidence, the authors clearly intend their system to be correct and stable modulo comments and whitespaces. For example, once a C file is translated to a Jekyll file, JT checks that the Jekyll file can be translated back to a C file with the original semantic meaning and, if possible, the same comments and layout. The translations are also assumed to be total – they advocate that all features in one language shall have reasonable translations into the other.

3.2.4 Synchronization Frameworks

Up until now, the studied bidirectional frameworks permit to synchronize different models but only describe how to propagate an update to a single model into an update to a previously consistent one. This contrasts with more general (partial) synchronization procedures that are concerned with bringing pairs of models into a consistent state by evolving both simultaneously, thus allowing parallel updates. Take for example the trivial consistency relation $id : S \to S$: since it is a bijection, a corresponding bidirectional transformation would just copy modifications from one model to the other, while a synchronizer would need to somehow merge two arbitrary models into a consistent duplicated state.

Nevertheless, bidirectional transformations may serve as intermediate constructs to build parallel synchronization systems. The first example of such an approach is the Harmony framework for the synchronization of heterogeneous data formats (Foster et al., 2005). In Harmony, developers design bidirectional lenses that transform source S and target T formats into a common abstract format A, and the framework employs a core three-way merger engine that takes two replicas and a common ancestor (all values of type A) to yields two updated replicas in which all non-conflicting changes have been merged. Using these components, they devise an heterogeneous synchronizer $sync : S \times T \to A \to S \times T$ that takes modified source and target models and a common ancestor to returns new synchronized source and target models, by using the forward transformations of the lenses to translate the modified artifacts into the abstract format and their backward transformations to translate abstract updates into updates to the original source and target models.

Another example is the SyncATL tool (Xiong et al., 2007), that proposes to synchronize models related by an ATL model transformation. The idea is to “lensify” the byte-code instructions of a stack-based language corresponding to an ATL low-level virtual machine, by extending each instruction with putting-back functions that resemble the bidirectional programs of Takeichi and are called when target values are replaced, deleted or inserted. They then define a synchronizer $sync : S \times T \to S \to S \times T$ that takes the modified models and the original source model (from which the original target model can be calculated) and returns synchronized models as follows: they compute source and target updates using a model differencing algorithm, merge the source update with the result of putting back the target update to compute the synchronized source value, and compute the synchronized target value by applying the transformation.

The authors define the properties of their synchronizer in terms of correctness (it returns a consistent state), stability (if the models are not modified, it produces no change) and an additional update preservation property meaning that synchronization shall preserve user modifications or else track inappropriate non-reflectable modifications.

Following a different approach, Xiong et al. (2009) propose the construction of a model synchronizer $sync : S \times T \to S \times T \to S \times T$ from a QVT-R bidirectional maintainer and a user-defined model difference procedure. To lift the bidirectional transformation into a parallel synchronizer, they apply the backward transformation to the modified target model and the original source model, use a three-way merger (that is calculated from the model difference procedure) to combine the modified source, the original source and the result into a synchronized source and apply the forward transformation to propagate the source modifications to the updated target. The authors show that synchronization is correct if the maintainer is so, and stable if the maintainer is hippocratic. However, their implementation does not check automatically if the QVT transformation obeys these postulates. The third preservation property is ensured by an additional check at the end of synchronization; if user modifications are deleted, then it reports a conflict to the user.

3.3 Summary

In this chapter, we have presented a taxonomy for the classification of the defining features of BX frameworks. In table Table 3.4, we have taken a global picture of the state of the art of the field at the light of our taxonomy.

Looking at Table 3.4, we can see that most frameworks are essentially state-based. Some operation-based approaches exist, but many still consider only state-based laws. It is also noteworthy that most frameworks only guarantee (state-based) stability, correctness and invertibility properties. Unfortunately, these properties still leave a lot of room in the behavior of the transformations, and make it harder to compare the effectiveness of two bidirectional frameworks solely based on their stated properties.

Another relevant aspect is that most of our surveyed approaches are based on the abstract frameworks of mappings and lenses. In more recent years, a few symmetric frameworks inspired by (slightly more general formulations of) maintainers have been proposed. In this context, considerable effort has been put in trying to formalize model-driven transformation scenarios with typically language-based bidirectional transformation techniques. In fact, we can identify two more or less clear (and not exclusive) subcommunities within the field of bidirectional transformations: a *programming languages* subcommunity, concerned with the semantic properties and linguistic mechanisms required to specify bidirectional transformations; and a *model transformations* subcommunity, more focused on software-model synchronization, tool integration and real-world application scenarios. However, the significantly distinct motivations and techniques (language-based solutions usually go by the framework of lenses, while model-based solutions are more inspired by QVT specifications or graph formalisms like TGGs) still make it very challenging to compare work originating from different subcommunities.

More related to this thesis, we can identify other open issues that are not directly inferable from Table 3.4. First, despite the huge majority of bidirectional approaches are combinatorial and therefore suffer from inherent efficiency limitations, the problem of optimizing bidirectional programs remains largely unaddressed. Second, although state-based languages are already quite mature, they are often poorly configurable as long as they do not allow users to control the alignment mechanisms that affect the behavior of the bidirectional transformations. Existing delta-based frameworks bring up this issue and discuss it in abstract way, but stop short of proposing a concrete solution. Third, the specification style of lens-based bidirectional languages is usually not very flexible, in the sense that it requires users to describe exactly how source models are mapped to target models. In order to make these languages applicable to very large models, some generic features are necessary to reduce the specification cost and foster the reusability. These problems will be studied and addressed in the succeeding chapters.

Feature Approach	Scheme			Properties							Deployment					
	Framework	Source Updates	Target Updates	Stability	Invertibility	Undoable	History Ignorance	Correctness	Hippocraticness	Totality	Source Domain	Target Domain	Typing	Specification	Language	Bidirectionalization
Brabrand et al. (2008)	⇌	S			⇢⇠			⇄	⇄		S		T	R	D	S
Kawanaka and Hosoya (2006)	⇌	S						⇄		⇄	S		T	R	D	S
Yokoyama et al. (2008)	⇌	S			⇄					⇄	T		T	T	D	C
Kennedy (2004)	⇌	S			→⇠					⇢⇠	T	S	𝕋	T	D	C
Wadler (1987)	⇌	S			⇄					⇢⇠	T		𝕋	T	G	A
Atanassow and Jeuring (2007)	⇌	S			⇄					⇄	T		𝕋	T	G	S
Terwilliger et al. (2007)	⇌	S	E		⟶					→⇠	T	R	𝕋	T	D	C
Cunha et al. (2012)	⇌	E		⇄	⟶		⇄	⇄	⟵	⇄	R		𝕋			A
Mu et al. (2004)	⇌	S	E		→⇠					⇠	T		𝕋	T	D	C
Berdaguer et al. (2007)	⇌	S			⟶					→⇠	T		𝕋	T	D	C
Wang et al. (2010)	⇌	S			⟵					⇄	T		𝕋	T	D	C
Foster et al. (2007)	⊵	S		⟵	⟵					⇄	T		T	T	D	C
Bohannon et al. (2006)	⊵	S		⟵	⟵					⇄	R		T	T	D	C
Bohannon et al. (2008)	⊵	S		⇠	⟵					⇄	S		𝕋	T	D	C
Foster et al. (2008)	⊵	S		⇠	⇠					⇄	S		𝕋	T	D	C
Barbosa et al. (2010)	⊵	S	D	⟵	⟵					⇄	S		𝕋	T	D	C
Hu et al. (2008)	⊵	S	E	⇢⇠							T		𝕋	T	D	C
Liu et al. (2007)	⊵	S	E	⟵	⟵					⟶	T		𝕋	T	G	C
Hidaka et al. (2010)	⊵	S	E	⟵	⇠					⟶	G		𝕋	T	G	C𝕊
Takeichi (2009)	⊵	S		⟵						⟶	T		𝕋	T	G	C
Matsuda et al. (2007)	⊵	S		⟵	⟵	⟵	⟵			⟶	T		𝕋	T	G	S
Voigtländer (2009)	⊵	S		⟵	⟵	⟵	⟵				T		𝕋	T	G	𝕊
Voigtländer. et al. (2010)	⊵	S		⟵	⟵					⟶	T		𝕋	T	G	S𝕊
Fegaras (2010)	⊵	S	E	⟵	⇠					⟶	R	T	𝕋	T	G	S𝕊
Melnik et al. (2007)	⊵	S		⟵	⟵					⇢←	G	R	𝕋	R	G	S
Diskin et al. (2011a)	⊵	D		⟵	⟵		⟵			→⇠						
Wang et al. (2011)	⊵	S	F	⟵	⟵		⟵			→⇠	T					
Chapter 4	⊵	S		⟵	⟵					⇄	T		𝕋	T	D	C
Chapter 5	⊵	D		⟵	⟵					⇄	T		𝕋	T	D	C
Meertens (1998)	≷	S/E						⇄	⇄	⇄	T		T	T	D	C
Hofmann et al. (2011)	≷	S		⇄						⇄	T		𝕋	T	D	C
Hofmann et al. (2012)	≷	E		⇄				⇄		⇄	T		𝕋	T	D	C
Diskin et al. (2011b)	≷	D		⇄	⇢⇠	⇢⇠		⇄								
Ehrig et al. (2007)	≷	S			⇄			⇄		⇄	G		T	R	D	S
Hermann et al. (2011)	≷	D			⇢⇠			⇄		⇢⇠	G		T	R	D	S
Cicchetti et al. (2011)	≷	S		⇄						⇄	G		𝕋	R	D	S𝕊
Ennals and Gay (2007)	≷	S			⇢⇠			⇄		⇄	T		𝕋			A

Table 3.4: Comparison of existing bidirectional transformation approaches.

Chapter 4

Generic Point-free Lenses

This chapter shows that most of the standard point-free combinators can be lifted to a language of total well-behaved lenses, allowing us to use the point-free style to define powerful bidirectional transformations by composition. Since general recursion is usually deterred in the point-free style of programming in favor of more calculation-friendly recursion patterns, we define generic lenses over arbitrary inductive data types by lifting standard constructs like folds and unfolds. These constructs allow to define lenses over rich data types like lists and trees in terms of their structure and using only a small set of generic combinators. This often-called *datatype-generic programming* style (Gibbons, 2007) leads to shorter and clearer specifications in comparison to datatype-specific programs defined directly by induction on specific base types.

The point-free style is also characterized by a rich set of algebraic laws, opening interesting perspectives towards a lens calculus to reason about bidirectional transformations. The second goal of this chapter is to develop such a calculus, by showing that many of the laws characterizing point-free combinators and recursion patterns are also valid at the lens level. A key result of our work is that uniqueness also holds for bidirectional folds and unfolds, thus unleashing the power of fusion as a bidirectional program optimization technique.

This algebraic theory mitigates the inefficiency of combinatorial bidirectional languages and draws practical applications for the optimization of lens programs, by ensuring that optimization at the lens level preserves the bidirectional semantics and subsumes the independent optimization of all unidirectional transformations. This contrasts with traditional bidirectional approaches, that either do not support optimization or do not consider the implications of optimizing the forward transformation on the behavior of the bidirectional transformation.

4.1 Point-free Combinators as Lenses

In this chapter, we will develop the building blocks of our point-free language of total well-behaved lenses over algebraic data types. Instantiating the design space from Chapter 3, our formulation of lenses can be defined as follows:

Definition 4 (Lens)**.** *A well-behaved lens l, denoted by $l : C \trianglerighteq A$, is a bidirectional transformation that comprises three total functions $get : C \to A$, $put : A \times C \to C$ and $create : A \to C$, satisfying the following properties:*

$$get \circ create = id \qquad \textsc{CreateGet}$$

$$get \circ put = \pi_1 \qquad \textsc{PutGet}$$

$$put \circ (get \vartriangle id) = id \qquad \textsc{GetPut}$$

Definition 5 (Lens equality)**.** *Two lenses f and g are equal, written $f = g$, iff their transformations are equal, i.e., $get_f = get_g$, $put_f = put_g$ and $create_f = create_g$.*

Recapitulating, property CREATEGET guarantees that get is an abstraction function, i.e., A contains at most as much information as C. PUTGET guarantees that the lens is *acceptable* (put-Invertibility in Chapter 3), i.e., updates to a view cannot be ignored and are to be translated exactly. Finally, GETPUT states that the lens is *stable* (put-Stability in Chapter 3), i.e, if the view does not change, then neither does the source.

By favoring totality and a simple data domain, not all point-free combinators will be valid lenses in our setting. Therefore, our first research question is: which point-free combinators in SET also denote lenses? In order to answer this question, we will scrutinize each point-free combinator from Chapter 2 and, whenever possible, lift it to a lens combinator. The point-free style being so intertwined with algebraic calculation, the followup question must necessarily be: are the laws characterizing the standard point-free combinators also valid for the lifted lenses? To answer this second question, we will state for each bidirectionalized combinator the laws that characterize it.

To avoid introducing new notation, we will denote the lens corresponding to a particular combinator using the same syntax. It shall be clear from the context if we are referring to the lens combinators and laws or to the standard point-free combinators and laws. For some lenses there is some freedom in the design of the backward transformations (namely, *create* and *put*). As such, they will receive extra parameters to plugin in contexts where such freedom exists.

4.1.1 Basic Lens Combinators

Two of the most fundamental lens combinators are identity and composition, first defined in (Foster et al., 2007). Both can be restated in point-free as follows:

$$\begin{array}{ll}
id : C \trianglerighteq A & \forall f : B \trianglerighteq A, g : C \trianglerighteq B.\ (f \circ g) : C \trianglerighteq A \\
get \quad = id & get \quad = get_f \circ get_g \\
put \quad = \pi_1 & put \quad = put_g \circ (put_f \circ (id \times get_g) \vartriangle \pi_2) \\
create = id & create = create_g \circ create_f
\end{array}$$

Identity simply copies the input value for *get* and *create*, and ignores the original source value for *put*. For composition, if the concrete domain of f and the abstract domain of g have the same type, then f and g are composable and $f \circ g$ is a lens with the concrete domain of g and the abstract domain of f. In the *get* and *create* directions, the composed transformation is just the composition of the respective transformations from f and g. In the *put* direction, in order to apply the *put* functions in sequence, the original concrete value is duplicated. Note that, while put_g consumes the original concrete value with type C, the intermediate concrete value with type B passed to put_f is calculated by applying get_g to the original concrete value.

The identity and associativity axioms that characterize these combinators are also valid for the lens versions:

$$id \circ f = f = f \circ id \qquad id\text{-NAT}$$

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \circ\text{-ASSOC}$$

Since both these laws are valid, we can define a category LENS, whose objects are the same objects of SET and arrows are well-behaved lenses.

Proof. The first equality is always trivially true because the *get* function has exactly the same point-free definition as the lens itself. The trickiest part is always proving that both *put*s are equal (especially when involving lots of compositions). For example, for ∘-ASSOC such proof can done as follows:

$$
\begin{array}{l}
put_{f \circ (g \circ h)} \\
\ = \{\text{definition of } put\} \\
put_{g \circ h} \circ (put_f \circ (id \times get_{g \circ h}) \vartriangle \pi_2) \\
\ = \{\text{definition of } put\} \\
put_h \circ (put_g \circ (id \times get_h) \vartriangle \pi_2) \circ (put_f \circ (id \times get_{g \circ h}) \vartriangle \pi_2) \\
\ = \{\times\text{-ABSOR}; \times\text{-ABSOR}; \times - \text{CANCEL}\} \\
put_h \circ (put_g \circ (put_f \circ (id \times get_{g \circ h}) \vartriangle get_h \circ \pi_2) \vartriangle \pi_2) \\
\ = \{\text{definition of } get\} \\
put_h \circ (put_g \circ (put_f \circ (id \times get_g \circ get_h) \vartriangle get_h \circ \pi_2) \vartriangle \pi_2) \\
\ = \{\times - \text{FUSION}; \times - \text{FUNCTOR-COMP}; \times - \text{CANCEL}\} \\
put_h \circ (put_g \circ (put_f \circ (id \times get_g) \vartriangle \pi_2) \circ (id \times get_h) \vartriangle \pi_2) \\
\ = \{\text{definition of } put\} \\
put_h \circ (put_{f \circ g} \circ (id \times get_h) \vartriangle \pi_2) \\
\ = \{\text{definition of } put\} \\
put_{(f \circ g) \circ h}
\end{array}
$$

The proof for $create$ is trivial and will be elided. □

Note that, in the above proof, the equalities are proven using laws valid for point-free functions on SET (see Chapter 2 for a compendium). Throughout the chapter, we will use the same name to denote laws valid both on LENS and SET. Disambiguation should be trivial from the context. The well-behavedness (and equational) proofs for composition and other crucial combinators introduced in this section can be found in Appendix B. The remaining proofs, although a bit more complex than the one above, are still fairly easy, at least for someone experienced with the point-free style.

The *bang* combinator is the primitive lens for defining abstractions and ignores all the source information:

$$
\begin{array}{l}
\forall f : 1 \to C.\ !^f : C \trianglerighteq 1 \\
get \quad\ = \,! \\
put \quad\ = \pi_2 \\
create = f
\end{array}
$$

Since get drops the source value, in put we simply need to restore the original source. Here, the superscript is a function $f : 1 \to C$ that generates default concrete values to be used by $create$. Due to this parameter function, we cannot state that 1 is a proper terminal object of LENS, because there is more than one lens with type $C \trianglerighteq 1$. Nonetheless, we can phrase a lifted version of the uniqueness law for bang:

$$f = !^{create_f} \Leftrightarrow f : C \trianglerighteq 1 \qquad \text{!-UNIQ}$$

4.1.2 Products

Like the bang combinator, the projections π_1 and π_2 are two main ingredients for defining more complex lenses that project away components of a concrete data type:

$$\begin{array}{ll}
\forall f : A \to B.\, {\pi_1}^f : A \times B \trianglerighteq A & \forall f : B \to A.\, {\pi_2}^f : A \times B \trianglerighteq B \\
get \quad = \pi_1 & get \quad = \pi_2 \\
put \quad = id \times \pi_2 & put \quad = swap \circ (id \times \pi_1) \\
create = id \vartriangle f & create = f \vartriangle id
\end{array}$$

Since π_1 and π_2 project the corresponding elements of the source pair, the backward transformations have to recover the projected out elements to construct a new pair. While $create$ simply generates a new value (using the parameter function) for the "lost" value of the pair, put copies it from the original source pair.

In general, the split of two lenses $f : C \trianglerighteq A$ and $g : C \trianglerighteq B$ sharing the same domain is not a well-behaved lens $f \vartriangle g : C \trianglerighteq A \times B$. For example, the duplication combinator $id \vartriangle id : A \trianglerighteq A \times A$ would be a valid lens iff the invariant $\pi_1 = \pi_2$ was imposed on the codomain $A \times A$, stating that both components of the pair are always equal, i.e., there are no lenses of type $A \trianglerighteq A \times A$ (unless A is the unit type), because the target is not an abstraction of the source. Nevertheless, if f and g project distinct concrete information from C, such that $get_{f \vartriangle g}$ is surjective (what can be proven by showing that $id \subseteq [get_g \circ {get_f}^{\circ}]$), then from Theorem 2 there exists a well-behaved lens $f \vartriangle g$. Additionally, we would like the transformations of the split lens to be constructively defined using the transformations of its argument lenses. A tempting (incomplete) definition for the split lens is the following:

$$\begin{array}{l}
\forall f : A \trianglerighteq B, g : A \trianglerighteq C, h : B \times C \to A. \\
\quad get_f \circ put_g = get_f \circ \pi_1 \;\wedge\; get_g \circ put_f = get_g \circ \pi_2 \Rightarrow (f \vartriangle g) : A \trianglerighteq B \times C \\
get \quad\;\; = get_f \vartriangle get_g \\
put \quad\;\; = put_f \circ (id \times put_g) \circ assocr \\
create \subseteq (f \vartriangle g)^{\circ}
\end{array}$$

For our lens to be well-behaved, we must impose some additional conditions in the style of PUTCPL stating that get_f is a complement of the lens g and that get_g is a complement of the lens f. This ensures that put_f and put_g preserve the distinct information in the other side of the view pair, and thus that we can propagate updates to the view to the concrete model by independent inspection of both components of the pair, for example by defining put as the expression $put_f \circ (id \times put_g) \circ assocr$. Unfortunately, it is hard to provide a reasonable general definition for $create$ at the cost of $create_f$ and $create_g$, since it would require the relational intersection of the possible results produced by $create_f$ and $create_g$. This prevents us from giving a generic definition of split as a well-behaved lens, and $create$ can be any total function satisfying the CREATEGET law.

For $swap$, it is rather easy to show that the complement conditions are valid, that the suggested definition for put is equal to the expected $swap \circ \pi_1$ (according to the further presented generic definition of an isomorphism as a well-behaved lens), and that $create$ can be done using $swap$ itself.

Another instance of split that satisfies the complement conditions is the product combinator $f \times g$, defined as follows:

$$\begin{aligned}
&\forall f : C \trianglerighteq A, g : D \trianglerighteq B.\, f \times g : C \times D \trianglerighteq A \times B\\
&get \quad\;\; = get_f \times get_g\\
&put \quad\;\; = (put_f \times put_g) \circ distp\\
&create = create_f \times create_g
\end{aligned}$$

Again, it is easy to show that the put of the split is equivalent to $(put_f \times put_g) \circ distp$. For this particular split, creating a concrete pair from an abstract one can be done by independently creating both components of the pair. In practice, most expressions involving split whose argument lenses are complements of each other can be transformed into point-free expressions using other valid lens combinators on products (like $\times$ or $swap$).

Due to the non-existence of splits and thus categorical products, our category of lenses is not as "well-behaved" as SET. However, our product and projections still satisfy some interesting laws. The following laws guarantee that the product lens is also a bifunctor in the category of lenses:

$$id \times id = id \qquad \times\text{-FUNCTOR-ID}$$

$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-FUNCTOR-COMP}$$

Projections also enjoy a kind of naturality law, with a precise characterization of how the default generation function must be adapted.

$$\pi_1{}^f \circ (f \times g) = f \circ \pi_1{}^{create_g \circ f \circ get_f} \qquad \pi_1\text{-NAT}$$

$$\pi_2{}^f \circ (f \times g) = g \circ \pi_2{}^{create_f \circ f \circ get_g} \qquad \pi_2\text{-NAT}$$

4.1.3 Sums

Moving to sums, the either (or junc) combinator can be lifted into the following lens:

$$\begin{aligned}
&\forall p : A \to 2, f : C \trianglerighteq A, g : B \trianglerighteq A.\ (f \triangledown g)^p : C + B \trianglerighteq A \\
&get \quad\;\; = get_f \triangledown get_g \\
&put \quad\;\; = (put_f + put_g) \circ distr \\
&create = (create_f + create_g) \circ p?
\end{aligned}$$

When putting back, put_f is used if the concrete value is a left alternative and put_g otherwise. For $create$ we have two alternatives – either apply $create_f$ or $create_g$ – depending on the result of applying the predicate to the view value. We can also define left-biased (${}_{\bullet}\!\triangledown$) and right-biased ($\triangledown_{\bullet}$) versions of this lens, for the cases when the predicate always returns true or false, respectively. Assuming that predicates are represented using sums (as fed to $p?$), this lens corresponds to a point-free formulation of the concrete conditional combinator $ccond$ from (Foster et al., 2007).

The sum injections $i_1 : A \to A + B$ and $i_2 : B \to A + B$ are non-surjective functions that are classic examples of refinements (Cunha et al., 2006a). Similarly to split, the only way to lift them into lenses would be by imposing an invariant on the codomain $A + B$, constraining its values to be all left or all right alternatives, respectively. Since this semantic constraint is not supported by standard type systems, unrestricted usage of the injections will be disallowed in our lens language. Notwithstanding, if injections are used inside an expression that is jointly surjective as in (Wang et al., 2010), they can sometimes build up well-behaved lenses. Two particular useful cases are the lenses $i_1 \triangledown f$ and $f \triangledown i_2$. These eithers are necessarily surjective (because f, being a well-behaved lens, is already surjective) and give rise to the following lens combinators:

$$\forall f : C \trianglerighteq A + B.\ i_1 \triangledown f : A + C \trianglerighteq A + B$$

$$i_1 \triangledown f = (id \,{}_{\bullet}\!\triangledown\, id + id) \circ coassocl \circ (id + f) \qquad i_1\text{-EITHER-DEF}$$

$$\forall f : C \trianglerighteq A + B.\, f \triangledown i_2 : C + B \trianglerighteq A + B$$
$$f \triangledown i_2 = (id + id \triangledown_{\bullet} id) \circ coassocr \circ (f + id) \qquad i_1\text{-EITHER-DEF}$$

In practice, they will allow us to write non-trivial (but still jointly surjective) constructor lenses like the following ones for naturals and lists, that will be useful for some of the examples presented later:

$$\begin{array}{ll} zero \triangledown id = in_{\mathsf{Nat}} \circ (i_1 \triangledown out_{\mathsf{Nat}}) & id \triangledown succ = (in_{\mathsf{Nat}} \circ (out_{\mathsf{Nat}} \triangledown i_2)) \\ nil \triangledown id \;\; = in_{\mathsf{List}_A} \circ (i_1 \triangledown out_{\mathsf{List}_A}) & id \triangledown cons = (in_{\mathsf{List}_A} \circ (out_{\mathsf{List}_A} \triangledown i_2)) \end{array}$$

The sum combinator can have the following lifting into a lens:

$$\begin{array}{l} \forall h : A \times D \to C, i : B \times C \to D, f : C \trianglerighteq A, g : D \trianglerighteq B. \\ \;\; get_f \circ h = \pi_1 \;\wedge\; get_g \circ i = \pi_1 \Rightarrow (f + g)^{h,i} : C + D \trianglerighteq A + B \\ get \;\;\;\; = get_f + get_g \\ put \;\;\;\; = (put_f \triangledown h + i \triangledown put_g) \circ dists \\ create = create_f + create_g \end{array}$$

In the definition of put, $dists : (A + B) \times (C + D) \to (A \times C + A \times D) + (B \times C + B \times D)$ is first used to span the four possible cases. If the abstract and concrete values match (cases $A \times C$ and $B \times D$), then we apply put_f and put_g as expected. Otherwise (cases $A \times D$ and $B \times C$), we need some mechanism to bring the "out of sync" abstract and concrete values into a new concrete value that is consistent with the abstract value: we use the parameter functions $h : A \times D \to C$ and $i : B \times C \to D$ to reconstruct such values from the available information. The conditions $get_f \circ h = \pi_1$ and $get_g \circ i = \pi_2$ force these functions to be acceptable (likewise put), i.e., the view cannot be ignored when computing the defaults. Useful candidates for h and i are $create_f \circ \pi_1$ and $create_g \circ \pi_1$; in fact, when superscripts are omitted from the sum these are assumed to be the parameters. The definition of $create$ is merely the sum of the $create$ functions of f and g. This sum combinator is essentially the point-free analogue of the abstract conditional combinator $acond$ from (Foster et al., 2007).

Likewise its functional counterpart, the either lens combinator also satisfies fusion and absorption laws in LENS:

$$f \circ (g \triangledown h)^p = (f \circ g \triangledown f \circ h)^{p \circ create_f} \qquad +\text{-FUSION}$$
$$(f \triangledown g)^p \circ (h + i)^{j,k} = (f \circ h \triangledown g \circ h)^p \qquad +\text{-ABSOR}$$

Note how the first law constrains the new predicate to be coherent with the $create$ of the fused lens. Compositions of sums can be fused according to the following law, that states how the new parameter functions can be deduced:

$$\begin{gathered} (f+g)^{j,k} \circ (h+i)^{l,m} = (f \circ h + g \circ i)^{n,o} \\ \Leftrightarrow \qquad\qquad +\text{-}\textsc{Comp} \\ n = l \circ (j \circ (id \times get_i) \vartriangle \pi_2) \quad \wedge \quad o = m \circ (k \circ (id \times get_h) \vartriangle \pi_2) \end{gathered}$$

If the parameters are the standard $create \circ \pi_1$, we have the following simplified version:

$$(f+g) \circ (h+i) = f \circ h + g \circ i \qquad +\text{-}\textsc{Functor-Comp}$$

This and the following law ensure that the sum combinator is also a bifunctor in LENS:

$$(id + id)^{f,g} = id \qquad +\text{-}\textsc{Functor-Id}$$

4.1.4 Isomorphisms as Lens Combinators

The simplest cases of bidirectional transformations are isomorphisms. Given a bijective function $f : A \to B$, we can trivially define a *lens isomorphism* $f : A \trianglerighteq B$ with:

$$\begin{aligned} get &= f \\ put &= f^{-1} \circ \pi_1 \\ create &= f^{-1} \end{aligned}$$

It is trivial to prove that $f \circ f^{-1} = f^{-1} \circ f = id$ is also valid at the lens level. There are many useful examples of such lens isomorphisms, such as $swap$, $assocl$, $coswap$, $coassocl$ and $distl$. Since splits and injections are not valid lenses, these lens isomorphisms play an important role in extending the expressivity of our point-free lens language. For example, all lenses that rearrange nested pairs can be defined as compositions of $swap$, $assocl$, $assocr$ and products (Mu et al., 2004).

We name a lens η describing a bidirectional natural transformation between functors F and G a *natural lens* and type it with the signature $\eta : \mathsf{F} \mathrel{\dot{\trianglerighteq}} \mathsf{G}$, where $get : \mathsf{F} \mathrel{\dot{\to}} \mathsf{G}$, $put : \mathsf{G} \otimes \mathsf{F} \mathrel{\dot{\to}} \mathsf{F}$ and $create : \mathsf{G} \mathrel{\dot{\to}} \mathsf{F}$ are natural transformations. It assigns to each type A an arrow $\eta_A : \mathsf{F}\ A \trianglerighteq \mathsf{G}\ A$ such that, for any lens $f : A \trianglerighteq B$, the following naturality condition holds:

$$\eta \circ \mathsf{F}\, f = \mathsf{G}\, f \circ \eta \qquad \eta\text{-NAT}$$

This concept can be generalized to functors of higher arity. If a natural lens η is also an isomorphism, then it is called a *natural lens isomorphism*. Such is the case of $swap$ and all similar lenses. For example, the following bidirectional naturality and isomorphism laws are also valid in the LENS category:

$$\begin{aligned}
&swap \circ (f \times g) = (g \times f) \circ swap && swap\text{-NAT}\\
&swap \circ swap = id && swap\text{-ISO}\\
&coswap \circ (f + g)^{f,g} = (g + f)^{g,f} \circ coswap && coswap\text{-NAT}\\
&coswap \circ coswap = id && coswap\text{-ISO}
\end{aligned}$$

The naturality law for $distl$ is a bit more tricky because we need to adapt the default functions passed to the put of the sum:

$$\begin{gathered}
distl \circ ((f + g)^{j,k} \times h) = (f \times h + g \times h)^{l,m} \circ distl\\
\Leftrightarrow \qquad distl\text{-NAT}\\
l = (j \times put_h) \circ distp \quad \wedge \quad m = (k \times put_h) \circ distp
\end{gathered}$$

To understand the bottom equation of this law, consider that $f : A \rightarrow A'$, $g : B \rightarrow B'$ and $h : C \rightarrow C'$. For the left side of the top equation, the backward semantics of the sum $f + g$ considers only the view $A' + B'$, and C' values are processed independently using put_h. For the right side of the top equation, that first distributes the source sum over the product, the backward semantics of the sum considers the view $(A' \times C') + (B \times C')$, what may lead to a different "out of sync" behavior if the choices of the parameter functions are unconstrained. For an example, if we consider that the parameter functions are the default $create \circ \pi_1$, the left case would apply $create_f$ or $create_g$ to "out of sync" values, while the right case would instead apply $create_{f \times h}$ or $create_{g \times h}$, independently of the fact that the original source of type C required to restore C' values is always available. Our side-condition ensures that the parameter functions for the right equation apply put_h instead.

Many other useful laws can be proved about these lens isomorphisms, such as the following cancelation laws:

$$\begin{aligned}
&{\pi_1}^f \circ \mathit{swap} = {\pi_2}^f \quad \wedge \quad {\pi_2}^f \circ \mathit{swap} = {\pi_1}^f && \mathit{swap}\text{-}\textsc{Cancel}\\
&(f \triangledown g)^p \circ \mathit{coswap} = (g \triangledown f)^{\mathit{coswap} \circ p} && \mathit{coswap}\text{-}\textsc{Cancel}\\
&({\pi_1}^f + {\pi_1}^g)^{\mathit{put}_{\pi_1}, \mathit{put}_{\pi_1}} \circ \mathit{distl} = {\pi_1}^{f \,\triangledown\, g} && \mathit{distl}\text{-}\pi_1\text{-}\textsc{Cancel}\\
&({\pi_2}^f \triangledown {\pi_2}^g)^p \circ \mathit{distl} = {\pi_2}^{(f+g) \circ p?} && \mathit{distl}\text{-}\pi_2\text{-}\textsc{Cancel}\\
&(\mathit{id} \triangledown \mathit{id})^{p \circ \pi_1} \circ \mathit{distl} = (\mathit{id} \triangledown \mathit{id})^p \times \mathit{id} && \mathit{distl}\text{-}\mathit{id}\text{-}\textsc{Cancel}
\end{aligned}$$

4.1.5 Higher-order Lens Combinators

Higher-order lenses are also definable in our category of lenses through exponentiation. The exponentiation combinator • can be lifted into a lens as follows:

$$\begin{aligned}
&\forall f : B \trianglerighteq C.\, f^{\bullet} : B^A \trianglerighteq C^A\\
&\mathit{get} = {\mathit{get}_f}^{\bullet}\\
&\mathit{put} = {\mathit{put}_f}^{\bullet} \circ \hat{\triangle}\\
&\mathit{create} = {\mathit{create}_f}^{\bullet}
\end{aligned}$$

Here, $\hat{\triangle} : C^A \times B^A \rightarrow (C \times B)^A$ denotes the uncurried version of the split combinator (Cunha and Pinto, 2005), and is defined as the following point-free expression:

$$\hat{\triangle} = \overline{(\mathit{ap} \times \mathit{ap}) \circ ((\pi_1 \times \mathit{id}) \triangle (\pi_2 \times \mathit{id}))} \qquad \hat{\triangle}\text{-}\textsc{Def}$$

Again, exponentiation is a functor in the LENS category:

$$\begin{aligned}
&\mathit{id}^{\bullet} = \mathit{id} && \textsc{Exp-Functor-Id}\\
&(f \circ g)^{\bullet} = f^{\bullet} \circ g^{\bullet} && \textsc{Exp-Functor-Comp}
\end{aligned}$$

The ap combinator can also be lifted to a lens, as follows:

$$\begin{aligned}
&\forall f : B \rightarrow A.\, \mathit{ap}^f : B^A \times A \trianglerighteq B\\
&\mathit{get} = \mathit{ap}\\
&\mathit{put} = \overline{(\pi_1 \circ \pi_1 \triangledown \mathit{ap} \circ (\pi_1 \circ \pi_2 \times \mathit{id})) \circ [\pi_2 \circ \pi_2]?} \triangle \pi_2 \circ \pi_2\\
&\mathit{create} = \overline{\pi_1} \triangle f
\end{aligned}$$

Since the point-free definition is a bit tricky, we also present the point-wise versions of the backward transformations for better comprehension:

$$\begin{aligned} put\ (y, (g, x)) &= (\lambda z \rightarrow \mathbf{if}\ x \equiv z\ \mathbf{then}\ y\ \mathbf{else}\ g\ z, x) \\ create\ y &= (\lambda z \rightarrow y, f\ y) \end{aligned}$$

Note how the put function updates the original function with a new result for the input value that was applied. The parameter function f is used in $create$ to choose a value of the domain A. Any such value can be chosen, since the constant function returned by $\overline{\pi_1} : B \rightarrow B^A$ will always restore the view value passed to $create$.

Application cancels exponentiation, according to the following law:

$$ap^g \circ (f^\bullet \times id) = g \circ ap^{g \circ get_f} \qquad \text{EXP-CANCEL}$$

Unfortunately, we also do not have categorical exponentiation in LENS because the curry of a well-behaved lens may not be a well-behaved lens. For example, note that, although $\pi_2 : A \times B \unrhd B$ is a lens, $\overline{\pi_2} : A \rightarrow B^B$ is not surjective and thus cannot be made into a lens (given a value of type A it returns the function id).

4.2 Recursion Patterns as Lenses

In this section, we investigate the specification of recursive lenses over inductive data types. As we have seen in Chapter 2, the sum-of-products structure of such types can be represented generically as a fixed point of a regular functor. Instead of defining lenses "by-hand" using general recursion as in (Foster et al., 2007), we resort to well-known recursion patterns and use their rich algebraic laws to prove that the resulting lenses are well-behaved. These patterns are examples of datatype-generic programs (Gibbons, 2007) that are parameterized by the base functor of any regular data type, and can be instantiated on many base functors to obtain the respective datatype-specific lenses. Also, the lens arguments passed to our generic recursion patterns can be defined by reusing the non-recursive point-free lens combinators from the previous section.

4.2.1 Functor Mapping

Building on our lenses over products and sums, we can define a polytypic functor mapping lens combinator simply by induction on the structure of polynomial functors:

$$\begin{array}{ll} \forall f : C \trianglerighteq A.\, \mathsf{F}\ f : \mathsf{F}\ C \trianglerighteq \mathsf{F}\ A & \\ \mathsf{Id}\ f \qquad\quad = f & \\ \underline{C}\ f \qquad\quad = id & \\ (\mathsf{F} \otimes \mathsf{G})\ f = \mathsf{F}\ f \times \mathsf{G}\ f & \qquad\qquad \textsc{Functor-Def} \\ (\mathsf{F} \oplus \mathsf{G})\ f = \mathsf{F}\ f + \mathsf{G}\ f & \\ (\mathsf{F} \odot \mathsf{G})\ f = \mathsf{F}\ (\mathsf{G}\ f) & \end{array}$$

Given two lenses $f : A \trianglerighteq C$ and $g : B \trianglerighteq D$, the bifunctor mapping lens $\mathsf{B}\ f\ \ g : \mathsf{B}\ A\ \ B \trianglerighteq \mathsf{B}\ C\ \ D$ can be defined in a similar way.

Using the laws from the previous section, we can polytypically prove, in the style of Hinze (2000), that this definition trivially satisfies the functor laws:

$$\mathsf{F}\ id = id \qquad\qquad \textsc{Functor-Id}$$

$$\mathsf{F}\ f \circ \mathsf{F}\ g = \mathsf{F}\ (f \circ g) \qquad\qquad \textsc{Functor-Comp}$$

In the put direction, the functor mapping lens will first align the abstract and concrete instances of the same F-structure, and then apply put_f to each instance. We can decompose this first alignment step as a polytypic functor zipping function $fzip_{\mathsf{F}}$, defined as follows for polynomial functors:

$$\begin{array}{ll} fzip_{\mathsf{F}} : (A \to C) \to \mathsf{F}\ A \times \mathsf{F}\ C \to \mathsf{F}\ (A \times C) & \\ fzip_{\mathsf{Id}} \qquad f = id & \\ fzip_{\underline{C}} \qquad\ f = \pi_1 & \\ fzip_{(\mathsf{F} \otimes \mathsf{G})}\ f = (fzip_{\mathsf{F}}\ f \times fzip_{\mathsf{G}}\ f) \circ distp & \qquad fzip\textsc{-Def} \\ fzip_{(\mathsf{F} \oplus \mathsf{G})}\ f = (fzip_{\mathsf{F}}\ f \bigtriangledown \mathsf{F}\ (id \bigtriangleup f) \circ \pi_1 + \mathsf{G}\ (id \bigtriangleup f) \circ \pi_1 \bigtriangledown fzip_{\mathsf{G}}\ f) \circ dists & \\ fzip_{(\mathsf{F} \odot \mathsf{G})}\ f = \mathsf{F}\ (fzip_{\mathsf{G}}\ f) \circ fzip_{\mathsf{F}}\ (\mathsf{G}\ f) & \end{array}$$

As usual for lenses, *fzip* gives preference to the values from the abstract data type. In the case of sums (similarly to the definition of the $+$ lens), *fzip* is applied recursively to the subfunctors F and G, and whenever the abstract and concrete values are "out of sync", the abstract value is preserved and a new concrete value is created from the abstract value, by invoking the argument function. We can prove additional laws about $fzip_{\mathsf{F}}$:

$$put_{\mathsf{F}\,f} = \mathsf{F}\ put_f \circ fzip_{\mathsf{F}}\ create_f \qquad fzip\text{-PUT}$$

$$\mathsf{F}\ \pi_1 \circ fzip_{\mathsf{F}}\ f = \pi_1 \qquad fzip\text{-CANCEL}$$

$$fzip_{\mathsf{F}}\ f \circ (\mathsf{F}\ g \vartriangle \mathsf{F}\ h) = \mathsf{F}\ (g \vartriangle h) \qquad fzip\text{-SPLIT}$$

$$fzip_{\mathsf{F}}\ f \circ (\mathsf{F}\ g \times id) = \mathsf{F}\ (g \times id) \circ fzip_{\mathsf{F}}\ (f \circ g) \qquad fzip\text{-NAT}$$

The first shows that $fzip_{\mathsf{F}}$ does indeed subsume the alignment behavior of $put_{\mathsf{F}\,f}$. The second states that $fzip_{\mathsf{F}}$ cannot modify the shape of the abstract type, nor the data contained in it. The third states that zipping two "in sync" values can be trivially done just by mapping. The fourth law formulates a kind of naturality law for $fzip_{\mathsf{F}}$ modulo its concrete value generation function. The proof of the second property can be found in Appendix A.

4.2.2 Catamorphisms

Catamorphisms can be lifted to well-behaved lenses as follows:

$$\begin{aligned}
&\forall f : \mathsf{F}\ A \trianglerighteq A.\ \llparenthesis f \rrparenthesis_{\mathsf{F}} : \mu\mathsf{F} \trianglerighteq A \\
&get \quad\ \ = \llparenthesis get_f \rrparenthesis_{\mathsf{F}} \\
&put \quad\ \ = \llbracket fzip_{\mathsf{F}}\ create \circ (put_f \circ (id \times \mathsf{F}\ get) \vartriangle \pi_2) \circ (id \times out_{\mathsf{F}}) \rrbracket_{\mathsf{F}} \\
&create = \llbracket create_f \rrbracket_{\mathsf{F}}
\end{aligned}$$

Notice how put, encoded as an anamorphism, still gives preference to abstract values by using $fzip$, as depicted in the following diagram:

$$\begin{tikzcd}
\mu\mathsf{F} & & & & \mathsf{F}\ \mu\mathsf{F} \arrow[llll, "in_{\mathsf{F}}"'] \\
A \times \mu\mathsf{F} \arrow[u, "put"] \arrow[r, "id \times out_{\mathsf{F}}"'] & A \times \mathsf{F}\ \mu\mathsf{F} \arrow[r, "id \times \mathsf{F}\ get"'] \arrow[rr, bend left, "\pi_2"] \arrow[rr, phantom, "\vartriangle"] & A \times \mathsf{F}\ A \arrow[r, "put_f"'] & \mathsf{F}\ A \times \mathsf{F}\ \mu\mathsf{F} \arrow[r, "fzip_{\mathsf{F}}\ create"'] & \mathsf{F}\ (A \times \mu\mathsf{F}) \arrow[u, "\mathsf{F}\ put"']
\end{tikzcd}$$

To prove that $\llparenthesis f \rrparenthesis$ is a well-behaved lens, we must prove that $create$ terminates, i.e., is a recursive anamorphism. Assuming this condition, the proofs that put is always a recursive anamorphism and that $\llparenthesis f \rrparenthesis$ is well-behaved are given in Section B.2.2. The proofs of laws CREATEGET and PUTGET can be done using the uniqueness law for hylomorphisms, and the proof of law GETPUT uses the fusion law for anamorphisms. We can also prove that, whenever the *create* anamorphism is recursive, our catamorphism lens also has uniqueness in the LENS category:

$$f = \llparenthesis g \rrparenthesis_{\mathsf{F}} \;\Leftrightarrow\; f \circ in_{\mathsf{F}} = g \circ \mathsf{F}\, f \qquad \llparenthesis\cdot\rrparenthesis\text{-UNIQ}$$

Proof. This proof can be factorized into the following three lemmas:

$$\begin{array}{rcl}
get_f = get_{\llparenthesis g \rrparenthesis_{\mathsf{F}}} & \Leftrightarrow & get_{f \circ in_{\mathsf{F}}} = get_{g \circ \mathsf{F}\, f} \\
create_f = create_{\llparenthesis g \rrparenthesis_{\mathsf{F}}} & \Leftrightarrow & create_{f \circ in_{\mathsf{F}}} = create_{g \circ \mathsf{F}\, f} \\
put_f = put_{\llparenthesis g \rrparenthesis_{\mathsf{F}}} & \Leftrightarrow & put_{f \circ in_{\mathsf{F}}} = put_{g \circ \mathsf{F}\, f} \;\Leftarrow\; \begin{array}{l} get_f = get_{\llparenthesis g \rrparenthesis_{\mathsf{F}}} \\ create_f = create_{\llparenthesis g \rrparenthesis_{\mathsf{F}}} \end{array}
\end{array}$$

Again, the first follows directly from the unidirectional uniqueness. The proof of the remaining is presented in Section B.2.2. □

We can also derive the lens versions of the well-known fold laws from uniqueness in LENS:

$$\begin{array}{ll}
\llparenthesis in_{\mathsf{F}} \rrparenthesis_{\mathsf{F}} = id & \llparenthesis\cdot\rrparenthesis\text{-REFLEX} \\
\llparenthesis g \rrparenthesis_{\mathsf{F}} \circ in_{\mathsf{F}} = g \circ \mathsf{F}\, \llparenthesis g \rrparenthesis_{\mathsf{F}} & \llparenthesis\cdot\rrparenthesis\text{-CANCEL} \\
f \circ \llparenthesis g \rrparenthesis_{\mathsf{F}} = \llparenthesis h \rrparenthesis_{\mathsf{F}} \;\Leftarrow\; f \circ g = h \circ \mathsf{F}\, f & \llparenthesis\cdot\rrparenthesis\text{-FUSION}
\end{array}$$

Examples The *map* and *filter_left* functions over lists from Section 2.1 are examples of fold lenses for which it is not difficult to prove that the unfold for *create* (and thus the unfold for *put*) is indeed recursive. Although the lifted *map* is defined exactly as its unidirectional version, the definition of the filtering lens uses the left-biased version of the either combinator:

$$\begin{array}{ll}
\mathit{filter\_left} : [A + B] \trianglerighteq [A] & \\
\mathit{filter\_left} = \llparenthesis (in_{\mathsf{List}_A} \,{}_{\bullet}\!\nabla\, \pi_2) \circ coassocl \circ (id + distl) \rrparenthesis_{\mathsf{List}_{A+B}} & \mathit{filter\_left}\text{-DEF}
\end{array}$$

We do not provide a default function to π_2 because it would never be used by ${}_{\bullet}\!\nabla$. By examining this lens, we get the expected definitions for *create* and *put*. For better understanding, we present them using Haskell syntax and explicit recursion (easily derivable from the original point-free definition): The $create$ corresponds to the Haskell $map\ Left$ function that maps a list into a list of left alternatives:

$$\begin{aligned}
&create :: [\,a\,] \rightarrow [\,Either\ a\ b\,]\\
&create\ [\,] \qquad\quad = [\,]\\
&create\ (x : xs) = Left\ x : create\ xs
\end{aligned}$$

The put function restores right alternatives from the original concrete list:

$$\begin{aligned}
&put :: ([\,a\,], [\,Either\ a\ b\,]) \rightarrow [\,Either\ a\ b\,]\\
&put\ (xs, Right\ y : ys) &&= Right\ y : put\ (xs, ys)\\
&put\ ([\,], \_) &&= [\,]\\
&put\ (x : xs, [\,]) &&= Left\ x : put\ (xs, create\ xs)\\
&put\ (x : xs, Left\ y : ys) &&= Left\ x : put\ (xs, ys)
\end{aligned}$$

For an example of a catamorphism lens whose $create$ function is not recursive, just replace the left-biased either combinator by the right-biased version in the point-free definition presented above. This change yields the following redefinition of $create$ (assuming $\underline{b}$ to be a default constant function that parameterizes π_2) that is visibly not terminating:

$$\begin{aligned}
&create :: [\,a\,] \rightarrow [\,Either\ a\ b\,]\\
&create\ xs = Right\ b : create\ xs
\end{aligned}$$

We can also show that the mapping lens preserves its unidirectional properties, plus a lifted law ruling its interaction with filtering. Remember that these laws are proved directly at the lens level using algebraic laws in LENS:

$$map\ id = id \tag{map-FUNCTOR-ID}$$

$$map\ f \circ map\ g = map\ (f \circ g) \tag{map-FUNCTOR-COMP}$$

$$\llparenthesis g \rrparenthesis_{\mathsf{List}_B} \circ map\ f = \llparenthesis g \circ (id + f \times id) \rrparenthesis_{\mathsf{List}_A} \tag{$\llparenthesis\cdot\rrparenthesis$-$map$-FUSION}$$

$$filter\_left \circ map\ (f + g)^{j,k} = map\ f \circ filter\_left \tag{$filter\_left$-NAT}$$

4.2.3 Anamorphisms

We can also "lensify" anamorphisms using the same ingredients. For the resulting lens to be well-behaved, the coalgebra of the anamorphism must be recursive and itself a well-behaved lens. The generic definition is as follows:

$$\begin{aligned}
&\forall f : A \trianglerighteq \mathsf{G}\ A.\ \llbracket f \rrbracket_{\mathsf{G}} : A \trianglerighteq \mu\mathsf{G} \\
&get \quad\ \ = \llbracket get_f \rrbracket_{\mathsf{G}} \\
&put \quad\ \ = \llbracket put_f, fzip_{\mathsf{G}}\ create \circ (out_{\mathsf{G}} \times get_f) \vartriangle \pi_2 \rrbracket_{\mathsf{G} \otimes \underline{A}} \\
&create = (\!| create_f |\!)_{\mathsf{G}}
\end{aligned}$$

Knowing that $create_f$ is an algebra with type $\mathsf{G}\ A \to A$, $create$ is trivially defined using a catamorphism. The generic definition of put uses an accumulation technique (Pardo, 2003) implemented as a recursive hylomorphism: it proceeds inductively over the abstract value, using the concrete value as an accumulator. The function that propagates the accumulator to recursive calls is $\tau = fzip_{\mathsf{G}}\ create \circ (out_{\mathsf{G}} \times get_f)$. The diagram for this hylomorphism is the following:

$$\begin{tikzcd}
\mu\mathsf{G} \times A \arrow[r, "out_{\mathsf{G}} \times get_f"] \arrow[d, "put"'] \arrow[rr, bend right=20, "\vartriangle", "\pi_2"'] & \mathsf{G}\ \mu\mathsf{G} \times \mathsf{G}\ A \arrow[r, "fzip_{\mathsf{G}}\ create"] & \mathsf{G}\ (\mu\mathsf{G} \times A) \times A \arrow[d, "\mathsf{G}\ put \times id"] \\
A & & \mathsf{G}\ A \times A \arrow[ll, "put_f"]
\end{tikzcd}$$

The anamorphism is well-behaved if get is a recursive anamorphism. In the above definition, we defined put as a general hylomorphism instead of as an accumulation because the function τ is not *proper for accumulation* (Pardo, 2003): in particular, $fzip_{\mathsf{F}}$ is not polymorphic as witnessed by the $fzip$-NAT law. Nevertheless, we can prove that our put is always a recursive hylomorphism, and thus a terminating function (Section B.2.3). The proof that this lens is well-behaved is given in Section B.2.3. The proofs of laws CREATEGET and PUTGET can be done using the fusion law for anamorphisms. The proof of law GETPUT uses hylomorphism fusion and uniqueness. Likewise the fold, it is possible to prove that the bidirectional version of unfold also has uniqueness (Section B.2.3) and the remaining derived laws:

$$f = \llbracket g \rrbracket_{\mathsf{F}} \ \Leftrightarrow\ out_{\mathsf{F}} \circ f = \mathsf{F}\ f \circ g \qquad \llbracket \cdot \rrbracket\text{-UNIQ}$$

$$\llbracket out_{\mathsf{F}} \rrbracket_{\mathsf{F}} = id \qquad \llbracket \cdot \rrbracket\text{-REFLEX}$$

$$out_{\mathsf{F}} \circ \llbracket g \rrbracket_{\mathsf{F}} = \mathsf{F}\ \llbracket g \rrbracket_{\mathsf{F}} \circ g \qquad \llbracket \cdot \rrbracket\text{-CANCEL}$$

$$\llbracket g \rrbracket_{\mathsf{F}} \circ f = \llbracket h \rrbracket_{\mathsf{F}} \ \Leftarrow\ g \circ f = \mathsf{F}\ f \circ h \qquad \llbracket \cdot \rrbracket\text{-FUSION}$$

Examples As an example, the zip function from Section 2.1 can be lifted to the following lens:

$$\begin{aligned}
&\mathit{zip} : [A] \times [B] \rightarrow [A \times B] \\
&\mathit{zip} = [\![(!^{\underline{c} \circ !} + \mathit{distp}) \circ \mathit{coassocl} \circ \mathit{dists} \circ (\mathit{out}_{\mathsf{List}_A} \times \mathit{out}_{\mathsf{List}_B})]\!]_{\mathsf{List}_{A \times B}} \qquad \mathit{zip}\text{-DEF}
\end{aligned}$$

The coalgebra of the *zip* anamorphism lens is illustrated in the following diagram:

$$\begin{tikzcd}
{[A] \times [B]} \arrow[d, "\mathit{out}_{\mathsf{List}_A} \times \mathit{out}_{\mathsf{List}_B}"] \\
(1 + A \times [A]) \times (1 + B \times [B]) \arrow[d, "\mathit{coassocl} \circ \mathit{distp}"] \\
((1 \times 1 + 1 \times (B \times [B])) + (A \times [A]) \times 1) + (A \times [A]) \times (B \times [B]) \arrow[d, "(!^{\underline{c} \circ !} + \mathit{distp})"'] \\
1 + (A \times [A]) \times (B \times [B])
\end{tikzcd}$$

We define the constant c as $L\ (L\ (1,1))$, meaning that the empty list is put back as two empty source lists. From the diagram, other possible choices would be to append an additional suffix to only one of the generated source lists, what would satisfy the lens laws. The *create* induced by this lens is commonly known as the *unzip* function, a fold that recursively splits a list of pairs into two lists:

$$\begin{array}{ll}
\mathit{create} :: [(a,b)] \rightarrow ([a],[b]) & \\
\mathit{create}\ [\,] & = ([\,],[\,]) \\
\mathit{create}\ ((x,y) : t) & = \mathbf{let}\ (\mathit{xs},\mathit{ys}) = \mathit{create}\ t\ \mathbf{in}\ (x : \mathit{xs}, y : \mathit{ys})
\end{array}$$

The *put* has a more intricate behavior: it only recovers elements of one of the original concrete lists when the updated abstract list is smaller than it but with exactly the same length of the other concrete list. This guarantees that zipping the result again yields the same view. For example, $\mathit{put}\ ([(1,2),(3,4)],([4,5],[6,7,8,9]))$ returns $([1,3],[2,4,8,9])$. Notice how the elements 8 and 9 of the bigger list are recovered:

$$\begin{array}{ll}
\mathit{put} :: ([(a,b)],([a],[b])) \rightarrow ([a],[b]) & \\
\mathit{put}\ ([\,],([\,],r)) & = ([\,],r) \\
\mathit{put}\ ([\,],(l,[\,])) & = (l,[\,]) \\
\mathit{put}\ ((x,y) : t, (\_ : l, \_ : r)) & = \mathbf{let}\ (\mathit{xs},\mathit{ys}) = \mathit{put}\ (t,(l,r))\ \mathbf{in}\ (x : \mathit{xs}, y : \mathit{ys}) \\
\mathit{put}\ (l,\_) & = \mathit{create}\ l
\end{array}$$

Naturally, not all valid unidirectional laws in SET can be lifted to bidirectional laws in LENS. A perhaps surprising example of a law that is not preserved at the lens level is *zip*-NAT. For any two lenses $f : A \trianglerighteq C$ and $g : B \trianglerighteq D$, consider the following lens compositions corresponding to the left and right sides of the *zip*-NAT law, respectively:

$$\begin{aligned} zipmap &= zip \circ (map\ f \times map\ g) \\ mapzip &= map\ (f \times g) \circ zip \end{aligned}$$

In LENS, *mapzip* can be fused into the following lens anamorphism:

$$[\![(!^{\underline{c}\circ !} + ((f \times g) \times id) \circ distp) \circ coassocl \circ dists \circ (out_{\mathsf{List}_A} \times out_{\mathsf{List}_B})]\!]_{\mathsf{List}_{C \times D}}$$

However, due to the side-conditions of the *distl*-NAT and $+$-COMP laws on lenses, applying fusion to *zipmap* yields a slightly different lens anamorphism:

$$[\![(!^{\underline{c}\circ !} + ((f \times g) \times id) \circ distp)^{x,y} \circ coassocl \circ dists \circ (out_{\mathsf{List}_A} \times out_{\mathsf{List}_B})]\!]_{\mathsf{List}_{C \times D}}$$

Here, the x and y variables constitute more refined creator functions that resort to put_f and put_g in some particular cases when only an A or a B element exists in the source lists for a $A \times B$ pair in the view, instead of simply applying their *create*s.

Consider that we instantiate $f = \pi_1{}^{\underline{\texttt{'x'}}\circ !}$ and $g = \pi_1{}^{\underline{\texttt{'y'}}\circ !}$. Then, executing $put_{zipmap}\ ([(1,2)], ([], [(2, \texttt{'a'})]))$ returns $([(1, \texttt{'x'})], [(2, \texttt{'a'})])$, where the inserted number 1 has been assigned a default $\texttt{'x'}$ value and the $\texttt{'a'}$ value for the number 2 has been recovered from the original right list. On the other hand, if we run $put_{mapzip}\ ([(1,2)], ([], [(2, \texttt{'a'})]))$ the result is $([(1, \texttt{'x'})], [(2, \texttt{'y'})])$, where the original $\texttt{'a'}$ associated to 2 has been replaced with the default $\texttt{'y'}$. In the first case, put_{zip} is applied to the view list and to the result of *map*ping the source pair into $([], [2])$, followed by applying $put_{map\ f \times map\ g}$. In the second case, the composition is performed in a dual way: $put_{map\ (f \times g)}$ is applied to the view list and to the result of *zip*ping the source pair into $([], [])$ (since get_{zip} returns a zipped list with size equal to the smaller input list), followed by applying put_{zip}. The practical difference in the latter is that the original $\texttt{'a'}$ is not restored, because get_{zip} will remove the element $(2, \texttt{'a'})$ from the right source list, meaning that $put_{map\ (f \times g)}$ will not recover such element.

As for catamorphisms, we can prove that our mapping lens also preserves fusion with bidirectional anamorphisms (using its encoding as an unfold and unfold uniqueness):

$$map\ f \circ [\![g]\!]_{\mathsf{List}_A} = [\![(id + f \times id) \circ g]\!]_{\mathsf{List}_B} \qquad [\![\cdot]\!]\text{-}map\text{-FUSION}$$

4.2.4 Natural Transformations

A special case of the previous lenses occurs when the forward transformation is both expressible as a catamorphism and an anamorphism, with the same natural transformation in the recursive gene building its corresponding algebra or coalgebra. Unlike the previous cases, where we still have to check that the coalgebras are recursive, given a natural lens $\eta : \mathsf{F} \dot{\unrhd} \mathsf{G}$, both $(\!|in_{\mathsf{G}} \circ \eta|\!)_{\mathsf{F}}$ and $[\![\eta \circ out_{\mathsf{F}}]\!]_{\mathsf{G}}$ immediately determine well-behaved lenses between $\mu\mathsf{F}$ and $\mu\mathsf{G}$ because termination is guaranteed for the respective anamorphisms. In fact, for the lenses $(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}$ and $[\![f \circ out_{\mathsf{F}}]\!]_{\mathsf{G}}$ to be well-behaved, it is sufficient that the gene $f : \mathsf{F}\ A \unrhd \mathsf{G}\ A$ is “almost a natural lens” (Section B.2.4). By this, we mean that its forward transformation is a natural transformation $get_f : \mathsf{F} \dot{\to} \mathsf{G}$, but its backward transformations $put_f : \mathsf{G}\ A \times \mathsf{F}\ A \to \mathsf{F}\ A$ and $create_f : \mathsf{F}\ A \to \mathsf{G}\ A$ can be defined for the particular type A, with $A = \mu\mathsf{G}$ for folds and $A = \mu\mathsf{F}$ for unfolds.

There are several examples of these lenses. As seen before, the map function is a well-known example that can be expressed either as a catamorphism from lists or an anamorphism to lists. Likewise, we can lift generic mapping into a catamorphism or an anamorphism lens and show that all type functors (as least fixed points of regular binary functors) denote well-behaved lenses in LENS:

$$\forall f : A \to C.\ T\ f : T\ A \to T\ C$$

$$T\ f = (\!|in_{\mathsf{B}\ C} \circ \mathsf{B}\ f\ id|\!)_{\mathsf{B}\ A} \qquad \text{MAP-DEF}$$

$$T\ f = [\![\mathsf{B}\ f\ id \circ out_{\mathsf{B}\ A}]\!]_{\mathsf{B}\ C} \qquad \text{MAP-}[\![\cdot]\!]\text{-DEF}$$

We can also show that our generic mapping lens preserves its unidirectional laws:

$$T\ id = id \qquad \text{MAP-FUNCTOR-ID}$$

$$T\ f \circ T\ g = T\ (f \circ g) \qquad \text{MAP-FUNCTOR-COMP}$$

$$(\!|g|\!)_{\mathsf{B}\ C} \circ T\ f = (\!|g \circ \mathsf{B}\ f\ id|\!)_{\mathsf{B}\ A} \qquad (\!|\cdot|\!)\text{-MAP-FUSION}$$

$$T\ f \circ [\![g]\!]_{\mathsf{B}\ A} = [\![\mathsf{B}\ f\ id \circ g]\!]_{\mathsf{B}\ C} \qquad [\![\cdot]\!]\text{-MAP-FUSION}$$

This entails that we can also extend lens functor mapping to regular functors, and therefore define catamorphism and anamorphism lenses over regular inductive data types. For example, for a type functor T such that for every A we have $T\ A = \mu(\mathsf{B}\ A)$, we can define $fzip_T$ as a general zipping function that generalizes zip on lists[1]:

[1] Actually, *zip* merges two lists while dropping elements if one list is longer than the other, while

$$
\begin{aligned}
&\mathit{fzip}_{\mathsf{F}} : (A \to C) \to \mathsf{F}\ A \times \mathsf{F}\ C \to \mathsf{F}\ (A \times C)\\
&\mathit{fzip}_{T}\ f = [\![\mathit{bzip}_{\mathsf{B}}\ f\ (T\ f) \circ (\mathit{out}_{\mathsf{B}\ A} \times \mathit{out}_{\mathsf{B}\ C})]\!]_{\mathsf{B}\ (A \times C)}
\end{aligned}
$$

The proof that this anamorphism is recursive is similar to the proof that $\mathit{put}_{[\![g]\!]_{\mathsf{F}}}$ is recursive. General bifunctor zipping can be defined for regular bifunctors as follows:

$$
\begin{aligned}
&\mathit{bzip}_{\mathsf{B}} : (A \to C) \to (B \to D) \to \mathsf{B}\ A\ B \times \mathsf{B}\ C\ D \to \mathsf{B}\ (A \times C)\ (B \times D)\\
&\mathit{bzip}_{\mathsf{Id}}\ \ f\ g \quad = \mathit{id}\\
&\mathit{bzip}_{\mathsf{Par}}\ f\ g \quad = \mathit{id}\\
&\mathit{bzip}_{\underline{C}}\ \ f\ g \quad = \pi_1\\
&\mathit{bzip}_{\mathsf{F} \otimes \mathsf{G}}\ \ f\ g = (\mathit{bzip}_{\mathsf{F}}\ f\ g \times \mathit{bzip}_{\mathsf{G}}\ f\ g) \circ \mathit{distp} \qquad \mathit{bzip}\text{-}\mathrm{DEF}\\
&\mathit{bzip}_{(\mathsf{F} \oplus \mathsf{G})}\ f\ g = (\mathit{bzip}_{\mathsf{F}}\ f\ g \mathbin{\triangledown} \mathsf{G}\ (\mathit{id} \mathbin{\vartriangle} f)\ (\mathit{id} \mathbin{\vartriangle} g) \circ \pi_1\\
&\qquad\qquad + \mathsf{G}\ (\mathit{id} \mathbin{\vartriangle} f)\ (\mathit{id} \mathbin{\vartriangle} g) \circ \pi_1 \mathbin{\triangledown} \mathit{bzip}_{\mathsf{G}}\ f\ g) \circ \mathit{dists}\\
&\mathit{bzip}_{(\mathsf{F} \odot \mathsf{B})}\ f\ g = \mathsf{F}\ (\mathit{bzip}_{\mathsf{B}}\ f\ g) \circ \mathit{fzip}_{\mathsf{F}}\ (B\ f\ g)
\end{aligned}
$$

Another lens that establishes a natural transformation between the base functors of lists and naturals is *length*, that computes the length of a list:

$$
\begin{aligned}
&\mathit{length}^{A} : [A] \trianglerighteq \mathit{Nat}\\
&\mathit{length}^{v} = \llparenthesis \mathit{in}_{\mathsf{Nat}} \circ (\mathit{id} + {\pi_2}^{\underline{v} \circ !}) \rrparenthesis_{\mathsf{List}_A} = [\![(\mathit{id} + {\pi_2}^{\underline{v} \circ !}) \circ \mathit{out}_{\mathsf{List}_A}]\!]_{\mathsf{Nat}} \qquad \mathit{length}\text{-}\mathrm{DEF}
\end{aligned}
$$

The parameter v is the default value of type A to be inserted in the source list when the target length increases. In this example, the gene of the catamorphism and anamorphism is a natural lens $\mathit{id} + {\pi_2}^{\underline{v}} : \underline{1} \oplus \underline{A} \otimes \mathsf{Id} \trianglerighteq \underline{1} \oplus \mathsf{Id}$, since the constant function $\underline{v}$ does not depend on the type of the functor argument.

Using $\llparenthesis\cdot\rrparenthesis$-*map*-FUSION, the proof that computing the length of a list cancels mapping can be done as follows

$$
\begin{aligned}
&\mathit{length}^{v} \circ \mathit{map}\ f\\
&\ = \{\, \mathit{length}\text{-}\mathrm{DEF};\ \llparenthesis\cdot\rrparenthesis - \mathit{map}\text{-}\mathrm{FUSION} \,\}\\
&\llparenthesis \mathit{in}_{\mathsf{Nat}} \circ (\mathit{id} + {\pi_2}^{\underline{v} \circ !}) \circ (\mathit{id} + f \times \mathit{id}) \rrparenthesis_{\mathsf{List}_A}\\
&\ = \{\, +\text{-}\mathrm{FUNCTOR}\text{-}\mathrm{COMP};\ \pi_2 - \mathrm{NAT} \,\}\\
&\qquad \mathit{create}_f \circ \underline{v} \circ !\ \circ \mathit{get}_{\mathit{id}}\\
&\qquad\ = \{\, !\ - \mathrm{FUSION};\ \text{definition of } \mathit{get} \,\}
\end{aligned}
$$

$\mathit{fzip}_{[\,]}$ merges two lists such that the merged list has the same length as the left input list.

$$\begin{aligned}
&\underline{create_f\ v \circ !} \\
&(\!|in_{\mathsf{Nat}} \circ (id + \pi_2{}^{\underline{create_f\ v \circ !}})|\!)_{\mathsf{List}_A} \\
&\quad = \{\, length\text{-}\mathrm{DEF} \,\} \\
&length^{create_f\ v}
\end{aligned}$$

and is captured by the following naturality equation:

$$length^{v} \circ map\ f = length^{create_f\ v} \qquad length\text{-}\mathrm{NAT}$$

4.2.5 Hylomorphisms

It is well known that most recursive functions can be encoded using hylomorphisms over polynomial functors. Given that $[\![\cdot,\cdot]\!]$-SPLIT allows us to factorize a hylomorphism into the composition of a catamorphism after an anamorphism, the range of recursive functions that we can lift to well-behaved lenses is considerably enlarged. Of course, the algebras and coalgebras of the hylomorphism must themselves be lenses and the coalgebras must be recursive.

Examples Take as an example the natural number addition function $plus$ from Section 2.1, that can be lifted to a lens hylomorphism whose both algebra and recursive coalgebra are lenses:

$$\begin{aligned}
&plus : Nat \times Nat \trianglerighteq Nat \\
&plus = [\![id \triangledown succ, (\pi_2{}^{!} + id) \circ distl \circ (out_{\mathsf{Nat}} \times id)]\!]_{\underline{Nat} \oplus \mathsf{Id}} \qquad plus\text{-}\mathrm{DEF}
\end{aligned}$$

In order to constitute a well-behaved lens, the $create$ and put functions should guarantee that the sum of the generated pair of numbers equals the abstract value. The $create$ automatically derived by the techniques presented above simply creates a pair with the abstract value and a $Zero$ as the second element:

$$\begin{aligned}
&create :: Nat \to (Nat, Nat) \\
&create\ n = (n, Zero)
\end{aligned}$$

As usual, the induced put function is a bit more tricky: if the abstract value is greater than the first element of the concrete pair, that element is preserved and the second element becomes the difference between both; if the abstract value is smaller, it is paired with zero likewise $create$:

$$\begin{aligned}
&put :: (Nat, (Nat, Nat)) \to (Nat, Nat)\\
&put\ (Zero, \_) &&= (Zero, Zero)\\
&put\ (n, (Zero, \_)) &&= (Zero, n)\\
&put\ (Succ\ n, (Succ\ m, o)) &&= \mathbf{let}\ (a, b) = put\ (n, (m, o))\ \mathbf{in}\ (Succ\ a, b)
\end{aligned}$$

A similar lens is binary list concatenation, defined as follows:

$$\begin{aligned}
&cat : [A] \times [A] \trianglerighteq [A]\\
&cat = [\![id \triangledown cons, ({\pi_2}^{!} + assocr) \circ distl \circ (out_{\mathsf{List}_A} \times id)]\!]_{\underline{[A]} \oplus \underline{A} \otimes \mathsf{Id}}
\end{aligned} \tag*{cat-DEF}$$

Here, the intermediate functor $\underline{[A]} \oplus \underline{A} \otimes \mathsf{Id}$ is the base functor NeList_A of the inductive type of lists appended with a list suffix at the rear, defined in Haskell as follows:

$$\mathbf{data}\ NeList\ a = NeNil\ [a] \mid NeCons\ a\ (NeList\ a)$$

We can also generalize binary addition and concatenation to n-ary lenses over lists using catamorphisms:

$$\begin{aligned}
&sum : [Nat] \trianglerighteq Nat\\
&sum = (\!|(zero \triangledown id) \circ (id + plus)|\!)_{\mathsf{List}_{Nat}}
\end{aligned} \tag*{sum-DEF}$$

$$\begin{aligned}
&concat : [[A]] \trianglerighteq [A]\\
&concat = (\!|(nil \triangledown id) \circ (id + cat)|\!)_{\mathsf{List}_{[A]}}
\end{aligned} \tag*{$concat$-DEF}$$

These binary and n-ary lens combinators enjoy interesting laws, such as the following naturality laws modulo map and interaction laws with $length$:

$$map\ f \circ cat = cat \circ (map\ f \times map\ f) \tag*{cat-NAT}$$

$$map\ f \circ concat = concat \circ map\ (map\ f) \tag*{$concat$-NAT}$$

$$length^v \circ cat = plus \circ (length^v \times length^v) \tag*{$length$-CAT}$$

$$length^v \circ concat = sum \circ map\ length^v \tag*{$length$-CONCAT}$$

An example of a lens over a regular type is the following function that flattens a n-ary leaf tree into a list:

$$\begin{aligned}
&lflatten : LTree\ A \to [A]\\
&lflatten = (\!|(wrap \triangledown id) \circ (id + concat)|\!)_{\mathsf{LTree}_A}
\end{aligned} \tag*{$lflatten$-DEF}$$

Similarly to other jointly-surjective lenses, $\mathit{wrap} \triangledown \mathit{id} : A + [A] \trianglerighteq [A]$ can be defined by unwrapping the input list and using lens combinators on sums. The associated naturality law for $\mathit{lflatten}$, modulo type functor mapping, is also valid at the lens level:

$$\mathit{map}\ f \circ \mathit{lflatten} = \mathit{lflatten} \circ \mathit{LTree}\ f \qquad \mathit{lflatten}\text{-}\textsc{Nat}$$

4.3 Summary

We have shown how to lift most of the standard point-free combinators and recursion patterns to total well-behaved lenses, enabling the definition of complex generic lenses over inductive data types. Additionally, we have identified precise termination conditions to verify in order to guarantee that recursive lenses like folds and unfolds constitute well-behaved lenses. By putting the emphasis on totality, while keeping a decidable type system that is implemented in functional languages like Haskell, the counterpart is that we have a more limited set of well-behaved lenses than other approaches (Foster et al., 2007; Mu et al., 2004; Matsuda et al., 2007; Voigtländer, 2009), namely not all point-free combinators denote well-behaved lenses.

The point-free style being so intertwined with algebraic calculation, we have studied the algebraic laws preserved by our lifting to lenses and proposed an equational calculus to reason directly about lenses defined in our point-free language. Apart from few side-conditions to control the non-determinism of the backward transformations, this calculus allows us to hide the complex backward synchronization behavior and perform conventional proofs at the lens level, by calculating with lenses using only their forward point-free specification.

Unfortunately, for recursive lenses defined using our generic recursion patterns, the user still has to prove that the corresponding coalgebras are recursive. While for anamorphisms this problem is more controlled, since only the termination of the forward transformation (whose syntax matches that of the lens specification) must be proved independently of the backward transformations, for catamorphisms the dual create backward transformation must be proved to be a recursive anamorphism. Nevertheless, we have identified an interesting class of recursive lenses for which termination proofs can be discharged, namely if a lens can be specified either as a catamorphism or an anamorphism.

Chapter 5

Generic Point-free Delta Lenses

Our point-free lens language from the previous chapter is state-based: a lens $S \trianglerighteq V$ encompasses a forward transformation $get : S \rightarrow V$ that translates a modified source into a new view, and two backward transformations $create : V \rightarrow S$ and $put : V \times S \rightarrow S$ that translate a modified view (with optional knowledge of the original source) into a consistently modified source.

As discussed in Chapter 3, the above state-based formulation, where updates are represented simply by their post-states, underpins many bidirectional languages. Although very flexible, this formulation provides little knowledge to the put function, whose behavior can be largely non-deterministic due to the existence of many possible source translations for a particular view update. For that reason, put must somehow *align* models to recover a more informative high-level description estimating the performed view update (a *delta* describing the relation between elements of the modified and original view), to be used to guide the propagation of the view modifications to the source model. In fact, a large part of the non-determinism in the design space of a state-based bidirectional language concerns precisely the choice of a suitable alignment strategy.

Like our language from Chapter 4, some state-based languages (Foster et al., 2007; Matsuda et al., 2007) do not consider this alignment step and end up aligning values positionally, agnostically to the actual view update, i.e., elements of the view are always matched with elements of the source at the same positions, even when they are rearranged by an update. This suffices for in-place updates that only modify data locally without affecting their order, but produces unsatisfactory results for many other examples. Other state-based languages (Xiong et al., 2007; Bohannon et al., 2008) go slightly further and align values by keys rather than by positions. Nevertheless, this specific alignment strategy is likewise fixed in the language and might not be suitable for values without natural keys (or for translating updates that modify keys themselves).

On the other hand, some operation-based bidirectional languages (Mu et al., 2004; Hidaka et al., 2010; Hofmann et al., 2012) avoid this potential alignment mismatch by relying on an alternative formulation, where the backward transformation receives the exact low-level sequence of edit operations. The drawback of this approach is that put only considers a fixed update language (typically allowing just add, delete, and move operations), defined over very specific types, making it harder to integrate such languages in a legacy application that does not record such edits.

In-between both worlds is the abstract delta-based framework proposed by Diskin et al. (2011a) that encompasses an explicit alignment operation for computing view *deltas*, and where put is a delta-based transformation that propagates view deltas to source deltas. *Matching lenses* (Barbosa et al., 2010) are the first bidirectional language that we are aware of promoting this separation principle: they decompose strings into a rigid structure or shape, a container with “holes” denoting data placeholders, and a list of data elements that populate such shape. This enables elements to be freely rearranged according to the delta information. Users can then specify an alignment strategy that computes the view update delta as a correspondence between element positions.

The main limitation of matching lens combinators is that they are shape preserving: when recast in the context of general user-defined data types, their expressivity amounts to a mapping transformation $T\ l : T\ A \trianglerighteq T\ B$ over a polymorphic data type, with $l : A \trianglerighteq B$ being a regular state-based lens operating on its elements. In this setting, lenses are sensitive to data modifications (on data values of type A and B) but not to shape modifications (on the T shape component of values) and the behavior of the backward transformation is rather simple: it just copies the shape of the view, overlapping the original source shape, and realigns elements using their explicitly computed delta correspondences rather than by their positions.

Consider, as an example, an XML document representing a genealogical tree of persons conforming to the XML Schema from Figure 5.1

```
<tree>
 <person name="Peter" birth="1981"/>
 <tree>
  <person name="Joseph" birth="1955"/>
  <tree>
   <person name="Luigi" birth="1920"/>
   <tree/><tree/>
  </tree>
  <tree>
   <person name="Margaret" birth="1923"/>
   <tree/><tree/>
  </tree>
 </tree>
 <tree>
  <person name="Mary" birth="1956"/>
  <tree/><tree/>
 </tree>
</tree>
```

from which we can compute a sequence of names of left (male) ascendants in the tree:

```
<males>
 <male name="Peter"</male>
 <male name="Joseph"</male>
 <male name="Luigi"</male>
</males>
```

By encoding the recursive XML Schema from Figure 5.1 in Haskell as a binary tree of persons, we can write the above transformation using the language of point-free lenses from Chapter 4:

$$\begin{array}{ll}
\mathbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) & \\
\mathbf{data}\ \mathit{Males} = \mathit{Males}\ [\mathit{Male}] & \\
\mathbf{data}\ \mathit{Person} = \mathit{Person}\ \mathit{Name}\ \mathit{Birth} & \mathbf{type}\ \mathit{Name} = \mathit{String} \\
\mathbf{data}\ \mathit{Male} = \mathit{Male}\ \mathit{Name} & \mathbf{type}\ \mathit{Birth} = \mathit{Int}
\end{array}$$

$$\begin{array}{l}
\mathit{fathernames} : \mathit{Tree}\ \mathit{Person} \trianglerighteq \mathit{Males} \\
\mathit{fathernames} = \mathit{in}_{\mathsf{Males}} \circ \mathit{names} \circ \mathit{fatherline}
\end{array}$$

$$\begin{array}{l}
\mathit{names} : [\mathit{Person}] \trianglerighteq [\mathit{Male}] \\
\mathit{names} = \mathit{map}\ (\mathit{in}_{\mathsf{Male}} \circ \pi_1{}^{\underline{2012} \circ !} \circ \mathit{out}_{\mathsf{Person}})
\end{array}$$

$$\begin{array}{ll}
\mathit{fatherline} : \mathit{Tree}\ \mathit{Person} \trianglerighteq [\mathit{Person}] & \mathit{lspine} : \mathit{Tree}\ A \trianglerighteq [A] \\
\mathit{fatherline} = \mathit{lspine} & \mathit{lspine} = (\!|\mathit{in}_{\mathsf{List}_A} \circ (\mathit{id} + \mathit{id} \times \pi_1{}^{\underline{[\,]} \circ !})|\!)_{\mathsf{Tree}_A}
\end{array}$$

This transformation is defined in two steps: first compute the left ascendants with *fatherline* by calculating the left spine of the tree (using a default empty list of right ascendants for new elements in the backward direction), and then select only their names using the *names* mapping that converts each `person` element into a `male` element by forgetting birth years (using 2012 as the default birth year for created persons in the backward direction). By porting the matching lens approach to this domain, we could

```
<xs:schema>
 <xs:element name="tree">
  <xs:complexType>
   <xs:sequence minOccurs="0" maxOccurs="1">
    <xs:element name="person" type="Person"/>
    <xs:element ref="tree"/>
    <xs:element ref="tree"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:complexType name="Person">
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="birth" type="xs:short"/>
 </xs:complexType>
</xs:schema>
```

Figure 5.1: XML Schema modeling a family genealogical tree.

easily replace the *map* list mapping lens with a suitable alignment-aware combinator. Unfortunately, *lspine* does not fit the mapping corset imposed by the matching lens framework, since it reshapes the source tree into a list. Leaving *lspine* as a standard state-based (positional) lens would produce less than optimal results. For instance, if we insert a new male named `John` at the head of the view list, and use a "best match" alignment strategy (Barbosa et al., 2010) to infer a view delta (relating every `male` element in the updated view but `John` to the respective `person` element in the original view), *put* would return the following updated source XML tree:

```
<tree>
 <person name="John" birth="2012"/>
 <tree>
  <person name="Peter" birth="1981"/>
  <tree>
   <person name="Joseph" birth="1955"/>
   <tree>
    <person name="Luigi" birth="1920"/>
    <tree/><tree/>
   </tree>
   <tree/>
  </tree>
  <tree>
   <person name="Margaret" birth="1923"/>
   <tree/><tree/>
  </tree>
 </tree>
 <tree>
  <person name="Mary" birth="1956"/>
  <tree/><tree/>
  </tree>
 </tree>
```

Although the order of males in the view changes, the birth years of existing people are retrieved correctly due to the improved behavior of mapping modulo deltas, but the positional shape behavior of *lspine* makes Mary an incorrect parent of John and Margaret an incorrect parent of Peter. With the extra delta information at hand we could have done better though: *lspine* could recognize John as a new male and propagate his insertion to the head of the source tree without affecting the mother relationships of the existing persons in the source:

```
 <tree>
  <person name="John" birth="2012"/>
  <tree>
   <person name="Peter" birth="1981"/>
   <tree>
    <person name="Joseph" birth="1955"/>
    <tree>
     <person name="Luigi" birth="1920"/>
     <tree/><tree/>
    </tree>
    <tree>
     <person name="Margaret" birth="1923"/>
     <tree/><tree/>
    </tree>
   </tree>
   <tree>
    <person name="Mary" birth="1956"/>
    <tree/><tree/>
   </tree>
  </tree>
  <tree/>
 </tree>
```

It is easy to justify that this behavior on shapes induces a smaller change and is thus more predictable.

As another example, imagine that we have the same persons but organized in a list sorted by age, discriminating males and females, according to the schema from Figure 5.2

```
 <people>
  <male name="Peter" birth="1981">
  <female name="Mary" birth="1956"/>
  <male name="Joseph" birth="1955"/>
  <female name="Margaret" birth="1923"/>
  <male name="Luigi" birth="1920"/>
 </people>
```

```
<xs:schema>
 <xs:element name="people">
  <xs:complexType>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="male" type="Person"/>
    <xs:element name="female" type="Person"/>
   </xs:choice>
  </xs:complexType>
 </xs:element>
 <xs:complexType name="Person">
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="birth" type="xs:short"/>
 </xs:complexType>
</xs:schema>
```

Figure 5.2: XML Schema modeling a list of males and females.

and that we want to select only the females from the list:

```
<females>
 <female name="Mary" birth="1956"/>
 <female name="Margaret" birth="1923"/>
</females>
```

In Haskell, the shape of the schema from Figure 5.2 can be encoded as a list of optionals, whose data elements are the male or female elements in the source. We can select only the females according to the following transformation:

$$\begin{array}{l}
\mathbf{data}\ People = People\ (ListOpt\ Male\ Female) \\
\mathbf{data}\ Females = Females\ [Female] \\
\mathbf{data}\ ListOpt\ a\ b = NilOpt \mid ConsL\ a\ (ListOpt\ a\ b) \mid ConsR\ b\ (ListOpt\ a\ b) \\
\mathbf{data}\ Male = Male\ Name\ Birth \quad \mathbf{data}\ Female = Female\ Name\ Birth
\end{array}$$

$$\begin{array}{l}
females : People \trianglerighteq Females \\
females = in_{\mathsf{Females}} \circ filter\_right \circ out_{\mathsf{People}}
\end{array}$$

$$\begin{array}{l}
filter\_right : ListOpt\ A\ B \trianglerighteq [B] \\
filter\_right = \llparenthesis (in_{\mathsf{List}_B} \mathbin{\underset{\bullet}{\nabla}} \pi_2) \circ coassocl \circ (id + coswap) \rrparenthesis_{\mathsf{ListOpt}_{A\ B}}
\end{array}$$

Again, the specification of *filter_right* is not a mapping. If we consider that it behaves positionally, inserting a new female `Jane` at the head of the sorted view sequence and deleting the rear female `Margaret` (while preserving the delta correspondences for the other persons) would produce the following unsorted source:

```
<people>
 <male name="Peter" birth="1981">
```

```
  <female name="Jane" birth="2012">
  <male name="Joseph" birth="1955"/>
  <female name="Mary" birth="1956"/>
  <male name="Luigi" birth="1920"/>
</people>
```

A better solution would be to use the deltas to recognize the inserted and deleted elements, and propagate the updates to the same relative positions, what would induce a smaller source update that (for this case) would leave the source list sorted by age:

```
<people>
  <female name="Jane" birth="2012">
  <male name="Peter" birth="1981">
  <female name="Mary" birth="1956"/>
  <male name="Joseph" birth="1955"/>
  <female name="Luigi" birth="1920"/>
</people>
```

The lesson to learn is that like a positional *data alignment* (the matching of data elements) is only reasonable for in-place updates, a positional behavior on shapes (that ignores the shapes of the original source and overrides it with the shape of the updated view) is innate for mapping scenarios but again ineffective for shape-changing transformations that restructure source shapes into different target shapes and for which simple overriding for put is not possible. In this chapter, we focus on the treatment and propagation of generic deltas (independently of the more particular heuristic techniques that can be used to infer this information for specific application scenarios), identify the new problem of *shape alignment* (the matching of new and old shapes) and propose to answer it with the development of a delta lens language, whose inhabitants are lenses with an explicit notion of shape and data that can perform both data and shape alignment. Our language is designed in such a way that many lens programs over arbitrary types written in our generic state-based lens language from Chapter 4 can be lifted to generic delta lens programs over the corresponding shapes without significant effort by users.

5.1 Deltas over Polymorphic Inductive Types

Higher-order functors The central requirement for this chapter is the existence of types that have an explicit notion of shape and data. In functional programming, these are known as polymorphic data types like the trees and lists from our examples. As mentioned in Chapter 2, a polymorphic inductive data type $T\ A$ can be defined as the least fixed point $T\ A = \mu(\mathsf{B}\ A)$ of a partially applied bifunctor $\mathsf{B} : \textsc{Set} \to \textsc{Set} \to \textsc{Set}$, with $out_{\mathsf{B}\ A} : \mu(\mathsf{B}\ A) \to \mathsf{B}\ A\ (T\ A)$ and $in_{\mathsf{B}\ A} : \mathsf{B}\ A\ (T\ A) \to \mu(\mathsf{B}\ A)$. The application of $out_{\mathsf{B}\ A}$ to a type results in a one-level unfolding to a sum-of-products representation capable of being processed with point-free combinators. However, in this chapter we want to lift our point-free combinators so that they work not over "flattened" sum-of-products types but over explicit shapes, i.e., we want to compute an explicit sum-of-products functor representation $\mathsf{F}\ A$ that models the shape of $\mathsf{B}\ A\ (T\ A)$.

For this purpose, we will use an alternative formulation and characterize the polymorphic type constructor T as a fixed point $T = \mu\mathcal{F}$ of a higher-order functor $\mathcal{F} : (\textsc{Set} \to \textsc{Set}) \to (\textsc{Set} \to \textsc{Set})$, a functor over functor categories where objects are functors and arrows are natural transformations. This way, we have the applied fixed point $T\ A = \mu\mathcal{F}\ A$ and initial algebras and final coalgebras are the natural isomorphisms $out_{\mathcal{F}} : T \dot{\to} \mathcal{F}\ T$ and $in_{\mathcal{F}} : \mathcal{F}\ T \dot{\to} T$. Although this formulation is more expressive, namely enabling the encoding of non-regular nested data types (Johann and Ghani, 2007) by admitting higher-order functor composition, we restrict the syntax of higher-order functors to the following regular family equivalent to the formulation as regular bifunctors:

$$\mathcal{F} = \underline{A} \mid \mathcal{P}\!ar \mid \mathcal{I}d \mid \mathcal{F} \boxplus \mathcal{F} \mid \mathcal{F} \boxtimes \mathcal{F} \mid \mathsf{F} \boxdot \mathcal{F}$$

In this language, $\underline{A}$ returns the constant type A, $\mathcal{P}\!ar$ denotes the type parameter, $\mathcal{I}d$ denotes recursive invocation and $\boxplus$ and $\boxtimes$ represent higher-order sums and products. Note that we do not support higher-order functor composition as long as our composition $\boxdot$ applies a unary functor to a higher order functor. The functor to the left side of a composition can be a polymorphic inductive data type, as a fixed point of a higher-order regular functor. For example, the type functors of lists, trees and optional lists (by fixing the type of the first polymorphic argument) can be represented as follows:

$$\begin{array}{ll}
[\,] = \mu\mathcal{L}ist & \mathcal{L}ist = \underline{1} \boxplus \mathcal{P}\!ar \boxtimes \mathcal{I}d \\
Tree = \mu\mathcal{T}ree & \mathcal{T}ree = \underline{1} \boxplus \mathcal{P}\!ar \boxtimes (\mathcal{I}d \boxtimes \mathcal{I}d) \\
ListOpt\ A = \mu\mathcal{L}ist\mathcal{O}pt_{A} & \mathcal{L}ist\mathcal{O}pt_{A} = \underline{1} \boxplus (\underline{A} \boxtimes \mathcal{I}d \boxplus \mathcal{P}\!ar \boxtimes \mathcal{I}d)
\end{array}$$

A regular higher-order functor $\mathcal{F}$ can always be reduced to a binary functor B such that $\mathsf{B}\ A\ (\mathsf{F}\ A) \cong \mathcal{F}\ \mathsf{F}\ A$ holds, for every unary functor F and type A[1]. The application of a regular higher-order functor $\mathcal{F}$ to a unary functor F yields a unary functor $\mathcal{F}\ \mathsf{F}$, according to the following specialization equations:

[1] This restriction allows the definition of generic functions over higher-order functors that are not natural transformations, as detailed later in Section 5.4.

$$
\begin{aligned}
&\underline{A}\ \mathsf{F} = \underline{A}\\
&\mathcal{Par}\ \mathsf{F} = \mathsf{Id}\\
&\mathcal{Id}\ \mathsf{F} = \mathsf{F}\\
&(\mathcal{F} \boxplus \mathcal{G})\ \mathsf{F} = \mathcal{F}\ \mathsf{F} \oplus \mathcal{G}\ \mathsf{F}\\
&(\mathcal{F} \boxtimes \mathcal{G})\ \mathsf{F} = \mathcal{F}\ \mathsf{F} \otimes \mathcal{G}\ \mathsf{F}\\
&(\mathsf{G} \boxdot \mathcal{G})\ \mathsf{F} = \mathsf{G} \odot \mathcal{G}\ \mathsf{F}
\end{aligned}
$$

To emphasize the shape, we will often denote a transformation $f : \mathsf{F}\ A \to \mathsf{G}\ B$ between functors F and G applied to data elements of type A and B (i.e., a function with a notion of domain and target shapes) by $f : \mathsf{F}_A \to \mathsf{G}_B$, with the types as subscripts.

Dependent Types In this chapter, we will resort to a dependent type notation to characterize precisely the positions of elements in a shape functor. In non-dependently typed languages such as Haskell, universal type quantification is used to express polymorphism. For example, we can write the identify function $id : \forall A.\ A \to A$. This quantification over types (objects in a category such as SET) can be often made implicit by writing simply $id : A \to A$.

A dependent type is a type that depends on values. To elegantly introduce such dependencies, dependently-typed systems also permit quantification over values, with the types being special cases of values belonging to the sort of all types SET. The dependent function space $\forall a : A.\ B\ a$ characterizes functions that, given a value $a : A$, return values of the dependent type $B\ a$. For example, considering that a dependent type $\mathit{Vec}\ A\ n$ models an n-sized vector of elements of type A, this allows us to write $\forall n : \mathit{Nat}.\ \mathit{Vec}\ A\ n$ as the type of all functions that take a natural number and return a vector of precisely that size. Again using an explicit quantification, the signature for the identity function becomes $id : \forall A : \text{SET}, a : A.\ A$. When B does not depend on a, as is the case of id, this degenerates into the non-dependent function space $A \to B$.

The dependent cartesian product $\Sigma a : A.\ B\ a$ models the type of dependent pairs where the type of the second component depends on the first component. For instance, vectors of arbitrary length could be represented by a dependent pair $\Sigma n : \mathit{Nat}.\ \mathit{Vec}\ A\ n$. Again, when B does not depend on a, the dependent cartesian product models the normal cartesian product $A \times B$.

To simplify the presentation, we will often mark some arguments of a dependent function space as implicit using curly braces in signatures like $f : \forall \{ a : A \}.\ B\ a$, as found in dependently typed languages such as Agda (Norell, 2009). To provide an implicit argument explicitly, we assume implicit function application $f\ \{a\}$. In principle, these parameters can be omitted and their value inferred from the context[2].

Positions Polymorphic inductive data types can also be seen as instances of *container types* (Abbott et al., 2005). A container type $S \triangleright P$ consists of a type of shapes S together with a family P of position types indexed by values of type S. The extension of a container is a functor $[\![S \triangleright P]\!]$, that when applied to a type A (the type of the data) yields the dependent product $\Sigma s : S.\ (P\ s \to A)$. A value of type $[\![S \triangleright P]\!]\ A$ is thus a pair (s, f), where $s : S$ is a shape and $f : P\ s \to A$ is a total function from positions to data elements. A polymorphic data type $T\ A$ is isomorphic to the extension $[\![T\ 1 \triangleright P]\!]\ A$, where the dependent type P of positions can be inductively defined over functor representations (Abbott et al., 2005) and polymorphically for the type A of data elements. For each value $v : T\ A$, we define:

$$
\begin{array}{llll}
P & \multicolumn{2}{l}{: \forall \{T : \textsc{Set} \to \textsc{Set}\}, v : T\ A.\ \textsc{Set}} & \\
P\ \{\mathsf{Id}\} & a & = 1 & \text{-- unit type} \\
P\ \{\underline{C}\} & c & = 0 & \text{-- empty type} \\
P\ \{\mathsf{F} \oplus \mathsf{G}\} & (L\ a) & = P\ \{\mathsf{F}\}\ a & \text{-- left branches} \\
P\ \{\mathsf{F} \oplus \mathsf{G}\} & (R\ b) & = P\ \{\mathsf{G}\}\ b & \text{-- right branches} \\
P\ \{\mathsf{F} \otimes \mathsf{G}\} & (a, b) & = P\ \{\mathsf{F}\}\ a + P\ \{\mathsf{G}\}\ b & \text{-- left or right components} \\
P\ \{\mu \mathcal{F}\} & a & = P\ \{\mathcal{F}\ \mu \mathcal{F}\}\ (out\ a) & \text{-- recursive unfolding}
\end{array}
$$

The type of positions for $P\ \{\mathsf{F} \odot \mathsf{G}\}\ a$ is handled by unrolling the composition $\mathsf{F} \odot \mathsf{G}$ according to the following equations, where $\mathcal{F} \boxcircle \mathsf{G}$ applies a functor G to the parameters of a higher-order functor $\mathcal{F}$:

$$
\begin{array}{ll@{\qquad}ll}
\mathsf{Id} \odot \mathsf{G} & = \mathsf{G} & \mathcal{Id} \boxcircle \mathsf{G} & = \mathcal{Id} \\
\underline{C} \odot \mathsf{G} & = \underline{C} & \underline{C} \boxcircle \mathsf{G} & = \underline{C} \\
(\mathsf{F} \oplus \mathsf{G}) \odot \mathsf{H} & = (\mathsf{F} \odot \mathsf{H}) \oplus (\mathsf{G} \odot \mathsf{H}) & (\mathcal{F} \boxplus \mathcal{G}) \boxcircle \mathsf{G} & = (\mathcal{F} \boxcircle \mathsf{G}) \boxplus (\mathcal{G} \boxcircle \mathsf{G}) \\
(\mathsf{F} \otimes \mathsf{G}) \odot \mathsf{H} & = (\mathsf{F} \odot \mathsf{H}) \otimes (\mathsf{G} \odot \mathsf{H}) & (\mathcal{F} \boxtimes \mathcal{G}) \boxcircle \mathsf{G} & = (\mathcal{F} \boxcircle \mathsf{G}) \boxtimes (\mathcal{G} \boxcircle \mathsf{G}) \\
(\mathsf{F} \odot \mathsf{G}) \odot \mathsf{H} & = \mathsf{F} \odot (\mathsf{G} \odot \mathsf{H}) & (\mathsf{F} \boxdot \mathcal{G}) \boxcircle \mathsf{G} & = \mathsf{F} \boxdot (\mathcal{G} \boxcircle \mathsf{G}) \\
\mu \mathcal{F} \odot \mathsf{G} & = \mu(\mathcal{F} \boxcircle \mathsf{G}) & \mathcal{Par} \boxcircle \mathsf{G} & = \mathsf{G} \boxdot \mathcal{Par}
\end{array}
$$

[2] Throughout this chapter, we will often write implicit parameters mostly for presentation reasons. Although we believe that some of these implicit parameters may be actually inferable by an Agda implementation, we will only implement our concepts in the Haskell non-dependently typed language (Chapter 6), and we do not claim that all such parameters could be inferred by a dependently typed implementation.

Notice that the type of positions is polymorphic over the type of data elements and the shape of a type $T\ A$ is given by $T\ 1$. Therefore, the type of positions for a value of type $T\ A$ is the same as the type of positions for its shape of type $T\ 1$. Intuitively, they both have the same set of positions and the same number of placeholders.

For trees and lists, the above definition produces the following types of positions:

$$\begin{array}{lll}
P\ \{\mathit{Tree}\} & \mathit{Empty} & = 0 \\
P\ \{\mathit{Tree}\} & (\mathit{Node}\ x\ l\ r) & = 1 + P\ \{\mathit{Tree}\}\ l + P\ \{\mathit{Tree}\}\ r \\
P\ \{[\,]\} & [\,] & = 0 \\
P\ \{[\,]\} & (x : t) & = 1 + P\ \{[\,]\}\ t \\
P\ \{\mathit{ListOpt}\ A\} & \mathit{NilOpt} & = 0 \\
P\ \{\mathit{ListOpt}\ A\} & (\mathit{ConsL}\ x\ t) & = P\ \{\mathit{ListOpt}\ A\}\ t \\
P\ \{\mathit{ListOpt}\ A\} & (\mathit{ConsR}\ y\ t) & = 1 + P\ \{\mathit{ListOpt}\ A\}\ t
\end{array}$$

For the $\mathit{ListOpt}$ type, we consider only positions on its second polymorphic argument, by fixing its first argument to the type A. The idea is that the (dependent) type of positions P is a tree resembling the structure of the value on which it depends, but considering only polymorphic elements. This tree representation ensures that each placeholder in the shape of a value is referenced by an unique position. For example, for a list value with length n, the type of position is equivalent to the natural number n denoting the exact number of elements in the list.

Inspired by *shapely types* (Jay, 1995) notation, the isomorphism between a type $T\ A$ and its container $[\![T\ 1 \triangleright P]\!]\ A$ will be witnessed by three functions

$$\begin{array}{ll}
\mathit{shape} & : T\ A \to T\ 1 \\
\mathit{data} & : \forall v : T\ A.\ (P\ v \to A) \\
\mathit{recover} & : [\![T\ 1 \triangleright P]\!]\ A \to T\ A
\end{array}$$

where shape extracts the shape of a shapely type, data extracts a total function from positions to data elements, and $\mathit{recover}$ rebuilds a data value from a shape and an index of data elements. For lists, the shape $[1]$ is isomorphic to the type Nat of naturals, and thus we have $\mathit{shape}\ l = \mathit{length}\ l$, $P\ l = \{0, .., \mathit{length}\ l - 1\}$, and $\mathit{data}\ l = \lambda n \to l !! n$, where $!! : [A] \to \mathit{Nat} \to A$ is a function that returns the element at the n-th position of a list. Forming an isomorphism, these functions satisfy the following two equations:

$$\mathit{recover} \circ (\mathit{shape} \vartriangle \mathit{data}) = \mathit{id} \qquad \mathit{recover}\text{-}\textsc{Iso}$$

$$(\mathit{shape} \vartriangle \mathit{data}) \circ \mathit{recover} = \mathit{id} \qquad \mathit{shape}\text{-}\mathit{data}\text{-}\textsc{Iso}$$

In the standard shapely types formulation (Jay, 1995), the shape-data-ISO law is only an inclusion because $\mathit{recover}$ is a partial function that may fail if the number of argument data values does not match the argument shape. We are able to specify it as an equality due to the added precision provided by the dependent types.

Deltas In our work, we model a delta $b \,\Delta\, a$ between a target value b and a source value a as a correspondence relation $P\ b \rightarrow P\ a$ (an arrow in the REL category) from positions in the target value to positions in the source value. Matching lenses (Barbosa et al., 2010) capture the same concept but assume that a shape has a list of positions, while we formulate it in a type-safe manner with a dependent type. We will also distinguish *vertical deltas* that model updates between values of the same type, from *horizontal deltas* that establish correspondences between values of different (view and source) types (Diskin, 2011). In our setting, this correspondence relation must be simple, i.e., each target position has non-ambiguous provenance and is related to at most one source position. In practice, this assumption does not seriously restrict the kind of supported correspondences. For example, when constructing views every view element must necessarily be uniquely related to a source element and when performing an update we can still insert, delete and duplicate elements. The only implication is that elements must be considered atomically, this is, we can not express for example that an element in the view is the combination of two elements in the source.

To describe deltas we will use the point-free relational language from Section 2.2. To preserve simplicity, some combinators will only be used in controlled situations. For example, if a relation R is injective and simple (like get_{Δ} presented further on), then R° is also simple. By resorting to this language, we can reason about deltas using the powerful algebraic laws ruling its combinators (Section A.2).

5.2 Laying Down Delta Lenses

In the delta-based framework of Diskin et al. (2011a), updates are encoded as triples (s, u, s') where s, s' are the source and target values and u is a delta between elements of s and s', and lens transformations are arrows that simultaneously translate states and deltas. In our presentation, we choose to separate the state-based and delta-based components of the lenses. This, together with the dependent type notation, leads to a simpler formulation of delta lenses for polymorphic inductive data types: operationally, the delta-based components required for defining composite delta lenses can be ignored by end users, that are only required to understand the more intuitive interface of the state-based components. Also, the use of a dependent type notation removes redundancies and helps clarifying some subtleties in the delta lens formulation, that are specified in the form of incidence relations by (Diskin et al., 2011a). As a consequence, delta lens transformations are no longer partially defined modulo additional properties entailing preservation of the incidence between values and deltas. We adapt the definition of Diskin et al. (2011a) for our domain of polymorphic inductive data types as follows:

Definition 6 (Delta lens)**.** *A delta lens l, denoted by $l : \mathsf{S}\ A \unrhd_{\blacktriangle} \mathsf{V}\ B$, is a bidirectional transformation that comprises four total functions:*

$$\begin{array}{ll}
\mathit{get} & : \mathsf{S}\ A \to \mathsf{V}\ B \\
\mathit{get}_{\blacktriangle} & : \forall \{s' : \mathsf{S}\ A, s : \mathsf{S}\ A\}.\ s' \,\Delta\, s \to \mathit{get}\ s' \,\Delta\, \mathit{get}\ s \\
\mathit{put} & : \forall (v, s) : \mathsf{V}\ B \times \mathsf{S}\ A.\ v \,\Delta\, \mathit{get}\ s \to \mathsf{S}\ A \\
\mathit{put}_{\blacktriangle} & : \forall \{(v, s) : \mathsf{V}\ B \times \mathsf{S}\ A\}, d : v \,\Delta\, \mathit{get}\ s.\ \mathit{put}\ (v, s)\ d \,\Delta\, s
\end{array}$$

The delta lens is called well-behaved *iff it satisfies the following properties:*

$$\begin{array}{llll}
\mathit{get}\ (\mathit{put}\ (v, s)\ d) = v & \textsc{PutGet} & \mathit{get}_{\blacktriangle}\ (\mathit{put}_{\blacktriangle}\ d) = d & \textsc{PutGet}_{\blacktriangle} \\
\mathit{put}\ (\mathit{get}\ s, s)\ \mathit{id} = s & \textsc{GetPut} & \mathit{put}_{\blacktriangle}\ \mathit{id} = \mathit{id} & \textsc{PutId}_{\blacktriangle}
\end{array}$$

In the above definition, the state-based component of the delta lens is given by the functions get, that computes a view of a source value, and put, that takes a pair containing a modified view and an original source, together with a delta from the modified view to the original view, and returns a new modified source. The delta-based function $\mathit{get}_{\blacktriangle}$ translates a source delta into a delta between views produced by get, and $\mathit{put}_{\blacktriangle}$ receives a view delta and computes a delta from the new source produced by put to the original source. Properties PutGet and GetPut are the traditional state-based ones: view-to-view round-trips preserve view modifications; and put must preserve the original source for identity updates. PutGet$_{\blacktriangle}$ and PutId$_{\blacktriangle}$ denote similar laws on deltas: view-to-view round-trips preserve view updates; and $\mathit{put}_{\blacktriangle}$ must preserve identity updates. It is easy to see that our formulation is equivalent to the well-behaved delta lenses from (Diskin et al., 2011a). For example, their GetId property is a consequence of our axiomatization.

Abstractly, delta lenses are simple to understand since they transform updates (vertical deltas) into updates. However, to propagate view updates, $\mathit{put}_{\blacktriangle}$ must somehow recover a horizontal delta between the original view and the original source that provides the required traceability information to calculate a new source update (Hidaka et al., 2010).

From an implementation perspective, an alternative formulation of delta lenses that compute and process these horizontal deltas explicitly is preferable (instead of, for instance, having to infer them at run-time for specific executions as conducted in (Voigtländer, 2009)). As such, we propose an alternative framework of *horizontal delta lenses*, whose delta-based functions explicitly return the horizontal deltas induced by the state-based transformations. Moreover, it is convenient to include in this less abstract framework a $create$ function (Bohannon et al., 2008) that reconstructs a default source value from a view value for situations where the original source is not available.

Definition 7 (Horizontal delta lens)**.** *A horizontal delta lens l, denoted by $l : \mathsf{S}\ A \trianglerighteq_{\Delta} \mathsf{V}\ B$, comprises three total functions $get : \mathsf{S}\ A \to \mathsf{V}\ B$, $put : \forall (v, s) : \mathsf{V}\ B \times \mathsf{S}\ A.\ v \,\Delta\, get\ s \to \mathsf{S}\ A$, and $create : \mathsf{V}\ B \to \mathsf{S}\ A$, plus three horizontal deltas:*

$$\begin{aligned}
get_{\Delta} &: \forall \{ s : \mathsf{S}\ A \}.\ get\ s \,\Delta\, s \\
put_{\Delta} &: \forall \{ (v, s) : \mathsf{V}\ B \times \mathsf{S}\ A \}, d : v \,\Delta\, get\ s.\ put\ (v, s)\ d \,\Delta\, (v, s) \\
create_{\Delta} &: \forall \{ v : \mathsf{V}\ B \}.\ create\ v \,\Delta\, v
\end{aligned}$$

It is called well-behaved *iff it satisfies* PUTGET*,* GETPUT*, a third state-based law*

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

and the following delta-based properties:

$$\begin{aligned}
create_{\Delta} \circ get_{\Delta} &= id && \textsc{CreateGet}_{\Delta} \\
put_{\Delta}\ d \circ get_{\Delta} &= i_1 && \textsc{PutGet}_{\Delta} \\
(get_{\Delta} \triangledown id) \circ put_{\Delta}\ id &= id && \textsc{GetPut}_{\Delta}
\end{aligned}$$

The horizontal deltas are complements of the state-based functions and explicitly record the traceability of their execution: get_{Δ} denotes a delta from the original view to the original source and vice-versa for $create_{\Delta}$, while put_{Δ} is a delta from the new source to the input view-source pair. In practice, this duality will allow us to derive by construction the deltas of most of our lens combinators by reversing their behaviors on states[3]. The delta-based laws also dualize the state-based laws, with

[3] Since put is a dependent function, it first receives a view-source pair, with which the argument

the insight that the type of positions of a view-source pair is the disjoint sum of the positions in the view and in the source. For example, while the CREATEGET law states that abstracting a created source shall yield the original view, the CREATEGET$_\triangle$ law evidences that the corresponding delta on views shall also preserve all view elements (identity). The PUTGET$_\triangle$ law states the same for put: abstracting a source generated by put shall preserve all (left) view elements in the original view-source pair (hence the i_1 delta). GETPUT$_\triangle$ entails that abstracting a source and immediately putting it back (taking an identity delta on views) yields an identity delta on sources.

From these delta-based laws, we can also derive that: $get_\triangle$ is a total injective function that maps each position in the view to a distinct position in the source (since PUTGET$_\triangle$ and CREATEGET$_\triangle$ entail that $get_\triangle$ is a total and injective relation and GETPUT$_\triangle$ entails that $get_\triangle$ is simple); $create_\triangle$ is a partial surjective function that maps some positions in the source to all positions in the view (as entailed by CREATEGET$_\triangle$); and $put_\triangle\ d$ is a partial function such that $i_1{}^\circ \circ put_\triangle\ d$ is surjective (from PUTGET$_\triangle$), meaning that it relates all view positions to new source positions.

We now show how horizontal delta lenses can be used to implement the more abstract framework of delta lenses:

Definition 8. *A horizontal delta lens* $l : \mathsf{S}\ A \unrhd_\triangle \mathsf{V}\ B$ *can be lifted to a delta lens* $l_\blacktriangle : \mathsf{S}\ A \unrhd_\blacktriangle \mathsf{V}\ B$ *with the same state-based* get *and* put *functions and the delta-based transformations* $get_\blacktriangle\ d = get_\triangle^\circ \circ d \circ get_\triangle$ *and* $put_\blacktriangle\ d = (get_\triangle \circ d \triangledown id) \circ put_\triangle\ d$.

This definition is illustrated in Figure 5.3. Normal arrows denote state-based functions and dotted arrows denote deltas between the types of positions of particular values. Given a source update d_S and a view update d_V, the $get_\blacktriangle\ d_S$ (Figure 5.3a) and $put_\blacktriangle\ d_V$ (Figure 5.3b) deltas of the resulting delta lens can be calculated by composing the dotted arrows.

Theorem 8. *If a horizontal delta lens* $l : \mathsf{S}\ A \unrhd_\triangle \mathsf{V}\ B$ *is well-behaved, then the delta lens* $l_\blacktriangle$ *is well-behaved.*

Proof. The state-based laws dismiss proof obligations. The PUTGET$_\blacktriangle$ law is proven using PUTGET$_\triangle$ and by knowing that $get_\triangle^\circ \circ get_\triangle = id$, since $get_\triangle$ is a total and

delta must be consistent, and so we write expressions of the form $put\ (v, s)\ d_V$. Because this means that we must know the concrete view and source values before specifying the delta, in this chapter we will present the state-based functions of our horizontal delta lens combinators in the point-wise style. Anyway, to emphasize the duality between the (point-wise) state-based and the (point-free) delta-based functions, in diagrams we will sometimes abuse the notation and write $put\ d_V\ (v, s)$, so that $put\ d_V$ can be composed in the point-free style.

$$\begin{tikzcd}
\mathsf{S}\ A \arrow[r, shift right, "get"'] & \mathsf{V}\ B \arrow[l, dashed, shift right, "get_{\triangle}"'] \\
\mathsf{S}\ A \arrow[u, dashed, "d_S"] \arrow[r, shift left, "get"] & \mathsf{V}\ B \arrow[u, dashed, "get_{\blacktriangle}\ d_S"'] \arrow[l, dashed, shift left, "get_{\triangle}"]
\end{tikzcd}$$

(a) Forward delta-level transformation

$$\begin{tikzcd}
\mathsf{S}\ A \arrow[rr, shift right, "get"'] \arrow[dr, dashed, shift right, "i_2"'] & & \mathsf{V}\ B \arrow[ll, dashed, shift right, "get_{\triangle}"'] \\
\mathsf{S}\ A \arrow[u, dashed, "put_{\blacktriangle}\ d_V"] \arrow[r, dashed, shift right, "put_{\triangle}\ d_V"'] & \mathsf{V}\ B \times \mathsf{S}\ A \arrow[l, shift right, "put\ d_V"'] \arrow[ul, shift right, "\pi_2"'] \arrow[r, shift left, "\pi_1"] & \mathsf{V}\ B \arrow[u, dashed, "d_V"'] \arrow[l, dashed, shift left, "i_1"]
\end{tikzcd}$$

(b) Backward delta-level transformation

Figure 5.3: Construction of a delta lens from a horizontal delta lens.

injective relation. The PUTID$_{\blacktriangle}$ law follows directly from GETPUT$_{\triangle}$. The formal proofs are written in Section B.3.1. □

A horizontal delta lens $l : \mathsf{S}\ A \unrhd_{\blacktriangle} \mathsf{V}\ B$ can be embedded into a state-based lens $\lfloor l \rfloor_{\mathit{diff}} : \mathsf{S}\ A \unrhd \mathsf{V}\ B$ that receives a differencing function diff, estimating a delta from the pre- and post-states of a view update, but forgets shape and alignment for further compositions:

$$\begin{array}{l}
\forall (\mathit{diff} : \forall v' : \mathsf{V}\ B, v : \mathsf{V}\ B.\ v' \,\Delta\, v), l : \mathsf{S}_A \unrhd_{\triangle} \mathsf{V}_B.\ \lfloor l \rfloor_{\mathit{diff}} : \mathsf{S}\ A \unrhd \mathsf{V}\ B \\
\mathit{get} \qquad\quad = \mathit{get}_l \\
\mathit{create} \quad\; = \mathit{create}_l \\
\mathit{put}\ (v, s) = \mathit{put}_l\ d_V\ (v, s) \\
\quad \textbf{where}\ d_V = \mathit{diff}\ (v, \mathit{get}\ s)
\end{array}$$

As long as the resulting lens is state-based, this combinator ignores the delta-based functions. Since put_l demands a delta between the modified view and the original view, embedding uses the additional differencing function to estimate such delta. Similar constructions have already been studied for delta lenses by Diskin et al. (2011a) and put to practice in matching lenses by Barbosa et al. (2010). The lens $\lfloor l \rfloor_{\mathit{diff}}$ is well-behaved if diff is well-behaved, i.e., if it returns the identity update given the same states, according to the following law:

$$\mathit{diff}\ (v, v) = \mathit{id} \qquad \text{DIFF-ID}$$

5.3 Combinators for Horizontal Delta Lenses

We have developed a suitable framework for implementing delta lenses over inductive types. In this section, we introduce some primitive horizontal delta lens combinators for lifting state-based lenses into delta lenses, supporting in particular mapping and reshaping transformations, and define liftings of our point-free lens combinators from Chapter 4 to horizontal delta lens combinators that can be used to define more complex transformations in a compositional way.

5.3.1 Primitive Combinators

Lifting There are two trivial ways to lift a regular state-based lens into an horizontal delta lens: the constant lifting $\underline{l}$ applies the lens l to constant shapes with no elements (note that the types of elements can be arbitrary, since they are never populated); and the identity lifting $\mathsf{Id}\ l$ applies the lens l to single elements with identical shapes. These two combinators have exactly the same behavior on states, but process deltas differently:

$$\begin{array}{llll}
\multicolumn{4}{c}{\forall l : S \trianglerighteq V.\ \underline{l} : \underline{S}_A \trianglerighteq_\Delta \underline{V}_B} \\
\mathit{get}\ s & = \mathit{get}_l\ s & \mathit{get}_\Delta & = \bot \\
\mathit{put}\ (v,s)\ d & = \mathit{put}_l\ (v,s) & \mathit{put}_\Delta\ d & = \bot \\
\mathit{create}\ v & = \mathit{create}_l\ v & \mathit{create}_\Delta & = \bot
\end{array}
\qquad
\begin{array}{llll}
\multicolumn{4}{c}{\forall l : A \trianglerighteq B.\ \mathsf{Id}\ l : \mathsf{Id}_A \trianglerighteq_\Delta \mathsf{Id}_B} \\
\mathit{get}\ a & = \mathit{get}_l\ a & \mathit{get}_\Delta & = \mathit{id} \\
\mathit{put}\ (b,a)\ d & = \mathit{put}_l\ (b,a) & \mathit{put}_\Delta\ d & = i_1 \\
\mathit{create}\ b & = \mathit{create}_l\ b & \mathit{create}_\Delta & = \mathit{id}
\end{array}$$

Mapping The above primitive combinators can be generalized to more complex shapes that may contain multiple elements. For example, identity lifting can be generalized to a mapping horizontal delta lens (for an arbitrary functor T) as follows:

$$\begin{array}{llll}
\multicolumn{4}{c}{\forall l : A \trianglerighteq B.\ \mathsf{T}\ l : \mathsf{T}_A \trianglerighteq_\Delta \mathsf{T}_B} \\
\mathit{get}\ s & = \mathsf{T}\ \mathit{get}_l\ s & \mathit{get}_\Delta & = \mathit{id} \\
\mathit{put}\ (v,s)\ d & = \mathit{recover}\ (\mathit{shape}\ v, \mathit{dput}\ \cup\ \mathit{dcreate}) & \mathit{put}_\Delta\ d & = i_1 \\
\quad \mathbf{where}\ \mathit{dput} & = \mathit{put}_l \circ (\mathit{data}\ v \triangle \mathit{data}\ s \circ d) & & \\
\qquad\quad \mathit{dcreate} & = \mathit{create}_l \circ \mathit{data}\ v \circ (\mathit{id} - \delta\,\mathit{dput}) & & \\
\mathit{create}\ v & = \mathsf{T}\ \mathit{create}_l\ v & \mathit{create}_\Delta & = \mathit{id}
\end{array}$$

Like the state-based functor mapping lens from Chapter 4, the *get* and *create* functions simply map the components of the basic lens over the data elements, producing trivial deltas (all positions are preserved). Instead of aligning elements by their positions (as done by $\mathit{fzip}_{\mathsf{T}}\ \mathit{create}_l : \mathsf{T}\ B \times \mathsf{T}\ A \rightarrow \mathsf{T}\ (A \times B)$ for state-based functor mapping), put now performs global data alignment based on the view update delta: for each view element v_e, if it relates to a source element s_e, $\mathit{put}\ (v_e, s_e)$ is applied; otherwise, a default source is generated with $\mathit{create}_l\ v_e$. In the definition of put, $\mathit{dput} \cup \mathit{dcreate}$ builds a (total) function from view positions to source elements as a correspondence relation $P\ v \rightarrow A$, to pass as an argument to $\mathit{recover}$. The relation dput matches view elements with existing source elements, and $\mathit{dcreate}$ creates fresh source elements for the remaining unmatched view elements. Since dput and $\mathit{dcreate}$ are two simple relations with disjoint domains, as guaranteed by the filter $(\mathit{id} - \delta\,\mathit{dput})$, their relational union is simple. Their union is also entire, and thus a total function in SET. The put_Δ delta is trivial, since all elements in the new source come from elements in the view.

Mapping defines a functor on horizontal delta lenses, preserving identity and composition laws (composition of horizontal delta lenses is defined below):

$$\mathsf{T}\ \mathit{id} = \mathit{id} \qquad \text{MAP-FUNCTOR-ID}$$

$$\mathsf{T}\ f \circ \mathsf{T}\ g = \mathsf{T}\ (f \circ g) \qquad \text{MAP-FUNCTOR-COMP}$$

Reshaping Given a natural lens that only transforms shapes, we can lift it to a reshaping horizontal delta lens using the following combinator:

$$\forall \eta : \mathsf{S} \mathrel{\dot{\trianglerighteq}} \mathsf{V}.\ \overleftrightarrow{\eta} : \mathsf{S} \mathrel{\dot{\trianglerighteq}_\Delta} \mathsf{V}$$

$$\begin{array}{llll}
\mathit{get}\ s & = \mathit{get}_\eta\ s & \mathit{get}_\Delta\ \{s\} & = \overleftarrow{\mathit{get}_\eta}\ s \\
\mathit{put}\ (v, s)\ d & = \mathit{put}_\eta\ (v, s) & \mathit{put}_\Delta\ \{(v, s)\}\ d & = \overleftarrow{\mathit{put}_\eta}\ (v, s) \\
\mathit{create}\ v & = \mathit{create}_\eta\ v & \mathit{create}_\Delta\ \{v\} & = \overleftarrow{\mathit{create}_\eta}\ v
\end{array}$$

Although this combinator permits defining delta lenses that transform the shape of the source, it just infers suitable horizontal deltas for an existing state-based lens. Therefore, the state-based components of the horizontal delta lens are determined by the value-level functions of the argument lens. The horizontal deltas are calculated using a semantic approach inspired by (Voigtländer, 2009), by running the value-level functions against sources with the data elements replaced by the respective positions, thus inferring the correspondences in the target. This is performed by the auxiliary function $\overleftarrow{\cdot}$:

$$\forall \eta : \mathsf{F} \mathrel{\dot{\rightarrow}} \mathsf{G}.\ \overleftarrow{\eta} : \forall s : \mathsf{F}\ A.\ \eta\ s\ \Delta\ s$$

$$\overleftarrow{\eta} = data \circ \eta \circ recover \circ (shape \, \triangle (\underline{id} \circ \, ! \,))$$

The behavior of $\overleftarrow{\cdot}$ is illustrated in the following diagram:

$$\begin{tikzcd}
s : \mathsf{F}\ A \arrow[rr, "\overleftarrow{\eta}"] \arrow[d, "shape \, \triangle (\underline{id} \circ \, ! \,)"'] & & P\ (\eta\ s) \to P\ s \\
\mathsf{F}\ 1 \times (P\ s \to P\ s) \arrow[r, "recover"'] & \mathsf{F}\ (P\ s) \arrow[r, "\eta"'] & \mathsf{G}\ (P\ s) \arrow[u, "data"']
\end{tikzcd}$$

Note that the type of positions for $s : \mathsf{F}\ A$ and $shape\ \ s : \mathsf{F}\ 1$, and for $\eta\ \ s : \mathsf{G}\ A$ and the value of type $\mathsf{G}\ (P\ \ s)$ computed in the diagram, are the same. Another relevant observation is that $data$ returns a total function, and thus the delta produced by $\overleftarrow{\cdot}$ is always an entire and simple correspondence relation.

Many useful examples of these natural horizontal delta lens transformations are polymorphic versions of the usual isomorphisms handling the associativity and commutativity of sums and products, such as $swap : (\mathsf{F} \otimes \mathsf{G})_A \trianglerighteq_{\Delta} (\mathsf{G} \otimes \mathsf{F})_A$. Another primitive combinator that falls under this category is the identity horizontal delta lens $id : \mathsf{F}_A \trianglerighteq_{\Delta} \mathsf{F}_A$. Nevertheless, this combinator is only interesting to lift state-based lenses for which there is no ambiguity in view-update translation and whose alignment behavior cannot be improved by taking into account delta information, as is the case of isomorphisms. Since the behavior of the lifted delta lenses is completely determined by the argument state-based lens, using this combinator to define the $lspine$ and $filter\_right$ examples from the beginning of this chapter (which are indeed natural lenses) as horizontal delta lenses would not perform proper alignment.

Our reshaping combinator defines a functor on horizontal delta lenses, and is also natural transformation on horizontal delta lenses:

$$\begin{array}{lr}
\overleftrightarrow{id} = id & \overleftrightarrow{\cdot}\text{-FUNCTOR-ID} \\
\overleftrightarrow{(f \circ g)} = \overleftrightarrow{f} \circ \overleftrightarrow{g} & \overleftrightarrow{\cdot}\text{-FUNCTOR-COMP} \\
\mathsf{G}\ g \circ \overleftrightarrow{f} = \overleftrightarrow{f} \circ \mathsf{F}\ g & \overleftrightarrow{\cdot}\text{-NAT}
\end{array}$$

The last law opens the door to an interesting optimization, since in a composition involving only natural transformations and mappings, they can be grouped together into sequences of natural transformations and mappings and fused with $\overleftrightarrow{\cdot}$-FUNCTOR-COMP and MAP-FUNCTOR-COMP.

We can also define a more liberal reshaping combinator inspired by the mixed semantic approach from (Voigtländer et al., 2010) that, unlike our reshaping combinator that requires a natural lens transformation, takes as input a natural transformation for the forward transformation and a lens on shapes. As defined in (Voigtländer et al., 2010), the put of this combinator can then use the argument lens to calculate the shape of the source lists, and calculate the source positions for the view elements through the polymorphic interpretation of the natural transformation. Since the motivation for this combinator is different from the goal of this section, we refrain from providing a concrete definition.

5.3.2 Point-free Combinators

We now show that many of the point-free combinators from Chapter 4 can also be lifted to horizontal delta lenses.

Composition Two fundamental point-free combinators are identity and composition. Identity $id : \mathsf{S}_A \to \mathsf{S}_A$ can be trivially defined. Composition can be lifted to horizontal delta lenses as follows:

$$\forall f : \mathsf{V}_B \unrhd_\Delta \mathsf{U}_C, g : \mathsf{S}_A \unrhd_\Delta \mathsf{V}_B.\ (f \circ g) : \mathsf{S}_A \unrhd_\Delta \mathsf{U}_C$$

$$\begin{array}{llll}
get\ s & = get_f\ (get_g\ s) & get_\Delta & = get_{\Delta_g} \circ get_{\Delta_f} \\
put\ (u,s)\ d_U & = put_g\ (u,s)\ d_V & put_\Delta\ d_U & = (dv' \nabla i_2) \circ put_{\Delta_g}\ d_V \\
\quad \textbf{where}\ v' & = put_f\ (v, get_g\ s)\ d_U & \quad \textbf{where}\ dv' & = (id + get_{\Delta_g}) \circ put_{\Delta_f}\ d_U \\
\qquad\qquad d_V & = (get_{\Delta_f} \circ d_U \nabla id) & \qquad\qquad d_V & = (get_{\Delta_f} \circ d_U \nabla id) \\
 & \quad \circ\ put_{\Delta_f}\ d_U & & \quad \circ\ put_{\Delta_f}\ d_U \\
create\ u & = create_g\ (create_f\ u) & create_\Delta & = create_{\Delta_f} \circ create_{\Delta_g}
\end{array}$$

The behavior of our composition combinator is illustrated in Figure 5.4. In the get direction, we simply compose the respective functions of the lenses f and g, and the get_Δ delta is calculated by composing the argument deltas in the reverse order. The $create$ direction is dual. In the put direction, the intermediate delta d_V passed to put_g maps elements in the new value of type $\mathsf{V}\ \ B$ (computed with $put_f\ (u, get_g\ s)\ d_U$) to elements in the original view of type $\mathsf{V}\ \ B$ (computed with $get_g\ s$). This delta corresponds to the $put_\blacktriangle\ (u, get_g\ s)\ d_U$ delta from Definition 8 and can be calculated by composing the dotted arrows from Figure 5.4a. In the point-free style, $put_{f \circ g}$ would

$$\begin{tikzcd}
\mathsf{S}_A \arrow[rr, "get_g"'] & & \mathsf{V}_B \arrow[ll, dashed, bend right, "get_{\Delta g}"'] \arrow[rr, "get_f"'] \arrow[dr, dashed, "i_2"'] & & \mathsf{U}_C \arrow[ll, dashed, bend right, "get_{\Delta f}"'] \\
\mathsf{S}_A \arrow[u, dashed] \arrow[r, dashed, bend right, "put_{\Delta g}\ d_V"'] & \mathsf{V}_B \times \mathsf{S}_A \arrow[l, "put_g\ d_V"'] \arrow[ul, "\pi_2"'] \arrow[r, "\pi_1"] & \mathsf{V}_B \arrow[u, dashed, "d_V"] \arrow[l, dashed, bend left, "i_1"] \arrow[r, dashed, bend right, "put_{\Delta f}\ d_U"'] & \mathsf{U}_C \times \mathsf{V}_B \arrow[l, "put_f\ d_U"'] \arrow[ul, "\pi_2"'] \arrow[r, "\pi_1"] & \mathsf{U}_C \arrow[u, dashed, "d_U"'] \arrow[l, dashed, bend left, "i_1"]
\end{tikzcd}$$

Also the dashed arrow $\mathsf{S}_A \to \mathsf{V}_B \times \mathsf{S}_A$ labelled i_2:

$$\begin{tikzcd}
\mathsf{S}_A \arrow[dr, dashed, "i_2"'] & \\
& \mathsf{V}_B \times \mathsf{S}_A
\end{tikzcd}$$

(a) Delta-level transformations.

$$\begin{tikzcd}
\mathsf{U}_C \times \mathsf{S}_A \arrow[d, shift left, "(id \times get_g) \vartriangle \pi_2"] \\
(\mathsf{U}_C \times \mathsf{V}_B) \times \mathsf{S}_A \arrow[u, dashed, shift left, "(id + get_{\Delta g}) \triangledown i_2"] \arrow[d, shift left, "put_f\ d_U \times id"] \\
\mathsf{V}_B \times \mathsf{S}_A \arrow[u, dashed, shift left, "put_{\Delta f}\ d_U + id"] \arrow[d, shift left, "put_g\ d_V"] \\
\mathsf{S}_A \arrow[u, dashed, shift left, "put_{\Delta g}\ d_V"]
\end{tikzcd}$$

(b) Functions put and put_Δ.

Figure 5.4: Composition of horizontal delta lenses.

be defined as shown in Figure 5.4b, that essentially corresponds to the state-based definition from Chapter 4. As long as the type of positions of a pair value is a sum, we can elegantly calculate the delta-based arrows and the composite delta $put_{\Delta f \circ g}$ delta by dualizing the state-based point-free combinators from products to sums.

The following laws witness the existence of a category of horizontal delta lenses, whose objects are polymorphic inductive types and arrows are horizontal delta lenses:

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \circ\text{-Assoc}$$

$$id \circ f = f = f \circ id \qquad id\text{-Nat}$$

A more liberal kind of forgetful composition $\lfloor f \rfloor_{\mathit{diff}_1} \circ \lfloor g \rfloor_{\mathit{diff}_2}$ is also possible to define, by first converting the horizontal delta lenses f and g into normal lenses, for cases when they do not match on their intermediate shapes. This mechanism is used in (Barbosa et al., 2010) for the specification of nested matching lenses, but is deemed ill-formed in (Diskin et al., 2011a) since the resulting lenses may identify and align updates differently. Also, as long as the behavior of the resulting state-based lenses depends on the differencing functions, this alignment mismatch "pollutes" the algebraic laws with additional side conditions. For example, in order to fuse two mapping horizontal delta lenses $\mathsf{T}\ f : \mathsf{T}_B \unrhd_\Delta \mathsf{T}_C$ and $\mathsf{T}\ g : \mathsf{T}_A \unrhd_\Delta \mathsf{T}_B$ into a single mapping, we would need to prove that they agree on their differencing functions:

$$\begin{array}{c} \lfloor \mathsf{T}\ f \rfloor_{\mathit{diff}_1} \circ \lfloor \mathsf{T}\ g \rfloor_{\mathit{diff}_2} = \lfloor \mathsf{T}\ (f \circ g) \rfloor_{\mathit{diff}_1} \\ \Leftarrow \\ \mathit{diff}_2 = \mathit{diff}_1 \circ (\mathit{get}_{\mathsf{T}\ g} \mathbin{\triangle} \mathit{get}_{\mathsf{T}\ g}) \end{array} \qquad \textsc{Map-Diff-Comp}$$

Products The product projections can be lifted to horizontal delta lenses as follows:

$$\begin{array}{lll} \multicolumn{3}{c}{\forall f : \mathsf{F}_A \to \mathsf{G}_A.\ {\pi_1}^f : (\mathsf{F} \otimes \mathsf{G})_A \unrhd_{\triangle} \mathsf{F}_A} \\ \mathit{get}\ (x,y) = x & \mathit{get}_{\triangle} = i_1 \\ \mathit{put}\ (z,(x,y))\ d = (z,y) & \mathit{put}_{\triangle}\ d = (i_1 \triangledown i_2 \circ i_2) \\ \mathit{create}\ z = (z, f\ z) & \mathit{create}_{\triangle} = {i_1}^{\circ} \end{array}$$

$$\begin{array}{lll} \multicolumn{3}{c}{\forall f : \mathsf{G}_A \to \mathsf{F}_A.\ {\pi_2}^f : (\mathsf{F} \otimes \mathsf{G})_A \unrhd_{\triangle} \mathsf{G}_A} \\ \mathit{get}\ (x,y) = y & \mathit{get}_{\triangle} = i_2 \\ \mathit{put}\ (z,(x,y))\ d = (x,z) & \mathit{put}_{\triangle}\ d = (i_2 \circ i_1 \triangledown i_1) \\ \mathit{create}\ z = (f\ z, z) & \mathit{create}_{\triangle} = {i_2}^{\circ} \end{array}$$

These lifted combinators are sort of natural transformations that project away specific components of the shape of a pair. Although the forward transformation get is a natural transformation, the projection lenses are not natural, because the default parameter f may be defined for a concrete type A, making $create$ non-natural.

Nevertheless, they enjoy kind of naturality laws modulo product and modulo mapping (that is defined as a primitive delta lens and not polytypically like the state-based functor mapping lens), like the state-based lens projections:

$$\begin{array}{lr} {\pi_1}^h \circ (f \times g) = f \circ {\pi_1}^{\mathit{create}_g \circ h \circ \mathit{get}_f} & \pi_1\text{-}\textsc{Nat} \\ {\pi_2}^h \circ (f \times g) = g \circ {\pi_2}^{\mathit{create}_f \circ h \circ \mathit{get}_g} & \pi_2\text{-}\textsc{Nat} \\ {\pi_1}^h \circ (\mathsf{F} \otimes \mathsf{G})\ f = \mathsf{F}\ f \circ {\pi_1}^{\mathsf{F}\ \mathit{create}_f \circ h \circ \mathsf{F}\ \mathit{get}_f} & \pi_1\text{-}\textsc{Map} \\ {\pi_2}^h \circ (\mathsf{F} \otimes \mathsf{G})\ f = \mathsf{G}\ f \circ {\pi_2}^{\mathsf{G}\ \mathit{create}_f \circ h \circ \mathsf{G}\ \mathit{get}_f} & \pi_2\text{-}\textsc{Map} \end{array}$$

The lifted product bifunctor applies two horizontal delta lenses in parallel and is defined as follows:

$$\forall f : \mathsf{F}_A \trianglerighteq_{\Delta} \mathsf{H}_B, g : \mathsf{G}_A \trianglerighteq_{\Delta} \mathsf{I}_B.\, f \times g : (\mathsf{F} \otimes \mathsf{G})_A \trianglerighteq_{\Delta} (\mathsf{H} \otimes \mathsf{I})_B$$

$$\begin{array}{ll}
\mathit{get}\ (x, y) = (\mathit{get}_f\ x, \mathit{get}_g\ y) & \mathit{get}_{\Delta} = \mathit{get}_{\Delta f} + \mathit{get}_{\Delta g} \\
\mathit{put}\ ((z, w), (x, y))\ d = (x', y') & \mathit{put}_{\Delta}\ d = \mathit{dists} \circ (dx' + dy') \\
\quad \mathbf{where}\ x' = \mathit{put}_f\ (z, x)\ ({i_1}^{\circ} \circ d \circ i_1) & \quad \mathbf{where}\ dx' = \mathit{put}_{\Delta f}\ ({i_1}^{\circ} \circ d \circ i_1) \\
\qquad\qquad y' = \mathit{put}_g\ (w, y)\ ({i_2}^{\circ} \circ d \circ i_2) & \qquad\qquad dy' = \mathit{put}_{\Delta g}\ ({i_2}^{\circ} \circ d \circ i_2) \\
\mathit{create}\ (z, w) = (\mathit{create}_f\ z, \mathit{create}_g\ w) & \mathit{create}_{\Delta} = \mathit{create}_{\Delta f} + \mathit{create}_{\Delta g}
\end{array}$$

When computing put, the product combinator splits the delta over the view pair in two deltas mapping only left or only right elements, to be passed to put_f and put_g, respectively. By halving the deltas, the puts of the argument lenses will loose the delta correspondences for view elements that were swapped to a different side of the view pair. For example, the delta lens $\pi_1 \times \pi_1 : ((\mathsf{F} \otimes \mathsf{G}) \otimes (\mathsf{F} \otimes \mathsf{G}))\ A \trianglerighteq_{\Delta} (\mathsf{F} \otimes \mathsf{F})\ A$ would only be able to restore left/right information for left/right elements. Given the polymorphic nature of this combinator, which is agnostic to the concrete instantiations of the functors F and G, this is the only reasonable behavior. A more refined behavior, involving non-trivial fitting of the right data elements of y that are related to the left view z into the original left view x, to be restored by put_f, would only be possible for very specific functor instantiations.

The product horizontal delta lens is a bifunctor in the category of horizontal delta lenses, preserving identity and composition laws:

$$\mathit{id} \times \mathit{id} = \mathit{id} \qquad \times\text{-FUNCTOR-ID}$$

$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-FUNCTOR-COMP}$$

Sums The either combinator can be lifted to a horizontal delta lens as follows:

$$\forall p : \mathsf{H}_B \to 2, f : \mathsf{F}_A \trianglerighteq \mathsf{H}_B, g : \mathsf{G}_A \trianglerighteq \mathsf{H}_B.\, f \triangledown g^p : (\mathsf{F} \oplus \mathsf{G})_A \trianglerighteq \mathsf{H}_B$$

$$\begin{array}{ll}
\mathit{get}\ (L\ x) = \mathit{get}_f\ x & \mathit{get}_{\Delta}\ \{(L\ x)\} = \mathit{get}_{\Delta f}\ \{x\} \\
\mathit{get}\ (R\ y) = \mathit{get}_g\ y & \mathit{get}_{\Delta}\ \{(R\ y)\} = \mathit{get}_{\Delta g}\ \{y\} \\
\mathit{put}\ (z, L\ x)\ d = \mathit{put}_f\ (z, x)\ d & \mathit{put}_{\Delta}\ \{(z, L\ x)\}\ d = \mathit{put}_{\Delta f}\ \{(z, x)\}\ d \\
\mathit{put}\ (z, R\ y)\ d = \mathit{put}_g\ (z, y)\ d & \mathit{put}_{\Delta}\ \{(z, R\ y)\}\ d = \mathit{put}_{\Delta g}\ \{(z, y)\}\ d \\
\multicolumn{2}{l}{\mathit{create}\ z = \mathbf{if}\ p\ z\ \mathbf{then}\ \mathit{create}_f\ z\ \mathbf{else}\ \mathit{create}_g\ z} \\
\multicolumn{2}{l}{\mathit{create}_{\Delta}\ \{z\} = \mathbf{if}\ p\ z\ \mathbf{then}\ \mathit{create}_{\Delta f}\ \{z\}\ \mathbf{else}\ \mathit{create}_{\Delta g}\ \{z\}}
\end{array}$$

In the specification of the horizontal deltas, the implicit parameters must be known to disambiguate which side of the source sum was consumed by the forward transformation. For create, we use a conditional to test whether to generate left or right source values.

The lifted sum bifunctor applies two horizontal delta lenses to both sides of a sums and is defined as follows:

$$\forall f : \mathsf{F}_A \trianglerighteq \mathsf{H}_B, g : \mathsf{G}_A \trianglerighteq \mathsf{I}_B .\ f + g : (\mathsf{F} \oplus \mathsf{G})_A \trianglerighteq (\mathsf{H} \oplus \mathsf{I})_B$$

$$\begin{array}{llll}
\mathit{get}\ (L\ x) & = L\ (\mathit{get}_f\ x) & \mathit{get}_\triangle\ \{(L\ x)\} & = \mathit{get}_{\triangle f}\ \{x\} \\
\mathit{get}\ (R\ y) & = R\ (\mathit{get}_g\ y) & \mathit{get}_\triangle\ \{(R\ y)\} & = \mathit{get}_{\triangle g}\ \{y\} \\
\mathit{put}\ (L\ z, L\ x)\ d & = L\ (\mathit{put}_f\ (z, x)) & \mathit{put}_\triangle\ \{(L\ z, L\ x)\}\ d & = \mathit{put}_{\triangle f}\ \{(z, x)\}\ d \\
\mathit{put}\ (L\ z, R\ y)\ d & = L\ (\mathit{create}_f\ z) & \mathit{put}_\triangle\ \{(L\ z, R\ y)\}\ d & = i_1 \circ \mathit{create}_{\triangle f}\ \{z\} \\
\mathit{put}\ (R\ w, L\ x)\ d & = R\ (\mathit{create}_g\ w) & \mathit{put}_\triangle\ \{(R\ w, L\ x)\}\ d & = i_1 \circ \mathit{create}_{\triangle g}\ \{w\} \\
\mathit{put}\ (R\ w, R\ y)\ d & = R\ (\mathit{put}_g\ (w, y)) & \mathit{put}_\triangle\ \{(R\ w, R\ y)\}\ d & = \mathit{put}_{\triangle g}\ \{(w, y)\}\ d \\
\mathit{create}\ (L\ z) & = L\ (\mathit{create}_f\ z) & \mathit{create}_\triangle\ \{(L\ z)\} & = \mathit{create}_{\triangle f}\ \{z\} \\
\mathit{create}\ (L\ w) & = R\ (\mathit{create}_g\ w) & \mathit{create}_\triangle\ \{(R\ w)\} & = \mathit{create}_{\triangle g}\ \{w\}
\end{array}$$

Like the other lifted point-free combinators, the state-based functions mimic those of the sum combinator from Chapter 4. The delta-based functions are implicitly parameterized on the view and source sums and return the corresponding delta-based functions of the argument horizontal delta lenses.

Similarly to its state-based homologue, the either combinator satisfies fusion and absorption laws:

$$f \circ (g \nabla h)^p = (f \circ g \nabla f \circ h)^{p \circ \mathit{create}_f} \qquad \text{+-FUSION}$$

$$(f \nabla g)^p \circ (h + i) = (f \circ h \nabla g \circ h)^p \qquad \text{+-ABSOR}$$

The sum combinator is also a bifunctor in the category of horizontal delta lenses:

$$\mathit{id} + \mathit{id} = \mathit{id} \qquad \text{+-FUNCTOR-ID}$$

$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad \text{+-FUNCTOR-COMP}$$

Bang Similarly to the projection lenses, it is also possible to define a $!$ combinator that erases the shape of the source by returning the unit type lifted as a constant functor $\underline{1}$, and replaces each source with an empty value as follows:

$$\forall f : \underline{1}_A \rightarrow \mathsf{F}_A .\ !^f : \mathsf{F}_A \trianglerighteq_\triangle \underline{1}_A$$

$$\begin{array}{llll}
\mathit{get}\ x & = 1 & \mathit{get}_\triangle & = \bot \\
\mathit{put}\ (1, x)\ d & = x & \mathit{put}_\triangle\ d & = i_1 \\
\mathit{create}\ 1 & = f\ 1 & \mathit{create}_\triangle & = \bot
\end{array}$$

For $create$, the $!^{f}$ horizontal delta lens applies the argument function f to reconstruct a source value. We can also rephrase the state-based lens version of the bang uniqueness law to horizontal delta lenses:

$$f = \,!^{\,create_f} \;\Leftrightarrow\; f : \mathsf{F}_A \unrhd_{\triangle} \underline{1}_A \qquad\qquad !\text{-UNIQ}$$

5.4 Recursion Patterns as Horizontal Delta Lenses

Although useful for a combinatorial language, the previous horizontal delta lens combinators only propagate deltas over rigid shapes (in the sense that they process shapes polymorphically without further detail) and do not perform any sort of shape alignment. For mappings, updates may change the cardinality of the data (a container structure such as a list may increase or decrease in length), but alignment can be reduced to the special case of data alignment, with the shape of the update being copied to the result. This problem becomes more general whenever lenses are allowed to restructure the types, in particular recursive ones whose values have a more elastic shape: by changing the number of recursive steps, an update can alter the shape of the view (and thus the number of placeholders for data elements), requiring a non-trivial matching with the original source shape. If this shape alignment problem is not addressed, then the tendency of a positional shape alignment is to reflect these view modifications at the "leaves" of the source shape, causing the precise positions at which the modifications occur in the view shape to be ignored.

The goal of this section is to understand how we can use the delta information to infer meaningful shape updates. However, propagating shape updates requires knowing the behavior of the transformation, in order to establish correspondences between source and view shapes. Instead of considering arbitrary reshaping lenses, we introduce two regular structural recursion combinators that perform shape alignment: catamorphisms (folds) that consume recursive sources, and anamorphisms (unfolds) that produce recursive views.

5.4.1 Identifying and Propagating Shape Updates

Our proposal for shape alignment is to identify insertions and deletions at the "head" of the view shape, and propagate them to corresponding insertions and deletions at the "head" of the source shape. Consider, as an example, the following Haskell code for the forward transformation of the lspine lens:

$$\begin{aligned}&\mathit{get}_{\mathit{lspine}} :: \mathit{Tree}\ a \to [a]\\&\mathit{get}_{\mathit{lspine}}\ \mathit{Empty} = [\,]\\&\mathit{get}_{\mathit{lspine}}\ (\mathit{Node}\ x\ l\ r) = x : \mathit{get}_{\mathit{lspine}}\ l\end{aligned}$$

This function traverses the input tree and, for each non-empty node, builds a list whose head is the root element and whose tail is computed by recursively applying the transformation to the left child of the tree. The "head" of a value of an arbitrary recursive type can be considered as everything contained in its type constructors besides recursive invocations, i.e., something with the same top-level shape but with the recursive occurrences erased. For non-empty trees and lists, these heads coincide with the root and head elements of a value.

A suitable state-based $\mathit{put}_{\mathit{lspine}}\ (v, s)\ d$ would then recursively match an updated view list v with an original source tree s, consuming their respective heads at each recursive step, but disregarding the view delta d. To avoid this positional behavior, we propose to offset such default matching for insertions and deletions, using the view delta to infer shape modifications. In general, when executing put, if none of the elements at the "head" of the new view are related to elements in the original view, then we are confident that they were created with the update and shall be propagated as an insertion to the source. Conversely, if none of the elements at the "head" of the original view are related to elements in the new view, then such "head" shall be deleted from the original source before proceeding. Otherwise, we proceed positionally. Note that moving is not an operation on shapes. For instance, if we swap the elements of a list $[1, 2, 3]$ to [3,2,1], the shapes of both lists are the same, and the move modifications captured by the view delta shall be handled via data alignment.

For the lspine transformation, since each cons constructor (:) in the original list came from a Node in the original tree, if we insert a new cons at the head of the updated list, then we must insert a new Node at the head of the updated tree, with any default right child since it will be ignored by $\mathit{get}_{\mathit{lspine}}$. For insertions, we can define $\mathit{put}_{\mathit{lspine}}\ (x : \mathit{xs}, t) = \mathit{Node}\ x\ (\mathit{put}_{\mathit{lspine}}\ (\mathit{xs}, t))\ \mathit{Empty}$, with Empty as a default right child. If we delete a cons from the original view, then we must delete the corresponding Node from the original tree, leaving an additional choice on how to merge the children of the deleted source node into a single tree. Proceeding recursively, the left spine of the updated source tree will be copied from the updated view list and right children will be recovered from the merged tree. For deletions, we can define $put_{lspine}\ (xs, Node\ x\ l\ r) = put_{lspine}\ (xs, plus\ l\ r)$, where $plus$ is any function that merges left and right subtrees.

In the remainder of this section, we show how to formalize and generalize this mechanism for arbitrary folds and unfolds.

5.4.2 Higher-order Functor Mapping

A higher-order functor maps natural transformations to natural transformations via an operation $\forall f : \mathsf{F} \dot{\to} \mathsf{G}.\ \mathcal{F}\ f : \mathcal{F}\ \mathsf{F} \dot{\to} \mathcal{F}\ \mathsf{G}$ (MacQueen and Tofte, 1994). For our class of regular higher-order functors, we can define a similar operation $\forall f : \mathsf{F}\ A \to \mathsf{G}\ A.\ \mathcal{F}\ f : \mathcal{F}\ \mathsf{F}\ A \to \mathcal{F}\ \mathsf{G}\ A$ for argument functions that are not natural transformations, i.e., defined for a specific instantiation of type A and not polymorphically quantified for all possible instantiations of A. Moreover, this operation can be lifted to a horizontal delta lens $\forall f : \mathsf{F}_A \unrhd_{\Delta} \mathsf{G}_A.\ \mathcal{F}\ f : \mathcal{F}\ \mathsf{F}_A \unrhd_{\Delta} \mathcal{F}\ \mathsf{G}_A$ defined polytypically over the structure of the higher-order functor, as follows:

$$
\begin{aligned}
&\forall f : \mathsf{F}_A \unrhd_{\Delta} \mathsf{G}_A.\ \mathcal{F}\ f : \mathcal{F}\ \mathsf{F}_A \unrhd_{\Delta} \mathcal{F}\ \mathsf{G}_A\\
&\mathcal{I}d\ f = f\\
&\mathcal{P}ar\ f = id\\
&\underline{C}\ f = id\\
&(\mathcal{F} \boxtimes \mathcal{G})\ f = \mathcal{F}\ f \times \mathcal{G}\ g\\
&(\mathcal{F} \boxplus \mathcal{G})\ f = \mathcal{F}\ f + \mathcal{G}\ f
\end{aligned}
$$

Similarly to the calculus of positions, for the case of composition $(\mathsf{F} \boxdot \mathcal{G})\ f$, we can not use regular functor mapping $\mathsf{F}\ (\mathcal{G}\ f)$ (as done before in Chapter 4 for functor mapping) because the resulting shapes are not the same. Instead, we define auxiliary functions that operate by induction over the structure of the left functor F and unroll functor compositions according to the set of equations used above for calculating positions. The same kind of auxiliary functions will be required for further polytypic definitions over higher-order functions.

Note that, unlike our primitive horizontal delta lens functor mapping combinator, this time the transformation occurs at the level of shapes and not at the data level (the type A of elements is preserved), and no data alignment is due.

5.4.3 Catamorphism

A catamorphism recursively consumes source values, and at each recursive step generates a target value for each consumed source head. Given a horizontal delta lens algebra $f : \mathcal{F}\ \mathsf{G}_A \unrhd_{\Delta} \mathsf{G}_A$, the positional catamorphism $\llparenthesis f \rrparenthesis_{\mathcal{F}}^{pos} : \mu\mathcal{F}_A \unrhd_{\Delta} \mathsf{G}_A$ can be defined as the unique horizontal delta lens that satisfies the following equation:

$$
\begin{array}{l}
\forall f : \mathcal{F}\ \mathsf{G}_A \unrhd_{\Delta} \mathsf{G}_A.\ \llparenthesis f \rrparenthesis_{\mathcal{F}}^{pos} : \mu\mathcal{F}_A \unrhd_{\Delta} \mathsf{G}_A \\
\llparenthesis f \rrparenthesis_{\mathcal{F}}^{pos} = f \circ \mathcal{F}\ \llparenthesis f \rrparenthesis_{\mathcal{F}}^{pos} \circ out_{\mathcal{F}}
\end{array}
$$

The state-based components of this lens correspond to the state-based formulation using recursion patterns presented in Chapter 4. As before, for the resulting horizontal delta lens to be well-behaved we must assume that the respective anamorphisms are recursive, i.e. terminating.

Although this definition receives and propagates deltas, it is purely positional and does not use such information to perform shape alignment. In particular, $put_{\mathcal{F}\ \llparenthesis f \rrparenthesis_{\mathcal{F}}^{pos}}$ matches the view and source values positionally (according to the higher-order functor mapping lens) and passes incomplete delta information to the f argument of the fold due to the lossy behavior of the product horizontal delta lens.

For recursive source values, we can generically compute their head using the expression $get_{\mathcal{F}\ !} \circ out_{\mathcal{F}} : \mu\mathcal{F}\ A \to \mathcal{F}\ \underline{1}\ A$, where $! : \mathsf{F}\ \ A \to \underline{1}\ \ A$ is the forward transformation of the bang horizontal delta lens combinator. However, the view type is not recursive in general and the notion of head of the view is dependent on the semantics of the fold. In fact, what we need to compare are the elements of the view that would be necessary to build a head in the source. This head can be computed by issuing the $create$ of the argument algebra and then erasing the recursive occurrences as before: $get_{\mathcal{F}\ !} \circ create_f : \mathsf{G}\ A \to \mathcal{F}\ \underline{1}\ A$.

Equipped with these procedures to detect the head of source and view values, we specify our horizontal delta lens catamorphism $\llparenthesis f \rrparenthesis_{\mathcal{F}} : \mu\mathcal{F}_A \unrhd_{\Delta} \mathsf{G}_A$ by copying the behavior of $\llparenthesis f \rrparenthesis_{\mathcal{F}}^{pos}$ for get and $create$, and redefining put to handle alignment information as follows:

$$
put_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (v, s)\ d = \begin{cases} grow\,(v, s)\,d & \textbf{if } \rho V \neq \bot \wedge (\rho V \cap \delta d) = \bot \\ shrink\,(v, s)\,d & \textbf{if } \rho S \neq \bot \wedge (\rho S \cap \rho d) = \bot \\ put_{f \circ \mathcal{F}\ \llparenthesis f \rrparenthesis_{\mathcal{F}} \circ out_{\mathcal{F}}}\,(v, s)\,d & \textbf{otherwise} \end{cases}
$$

$$
\textbf{where} \quad V = create_{\triangle f} \circ get_{\triangle_{\mathcal{F}\,!}} \quad S = get^{\circ}_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ get_{\Delta \mathcal{F}\ f}
$$

Here, the V and S relations are the corresponding deltas between the computed head of the updated view and the (full) updated view and between the head of the original view and the (full) original view, respectively. These deltas can be obtained just by dualization of the respective state-based functions, and are illustrated (as dotted arrows) in the following diagram:

$$\begin{tikzcd}
& \mathsf{G}\ A \arrow[dr, dashed, "get_{\Delta (\!|f|\!)_{\mathcal{F}}}"] & \\
\mathsf{G}\ A \arrow[ur, dashed, "d"] \arrow[d, shift right, "create_f"'] & & \mu\mathcal{F}\ A \arrow[ul, "(\!|f|\!)_{\mathcal{F}}"] \arrow[d, shift left, "out_{\mathcal{F}}"'] \\
\mathcal{F}\ \mathsf{G}\ A \arrow[u, dashed, shift right, "create_{\Delta f}"] \arrow[d, shift right, "get_{\mathcal{F}}\ !"'] & & \mathcal{F}\ \mu\mathcal{F}\ A \arrow[u, dashed, shift left, "id"'] \arrow[d, shift left, "get_{\mathcal{F}}\ !"'] \\
\mathcal{F}\ \underline{1}\ A \arrow[u, dashed, shift right, "get_{\Delta\mathcal{F}}\ !"] & & \mathcal{F}\ \underline{1}\ A \arrow[u, dashed, shift left, "get_{\Delta\mathcal{F}}\ !"']
\end{tikzcd}$$

In this diagram, the V delta relates elements at the head of the updated view of type $\mathsf{F}\ \underline{1}\ A$ with the elements of the updated view $v : \mathsf{G}\ A$. If none of the positions in the range of V are contained in the domain of the update delta d, i.e., the set of updated view elements related to the original view, we insert the head of the view in the source with $grow$. The S delta (between the head of the original view and the original view) can be determined by composing the delta of the expression that calculates the head of the source with the converse of the delta of the forward catamorphism that computes the original view $get_{(\!|f|\!)_{\mathcal{F}}}\ s : \mathsf{G}\ A$ from the original source $s : \mu\mathcal{F}\ A$. This way, we select only the values at the head of the source that are also at the head of the view. This nuance is important to prevent the deletion of source elements that are abstracted by the view and should be restored by put. Thus, if none of the positions in the range of S are contained in the range of the update delta d, i.e., the set of original view elements related to the updated view, we can safely delete the head of the source with $shrink$.

Another detail of our definition is that we verify if the V and S relations are empty. Since our argument is based on data elements, if the head of a value does not contain any elements (e.g., like an empty list constructor), what would lead to an empty delta relation, we do not judge. The proof that our alignment-aware catamorphism delta lens is well-behaved is detailed in Section B.3.5.

Insertion The head of the view can be isolated by invoking $create_f$ to produce a value of type $\mathcal{F}\ \mathsf{G}\ A$. To propagate a newly created view head, we need a way to pair each $\mathsf{G}\ A$ inside $\mathcal{F}\ \mathsf{G}\ A$ with the original source of type $\mu\mathcal{F}\ A$, to which we can apply $put_{(\!|f|\!)_{\mathcal{F}}}$ recursively. In category theory, a functor is said *strong* if it is equipped with a natural transformation $\sigma_{\mathsf{F}} : \mathsf{F}\ A \times B \to \mathsf{F}\ (A \times B)$, denoted *strength*, that pairs the B with each A inside the functor. In the literature, strength usually involves additional conditions, namely:

$$\mathsf{F}\ \pi_1 \circ \sigma_{\mathsf{F}} = \pi_1 \qquad \sigma\text{-Cancel}$$

$$\mathsf{F}\ assocr \circ \sigma_{\mathsf{F}} \circ (\sigma_{\mathsf{F}} \times id) = \sigma_{\mathsf{F}} \circ assocr \qquad \sigma\text{-Assoc}$$

This strength function can easily be lifted and defined polytypically for every regular higher-order functor: $\sigma_{\mathcal{F}} : \mathcal{F}\ \mathsf{F}\ A \times \mathsf{G}\ A \to \mathcal{F}\ (\mathsf{F} \otimes \mathsf{G})\ A$.

Not taking deltas into account (as they can be computed by dualization of the state-based transformations), the $grow$ procedure can be specified as depicted in Figure 5.5. Notice that, if the functor contains more than one recursive occurrence (for trees for example), then $\sigma_{\mathcal{F}}$ will duplicate the original source for each recursive invocation of put. This is because, when invoking $\sigma_{\mathcal{F}}$ at a recursive step, the catamorphism does not know how to split the source so that each piece is related to the respective recursive view. Instead, the duplicated sources will be later aligned recursively. For example, unrelated source elements will be deleted by $shrink$. The actual implementation of $grow$ is $in_{\mathcal{F}} \circ \sigma put_{\mathcal{F}}\ (d \circ create_{\Delta f}) \circ (create_f \times id)$, where $\sigma put_{\mathcal{F}}$ is a polytypic auxiliary definition that implements the specification $\mathcal{F}\ put_{(\!|f|\!)_{\mathcal{F}}} \circ \sigma_{\mathcal{F}}$, with the necessary propagation of deltas to the recursive invocations of put.

For each lens $l : S_A \trianglerighteq_{\Delta} V_A$, we define a $\sigma put_{\mathcal{F}}$ function that receives a delta and pairs view values inside the functor with duplicated source values, invoking put_l (with the argument delta) to process recursive cases:

$$\begin{aligned}
&\sigma put_{\mathcal{F}} : \forall (v, s) : \mathcal{F}\ V_A \times S_A.\ v\, \Delta\, get\ s \to \mathcal{F}\ S_A\\
&\sigma put_{\mathcal{I}d}\ (v, s)\ d = put_l\ (v, s)\ d\\
&\sigma put_{\underline{C}}\ (v, s)\ d = v\\
&\sigma put_{\mathcal{P}ar}\ (v, s)\ d = v\\
&\sigma put_{\mathcal{F} \boxtimes \mathcal{G}}\ ((x, y), s)\ d = (\sigma put_{\mathcal{F}}\ (x, s)\ (d \circ i_1), \sigma put_{\mathcal{G}}\ (y, s)\ (d \circ i_2))\\
&\sigma put_{\mathcal{F} \boxplus \mathcal{G}}\ (i_1\ x, s)\ d = i_1\ (\sigma put_{\mathcal{F}}\ (x, s)\ d)\\
&\sigma put_{\mathcal{F} \boxplus \mathcal{G}}\ (i_2\ y, s)\ d = i_2\ (\sigma put_{\mathcal{G}}\ (y, s)\ d)
\end{aligned}$$

The induced $\sigma put_{\Delta \mathcal{F}}$ horizontal delta is defined as follows:

$$\sigma put_{\Delta \mathcal{F}} : \forall \{(v, s) : \mathcal{F}\ V_A \times S_A\}, d : v\, \Delta\, get\ s.\ (\sigma put_{\mathcal{F}}\ (v, s)\ d)\, \Delta\, (v, s)$$

$$\begin{tikzcd}
\mu\mathcal{F}\;A & & \mathcal{F}\;\mu\mathcal{F}\;A \arrow[ll, "in_{\mathcal{F}}"'] \\
 & & \mathcal{F}\;(\mathsf{G}\;A \times \mu\mathcal{F}\;A) \arrow[u, "\mathcal{F}\;put_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}"'] \\
\mathsf{G}\;A \times \mu\mathcal{F}\;A \arrow[uu, "grow"] \arrow[rr, "create_f \times id"'] & & \mathcal{F}\;\mathsf{G}\;A \times \mu\mathcal{F}\;A \arrow[u, "\sigma_{\mathcal{F}}"']
\end{tikzcd}$$

Figure 5.5: Specification of *grow* for catamorphisms.

$$\begin{aligned}
&\sigma put_{\Delta\mathcal{I}d}\;\{(v,s)\}\;d = put_{\Delta l}\;\{(v,s)\}\;d\\
&\sigma put_{\Delta\underline{C}}\;\{(v,s)\}\;d = i_1\\
&\sigma put_{\Delta\mathcal{P}ar}\;\{(v,s)\}\;d = i_1\\
&\sigma put_{\Delta\mathcal{F}\boxtimes\mathcal{G}}\;\{((x,y),s)\}\;d =\\
&\quad((i_1 + id)\circ \sigma put_{\Delta\mathcal{F}}\;\{(x,s)\}\;(d \circ i_1) \triangledown (i_2 + id) \circ \sigma put_{\Delta\mathcal{G}}\;\{(y,s)\}\;(d \circ i_2))\\
&\sigma put_{\Delta\mathcal{F}\boxplus\mathcal{G}}\;\{(i_1\;x,s)\}\;d = \sigma put_{\Delta\mathcal{F}}\;\{(x,s)\}\;d\\
&\sigma put_{\Delta\mathcal{F}\boxplus\mathcal{G}}\;\{(i_2\;y,s)\}\;d = \sigma put_{\Delta\mathcal{G}}\;\{(y,s)\}\;d
\end{aligned}$$

Deletion Again not taking deltas into account, the *shrink* procedure can be specified as depicted in Figure 5.6. To propagate a deletion, we unfold the original source value to expose its head to be erased by the auxiliary function $reduce_{\mathcal{F}}$, and then apply $put_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}$ to the argument view and the reduced source. In the implementation, $put_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}$ is invoked with the delta $get^{\circ}_{\Delta\llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ reduce_{\mathcal{F}}{}^{\circ}_{\Delta} \circ get_{\Delta\llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ d$ that reflects the shape changes performed by $reduce_{\mathcal{F}}$.

In order to erase the head, the function $reduce_{\mathcal{F}}$ merges the recursive occurrences of a functor into a single value and is defined polytypically as follows:

$$\begin{aligned}
&reduce_{\mathcal{F}} : \mathcal{F}\;\mathsf{F}_A \to \mathsf{F}_A\\
&reduce_{\mathcal{I}d}\;x = x\\
&reduce_{\underline{C}}\;x = zero\\
&reduce_{\mathcal{P}ar}\;x = zero\\
&reduce_{\mathcal{F}\boxtimes\mathcal{G}}\;(x,y) = plus\;(reduce_{\mathcal{F}}\;x, reduce_{\mathcal{G}}\;y)\\
&reduce_{\mathcal{F}\boxplus\mathcal{G}}\;(i_1\;x) = reduce_{\mathcal{F}}\;x\\
&reduce_{\mathcal{F}\boxplus\mathcal{G}}\;(i_2\;x) = reduce_{\mathcal{F}}\;y
\end{aligned}$$

Here, we assume that the source type $\mathsf{F}\;A$ is a monoid-like algebraic structure with a $zero : \mathsf{F}\;A$ element and a binary concatenation operation $plus : \mathsf{F}\;A \times \mathsf{F}\;A \to \mathsf{F}\;A$.

$$\begin{tikzcd}
\mu\mathcal{F}\ A & \mathsf{G}\ A \times \mu\mathcal{F}\ A \arrow[l, "put_{(\!|f|\!)_{\mathcal{F}}}"'] \\
\mathsf{G}\ A \times \mu\mathcal{F}\ A \arrow[u, "shrink"] \arrow[r, "id \times out_{\mathcal{F}}"'] & \mathsf{G}\ A \times \mathcal{F}\ \mu\mathcal{F}\ A \arrow[u, "id \times reduce_{\mathcal{F}}"']
\end{tikzcd}$$

Figure 5.6: Specification of *shrink* for catamorphisms.

This allows to define $reduce_{\mathcal{F}}$ just by folding the sequence of recursive occurrences using these two operations. For types other than lists, there could be possibly more than one such monoid-like structures. In the implementation of our point-free delta lens language (Chapter 6), we provide default instances for many types, but users are free to provide their own structures. By defining the type signatures of *zero* and *plus* as polymorphic functions, we ensure that they are natural transformations and thus the respective deltas can be computed using the semantic technique presented before.

Moreover, to guarantee that *put* terminates and the catamorphism lens is well-behaved, we must require the composition $reduce_{\mathcal{F}} \circ out_{\mathcal{F}} : \mu\mathcal{F}\ A \rightharpoonup \mu\mathcal{F}\ A$ to be a *well-founded relation* (Bird and de Moor, 1997), in the sense that a source value is always "reduced" to a strictly smaller value after applying $reduce_{\mathcal{F}}$.

Examples We can now encode the examples from the beginning of this chapter as horizontal delta lenses. For instance, the *lspine* lens used in the *fathernames* example can be defined as the following horizontal delta lens fold:

$$
\begin{aligned}
&lspine : Tree_A \trianglerighteq_{\Delta} [\,]_A \\
&lspine = (\![in_{\mathcal{L}ist} \circ (id + id \times \pi_1{}^{\underline{[\,]} \circ !})]\!)_{\mathcal{T}ree}
\end{aligned}
$$

$$
\begin{aligned}
&plus_{\mathcal{T}ree} : Tree \otimes Tree \xrightarrow{\cdot} Tree \qquad && zero_{\mathcal{T}ree} : \forall A.\ Tree\ A \\
&plus_{\mathcal{T}ree}\ Empty\ r = r && zero_{\mathcal{T}ree} = Empty \\
&plus_{\mathcal{T}ree}\ l\ r = l
\end{aligned}
$$

When ran against the *fathernames* example, this lens produces the desired result. For insertions, put_{lspine} will generate a default empty list of right children, that when aligned with any source tree will originate an empty right tree. For deletions, the given *plus* function selects one of the child trees if the other is empty, or discards the right child otherwise.

Note that although no deletion is performed by the view update, the σput procedure for insertions will duplicate the original source tree, but the right-side duplicated tree will be successfully marked as deleted by our backward semantics and disposed of by the $plus$ function. Also for the $lspine$ example, we know that an insertion followed by a deletion would lead to no effect on the source.

Since the state-based $lspine$ lens is of the special form $(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}$ and $id + id \times \pi_1 : \underline{1} \oplus \underline{A} \otimes (\mathsf{Id} \otimes \mathsf{Id}) \dot{\to} \underline{1} \oplus \underline{A} \otimes \mathsf{Id}$ is a natural transformation, to prove that the $lspine$ delta lens is well-behaved we only need to prove that $reduce_{\mathcal{F}} \circ out_{\mathcal{F}}$ is well-founded, what is trivial for the given $zero$ and $plus$ implementations.

In general, the choice of a monoid-like structure can be arbitrary and will only affect the way some information is recovered from the original source. A good principle in the design of such an algebraic structure is to safeguard that $plus$ actually merges two source values without introducing new or duplicate information, to somehow ensure that $reduce$ does indeed "reduce" the size of the source and satisfies the required well-foundedness properties. Still, a particular structure might make more sense for a particular lens. For example, as long as $lspine$ drops the right children of a tree, it makes more sense to ignore right values in the definition of $plus_{Tree}$; if we instead chose to merge both subtrees, right subtrees of the merged tree would eventually be recovered by put and appear as right elements in the new source tree. Note that we do not actually enforce the usual identity and associativity monoid laws, since non-monoidal binary operators such as alternating merge would may also be plausible for particular lenses.

The $filter\_right$ lens from the initial $females$ example can also be encoded as a horizontal delta lens fold:

$$\begin{aligned}
&filter\_right : (ListOpt\ A)_B \unrhd [\,]_B \\
&filter\_right = (\!|(in_{\mathcal{List}} \mathbin{\bullet\!\nabla} \pi_2) \circ coassocl \circ (id + coswap \circ distl)|\!)_{\mathcal{ListOpt}_A}
\end{aligned}$$

By running this lens against the $females$ example, left (male) elements are now restored properly. Here, the $coassoc : (\mathsf{F} \oplus (\mathsf{G} \oplus \mathsf{H}))_A \unrhd_{\Delta} ((\mathsf{F} \oplus \mathsf{G}) \oplus \mathsf{H})_A$, $coswap : (\mathsf{F} \oplus \mathsf{G})_A \unrhd_{\Delta} (\mathsf{G} \oplus \mathsf{F})_A$ and $distl : ((\mathsf{F} \oplus \mathsf{G}) \otimes \mathsf{H})_A \unrhd_{\Delta} ((\mathsf{F} \otimes \mathsf{H}) \oplus (\mathsf{G} \otimes \mathsf{H}))_A$ combinators are horizontal delta lens isomorphisms.

In our language, a type $\mathsf{F}\ (\mathsf{G}\ A)$ is different in shape from an isomorphic type $(\mathsf{F} \odot \mathsf{G})\ A$. For this reason, we have introduced a new $ListOpt\ A\ B$ type (isomorphic to $[A + B]$) to encode the shape of optional lists, despite a state-based $filter\_right$ could be defined directly as a catamorphism over lists (similarly to the $filter\_left$ lens from Chapter 4). To avoid introducing a new type and adapting the $filter\_right$ lens to work over $\mathit{ListOpt}$ instead of the standard list type, we could lift it to a horizontal delta-based catamorphism over the fixed point $\mu(\mathcal{List} \boxdot (\underline{A} \oplus \mathsf{Id}))$, that is isomorphic to the $\mathit{ListOpt}\ A$ functor and represents the desired shape.

5.4.4 Anamorphism

Dually to catamorphisms, anamorphisms recursively construct view values. Given a horizontal delta lens coalgebra $f : \mathsf{F}_A \trianglerighteq_{\triangle} \mathcal{G}\ \mathsf{F}_A$, the positional anamorphism $\llbracket f \rrbracket^{pos}_{\mathcal{G}} : \mathsf{F}_A \trianglerighteq_{\triangle} \mu\mathcal{G}_A$ can be defined as the unique horizontal delta lens that satisfies the following equation:

$$
\begin{array}{l}
\forall f : \mathsf{F}_A \trianglerighteq_{\triangle} \mathcal{G}\ \mathsf{F}_A .\ \llbracket f \rrbracket^{pos}_{\mathcal{G}} : \mathsf{F}_A \trianglerighteq_{\triangle} \mu\mathcal{G}_A \\
\llbracket f \rrbracket^{pos}_{\mathcal{G}} = in_{\mathcal{G}} \circ \mathcal{G}\ \llbracket f \rrbracket^{pos}_{\mathcal{G}} \circ f
\end{array}
$$

Again, for this combinator to be a well-behaved delta lens, we require that the corresponding point-free state-based functions yield recursive anamorphisms.

For anamorphisms, the view type is recursive and thus has a direct notion of head given by $get_{\mathcal{G}\ !} \circ out_{\mathcal{G}} : \mu\mathcal{G}\ A \rightarrow \mathsf{G}\ \underline{1}\ A$. Since this time the source type is not recursive in general, its notion of head is the one dependent on the semantics of the unfold. Hence, we need to consider the elements of the source that would be necessary to build a head in the view. This head can be computed by applying the get of the argument coalgebra to the source and erasing the recursive occurrences: $get_{\mathcal{G}\ !} \circ get_f : \mathsf{F}\ A \rightarrow \mathsf{G}\ \underline{1}\ A$.

Formally, we specify our horizontal delta lens anamorphism $\llbracket f \rrbracket_{\mathcal{G}} : \mathsf{F}_A \trianglerighteq_{\triangle} \mu\mathcal{G}_A$ by reproducing $\llbracket f \rrbracket^{pos}_{\mathcal{G}}$ for get and $create$, and redefining put as follows:

$$
put_{\llbracket f \rrbracket_{\mathcal{G}}}\ (v, s)\ d = \begin{cases} grow\,(v, s)\,d & \textbf{if}\ \rho V \neq \bot \wedge (\rho V \cap \delta d) = \bot \\ shrink\,(v, s)\,d & \textbf{if}\ \rho S \neq \bot \wedge (\rho S \cap \rho d) = \bot \\ put_{in_{\mathcal{G}} \circ \mathcal{G}\ \llbracket f \rrbracket_{\mathcal{G}} \circ f}\ (v, s)\,d & \textbf{otherwise} \end{cases}
$$

$$
\textbf{where}\quad V = get_{\triangle_{\mathcal{G}\,!}} \quad S = get^{\circ}_{\triangle \llbracket f \rrbracket_{\mathcal{G}}} \circ get_{\triangle f} \circ get_{\triangle_{\mathcal{G}\,!}}
$$

Likewise catamorphisms, the V and S relations are defined in the following diagram:

$$\begin{tikzcd}
 & \mu\mathcal{G}\ A \arrow[dr, dashed, "get_{\Delta[\![f]\!]_{\mathcal{F}}}"] & \\
\mu\mathcal{G}\ A \arrow[ur, dashed, "d"] \arrow[d, "out_{\mathcal{G}}"] & & \mathsf{F}\ A \arrow[ul, "[\![f]\!]_{\mathcal{F}}"] \arrow[d, "get_f"'] \\
\mathcal{G}\ \mu\mathcal{G}\ A \arrow[u, dashed, "id"] \arrow[d, "get_{\mathcal{G}}\ !"] & & \mathcal{G}\ \mathsf{F}\ A \arrow[u, dashed, "get_{\Delta f}"'] \arrow[d, "get_{\mathcal{G}}\ !"'] \\
\mathcal{G}\ \underline{1}\ A \arrow[u, dashed, "get_{\Delta\mathcal{G}}\ !"] & & \mathcal{G}\ \underline{1}\ A \arrow[u, dashed, "get_{\Delta\mathcal{G}}\ !"']
\end{tikzcd}$$

The behavior of $put_{[\![f]\!]_{\mathcal{G}}}$ modulo these deltas is also similar. The V delta relates elements at the head of the updated view with elements of the updated view $v : \mu\mathcal{G}\ A$. If none of the elements at the head of the updated view (in the range of V) are related to elements of the original view (in the domain of the update delta d), we insert the head of the view in the source using $grow$. The S delta (between the head of the original view and the original view) can be determined by applying the converse of the delta of the forward anamorphism, that computes the original view $get_{[\![f]\!]_{\mathcal{G}}}\ s : \mu\mathcal{G}\ A$ from the original source $s : \mu\mathcal{F}\ A$, after the delta of the expression that calculates the head of the source. Dually, if none of the elements at the head of the original view (in the range of S) are related to elements of the updated view (in the range of the update delta d), we delete the head of the corresponding source with $shrink$.

The proof that our anamorphism delta lens is well-behaved is similar to the one for catamorphisms, assuming that the positional anamorphism lens is well-behaved and that $reduce_{\mathcal{F}}$ produces smaller source values.

Insertion The $grow$ procedure is specified as depicted in Figure 5.7. The head of the view is isolated by applying $out_{\mathcal{G}}$ to produce a value of type $\mathcal{G}\ \mathsf{G}\ A$. As for catamorphisms, we propagate this newly created head using the strength combinator $\sigma_{\mathcal{F}}$ followed by applying $put_{[\![f]\!]_{\mathcal{G}}}$ recursively taking as argument the original delta d. The recursively computed value of type $\mathcal{G}\ \mathsf{F}\ A$ is then converted into a source value with a new head using $create_f$.

Deletion The $shrink$ procedure can be defined as depicted in Figure 5.8. To expose the head of the source, we invoke get_f to produce a value of type $\mathcal{G}\ \mathsf{F}\ A$, whose head can be erased by the $reduce_{\mathcal{G}}$ function. We then apply $put_{[\![f]\!]_{\mathcal{G}}}$ to the argument

$$\begin{tikzcd}
\mu\mathcal{G}\ A \times \mathsf{F}\ A \arrow[r, "out_{\mathcal{G}} \times id"] \arrow[dd, "grow"'] & \mathcal{G}\ \mu\mathcal{G}\ A \times \mathsf{F}\ A \arrow[d, "\sigma_{\mathcal{F}}"] \\
 & \mathcal{G}\ (\mu\mathcal{G}\ A \times \mathsf{F}\ A) \arrow[d, "\mathcal{G}\ put_{[\![f]\!]_{\mathcal{G}}}"] \\
\mathsf{F}\ A & \mathcal{G}\ \mathsf{F}\ A \arrow[l, "create_f"]
\end{tikzcd}$$

Figure 5.7: Specification of *grow* for anamorphisms.

$$\begin{tikzcd}
\mu\mathcal{G}\ A \times \mathsf{F}\ A \arrow[r, "id \times get_f"] \arrow[d, "shrink"'] & \mu\mathcal{G}\ A \times \mathcal{G}\ \mathsf{F}\ A \arrow[d, "id \times reduce_{\mathcal{F}}"] \\
\mathsf{F}\ A & \mu\mathcal{G}\ A \times \mathsf{F}\ A \arrow[l, "put_{[\![f]\!]_{\mathcal{G}}}"]
\end{tikzcd}$$

Figure 5.8: Specification of *shrink* for anamorphisms.

view and the reduced source. In the implementation, $put_{[\![f]\!]_{\mathcal{G}}}$ is invoked with the delta $get^{\circ}_{\Delta [\![f]\!]_{\mathcal{G}}} \circ reduce_{\mathcal{F}}{}^{\circ}{}_{\Delta} \circ get^{\circ}_{\Delta f} \circ get_{\Delta [\![f]\!]_{\mathcal{G}}} \circ d$ that reflects the shape changes performed by $reduce_{\mathcal{G}}$. For the anamorphism lens to be well-behaved, we require $reduce_{\mathcal{G}} \circ get_f : \mathsf{F}\ A \to \mathsf{F}\ A$ to be well-founded.

Examples Since the recursive gene of *lspine* is a natural lens transformation, *lspine* can be alternatively defined as the following horizontal delta lens anamorphism:

$$\begin{aligned}
&lspine : Tree_A \trianglerighteq_{\Delta} [\,]_A \\
&lspine = [\![(id + id \times \pi_1\frac{\mathit{Empty} \circ\, !}{}) \circ out_{\mathit{Tree}}]\!]_{\mathcal{L}ist}
\end{aligned}$$

Despite producing similar results for our running example, this definition is more intuitive in terms of deletion, because $reduce_{\mathcal{F}}$ is applied to the base functor of the view list (for which head deletion is intuitively defined) rather than to the base functor of the source tree, for which *mplus* has to merge the two left and right child trees.

In general, it is better to specify any recursive horizontal delta lens whose gene is (almost a) natural transformation as an anamorphism instead of as a catamorphism, since the lens laws entail that the view base functor will always be smaller than the source base functor, and thus have a tendency to contain a smaller number of recursive invocations. This means that (1) insertion is more intuitive because σ will have to duplicate the original source less times, and that (2) deletion is more intuitive because get_f will partially truncate the original source and $reduce$ will have to merge a smaller number of recursive children of the original source.

An example of an anamorphism lens whose gene is not a natural transformation is the combinator that sieves a list to keep every second element:

$$\begin{array}{l} sieve : (A \to A) \to [\,]_A \trianglerighteq_{\Delta} [\,]_A \\ sieve\ f = [\![(id \mathbin{\bullet\!\nabla} \pi_2 + {\pi_2}^{f \circ \pi_1}) \circ coassocl \circ (id + distr \circ (id \times out_{\mathcal{L}ist})) \circ out_{\mathcal{L}ist}]\!]_{\mathcal{L}ist} \end{array}$$

Note that the gene of the anamorphism is not “almost a natural lens” as termed in Chapter 4 (with regard to the base functor $\underline{1} \oplus \underline{A} \oplus \mathsf{Id}$ of the target type $[A]$) since it inspects the recursive values of the source (by unfolding the source twice to reveal the first two elements of a source list), even though the source type $[A]$ is the same. Here, the parameter function f is responsible for generating a default first element for each new second element in the view. Imagine that we sieve a list $[1,2,3,4]$ of odd and even numbers to get only the even ones $[2,4]$ (assumed to be in even positions of the source list), and modify the view list to $[0,2,6]$. A state-based $sieve\ pred : [Int] \to [Int]$, where $pred$ is a function that returns the predecessor of a number, would then produce the following result:

$$put_{sieve\ pred}\ [0,2,6]\ [1,2,3,4] = [1,0,3,2,5,6]$$

In this case, the backward transformation copies the even numbers from the view numbers and restores odd numbers positionally from the source. Using suitable deltas, our $sieve$ delta lens would behave as follows (with the deltas represented graphically):

$$\begin{array}{l} \textbf{let}\ s = [1,2,3,4] \\ \textbf{in}\ get_{sieve\ pred}\ s = [2,4] \quad put_{sieve\ pred}\ [0,2,6]\ s = [-1,0,1,2,3,6] \end{array}$$

This time, the backward transformation infers from the deltas that 0 has been inserted, 2 preserved and 4 modified to 6, and processes the preceding odd numbers accordingly: since get produces a view number for each pair of source numbers, put will create a new preceding odd number for insertions in the view and preserve the preceding numbers for view numbers related through the delta.

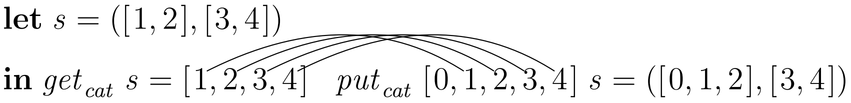
Another example that combines a fold with an unfold is list concatenation, that can be lifted to the following horizontal delta lens:

$$
\begin{aligned}
&\mathit{cat} : ([\,] \otimes [\,])_A \trianglerighteq_{\Delta} [\,]_A \\
&\mathit{cat} = \llparenthesis \mathit{id} \triangledown \mathit{cons} \rrparenthesis_{\mathcal{NeList}} \circ \llbracket (\pi_2{}^! + \mathit{assocr}) \circ \mathit{distl} \circ (\mathit{out}_{HL} \times \mathit{id}) \rrbracket_{\mathcal{NeList}}
\end{aligned}
$$

Here, $\mathcal{NeList}$ is the higher-order base functor $(\mathcal{List} \boxdot \mathcal{Par}) \boxplus \mathcal{Par} \boxtimes \mathcal{Id}$ of the NeList type. To understand the behavior of this lens, consider the following execution of the positional cat lens from Chapter 4:

$$\mathit{put}_{\mathit{cat}}\ [0,1,2,3,4]\ ([1,2],[3,4]) = ([0,1],[2,3,4])$$

Although the size of the view list changes, the state-based backward transformation still splits the view list in two according to the size of the original left list. A better solution would be to track the insertions and deletions in the view that affect the size of the left source list, as our delta lens would do:



$$
\begin{aligned}
&\textbf{let}\ s = ([1,2],[3,4]) \\
&\textbf{in}\ \mathit{get}_{\mathit{cat}}\ s = [1,2,3,4] \quad \mathit{put}_{\mathit{cat}}\ [0,1,2,3,4]\ s = ([0,1,2],[3,4])
\end{aligned}
$$

In this example, the new 0 element has been inserted before the splitting position in the view, and the put of our delta lens increments the size of the left source list.

5.5 Summary

The "holy grail" of bidirectional transformation approaches is to find solutions that mitigate the ambiguity of target update translation, by producing minimal source updates (and vice-versa). For the bidirectional framework of lenses and the application domain of inductive data types, we identify that a smaller update requires not only to align data elements, but also shapes.

In this chapter, we have lifted the point-free lens language from Chapter 4 to a point-free delta lens language for building bidirectional transformations with an explicit notion of shape and data over inductive data types. Our delta lens framework instantiates the abstract framework of delta lenses first introduced by Diskin et al. (2011a), meaning that lenses now propagate deltas to deltas and preserve additional delta-based bidirectional round-tripping axioms. In particular, we have instrumented the standard fold and unfold recursion patterns with mechanisms that use deltas to infer and propagate edit operations on shapes, thus producing smaller source updates that best reflect a certain view update. Thus far, we have considered insertion and deletion updates on shapes, that are sufficiently generic to express modifications on a wide range of data types. Nevertheless, other more refined edit operations (like tree rotations) might make sense for particular types and application scenarios, and our technique could be instrumented to cover more edits in the future. The use of dependent types has provided a more concise formalism that simplifies the existing delta-based bidirectional theory and clarifies the connection between the state- and delta-based components of the framework.

Chapter 6

The *Multifocal* Framework

Lenses, as a framework for data abstraction, can be of great value in scything through the complexity of large software systems. However, writing a lens transformation typically implies describing the concrete steps that convert values in a source schema to values in a target schema, making it hard to express even conceptually simple transformations over large schemas. In contrast, many XML-based languages allow writing structure-shy programs that manipulate only specific parts of XML documents without having to specify the behavior for the remaining structure.

This chapter describes the *Multifocal* framework for the specification, optimization and execution of structure-shy two-level bidirectional lens transformations. The framework is developed in Haskell and is constituted by three main components: a core library of type-specific bidirectional lenses, a high-level library of type-generic lens combinators that instantiate a well-known suite of combinators for strategic programming (Visser, 2001; Lämmel and Visser, 2003) and are compiled to lower-level lenses, and a point-free rewriting library that can be used to optimize the automatically-generated lens transformations.

To demonstrate the practical application of the framework, we also propose the particular *Multifocal* XML two-level bidirectional transformation language, providing both strategic and specific XML transformers, and discuss particular examples involving the generic evolution of recursive XML Schemas. XML transformations written in *Multifocal* are translated into our library of two-level lens combinators, by converting the source and target XML Schemas to equivalent algebraic data type representations.

6.1 A Point-free Lens Library

The point-free style allows functional programmers to write functions that are free of bound variables through composition of other functions. In this section, we re-implement the core of the Pointless Haskell library (Cunha, 2005) for point-free programming using recent language extensions such as type-indexed type families. Then, we implement a Haskell library of point-free lenses on top, including liftings of most point-free combinators presented in Chapter 2 and formalized as lenses in Chapter 4. Our library also implements the point-free delta lenses from Chapter 5.

6.1.1 Basic Lenses

In Haskell, the type of lenses can be defined as a record of three functions: a forward get function, a backward put function and a default $create$ function.

$$\mathbf{data}\ \mathit{Lens}\ c\ a = \mathit{Lens}\ \{\mathit{get} :: c \to a, \mathit{put} :: (a, c) \to c, \mathit{create} :: a \to c\}$$

The primordial lens combinators are identity, that simply copies the argument value in both directions, and composition, that applies two lenses in sequence. Their implementations follow the formal definitions from Chapter 4[1]:

$$\begin{array}{l}
\mathit{id\_lns} :: \mathit{Lens}\ c\ c \\
\mathit{id\_lns} = \mathit{Lens}\ \mathit{id}\ \mathit{fst}\ \mathit{id} \\
(\circ) :: \mathit{Lens}\ b\ a \to \mathit{Lens}\ c\ b \to \mathit{Lens}\ c\ a \\
(\circ)\ f\ g = \mathit{Lens}\ (\mathit{get}\ f\ .\ \mathit{get}\ g)\ \mathit{put}'\ (\mathit{create}\ g\ .\ \mathit{create}\ f) \\
\quad \mathbf{where}\ \mathit{put}' = \mathit{put}\ g\ .\ (\mathit{put}\ f\ .\ (\mathit{id} >\!< \mathit{get}\ g)\ /\backslash\ \mathit{snd})
\end{array}$$

Some of the point-free combinators used in these lens definitions, such as composition (.), id and fst, are primitives of the Haskell language. For others, like $><$ and $/\backslash$, their Haskell definitions are available as part of the Pointless Haskell library (Cunha, 2005), but are straightforward from their respective notation (like $\times$ and $\triangle$) from Chapter 2.

Two other examples of lens combinators over products and sums are left projection (fst_lns), that drops the second element of a concrete pair and accepts an argument

[1] In the Haskell implementation, we denote lens combinators by the $\_lns$ suffix to avoid overloading with the homologous point-free functions. We distinguish infix point-free lens combinators from the standard ones by pretty-printing them according to their mathematical notation.

function as a default value generator for the deleted component, and the either combinator (▽) that “conditionally” applies a lens based on the branching of the concrete value and requires a predicate that chooses the creation of left or right default branches:

```
fst_lns :: (a → b) → Lens (a, b) a
fst_lns f = Lens fst id (id /\ f)
( ▽ ) :: (c → Bool) → Lens a c → Lens b c → Lens (Either a b) c
( ▽ ) p f g = Lens (get f \/ get g) put′ create′
  where put′ = (put f −|− put g) . distr
        create′ = (create f −|− create g) . (p?)
```

The remaining primitive lens combinators can be transcoded directly to Haskell from their theoretical definitions from Chapter 4.

6.1.2 Recursive Lenses

Before implementing recursive lenses, we propose an encoding of functors that enables a restricted notion of structural equivalence between user-defined types and their sums-of-products representations, closely mimicking the theory.

Encoding functors and user-defined types Most inductive data types can be seen as fixed points of polynomial functors (sums of products), encoded in Haskell as follows:

```
newtype Id x = Id {unId :: x}
newtype Const t x = Const {unConst :: t}
data (g ⊕ h) x = Inl (g x) | Inr (h x)
data (g ⊗ h) x = Prod (g x) (h x)
newtype (g ⊙ h) x = Comp {unComp :: g (h x)}
```

Therefore, for an inductive type a, we need to represent its base functor F a. We establish this correspondence using type-indexed type families (Schrijvers et al., 2007; Chakravarty et al., 2005), a recent extension to the Haskell type system supported by the *Glasgow Haskell Compiler* (GHC). Developed with type-level programming in mind, type families are type constructors that represent sets of types. Set members are aggregated according to the type parameters passed to the type family constructor, called type indices: family constructors can have different representation types for different type indices. We can define a type family that when applied to a type returns its base functor, as exemplified for lists:

$$\begin{array}{l}\textbf{type family}\ \mathsf{F}\ a :: * \to * \\ \textbf{type instance}\ \mathsf{F}\ [a] = \mathsf{Const}\ \mathit{One} \oplus \mathsf{Const}\ a \otimes \mathsf{Id}\end{array}$$

Here, the Haskell type One models the categorical unit type possessing one single value.

Moreover, when applying a functor to a type, we want to get an isomorphic sum-of-products type capable of being processed with our original point-free combinators. Since type equivalence in Haskell is not structural but name-based, we define a $\mathit{Rep}\ f\ a$ type family that, given a functor f and a type a, returns the equivalent "flattened" type that results from applying the functor to the argument type. This solution is more elegant and concise than the implicit coercion type classes used in the PolyP (Jansson and Jeuring, 1997) and Pointless Haskell (Cunha, 2005) libraries for polytypic and point-free programming:

$$\begin{array}{l}\textbf{type family}\ \mathit{Rep}\ (f :: * \to *)\ a :: * \\ \textbf{type instance}\ \mathit{Rep}\ \mathsf{Id}\ a = a \\ \textbf{type instance}\ \mathit{Rep}\ (g \otimes h)\ a = (\mathit{Rep}\ g\ a, \mathit{Rep}\ h\ a) \\ \ldots\end{array}$$

Since $\mathit{Rep}\ f$ is a functor if and only if f is a functor, we can now define our own $\mathit{Functor}$ type class (that substitutes the already existent $\mathit{Functor}$ class provided in the Haskell prelude) for mapping functions over functors and zipping functors (required later for our recursive lens combinators). The corresponding instances of fmap are trivial to define, and the instances of fzip can be extrapolated from their formal definitions presented in Chapter 4:

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ (f :: * \to *)\ \textbf{where} \\ \quad \mathit{fmap} :: (\mu\ f) \to (x \to y) \to \mathit{Rep}\ f\ x \to \mathit{Rep}\ f\ y \\ \quad \mathit{fzip}\ \ :: (\mu\ f) \to (x \to y) \to (\mathit{Rep}\ f\ x, \mathit{Rep}\ f\ y) \to \mathit{Rep}\ f\ (x, y)\end{array}$$

The additional $\mu\ f$ parameter encodes the fixed point of the functor f and is required by the compiler as an annotation of the type of the functor. In Haskell, it can be defined as a recursive type that applies the argument functor to itself:

$$\textbf{newtype}\ \mathit{Functor}\ f \Rightarrow \mu\ f = \mathit{In}\ \{\mathit{ouT} :: \mathit{Rep}\ f\ (\mu\ f)\}$$

It is useful to define polytypic map and zip functions that work over the base functor of an argument type annotation:

$$\begin{array}{l}
pmap :: Functor\ (\mathsf{F}\ a) \Rightarrow a \to (x \to y) \to Rep\ (\mathsf{F}\ a)\ x \to Rep\ (\mathsf{F}\ a)\ y \\
pmap\ (a :: a)\ f = fmap\ (\bot :: \mu\ (\mathsf{F}\ a))\ f \\
pzip :: Functor\ (\mathsf{F}\ a) \Rightarrow a \to (x \to y) \to (Rep\ f\ x, Rep\ f\ y) \to Rep\ f\ (x, y) \\
pzip\ (a :: a)\ f = fzip\ (\bot :: \mu\ (\mathsf{F}\ a))\ f
\end{array}$$

Here, $\bot$ represents the undefined value that is a member of every type in the category of Haskell programs (see the discussion in Section 7.1). Nevertheless, it is only used as a type annotation for the Haskell compiler, and the $fmap$ and $fzip$ are non-strict in their first argument; since Haskell is a lazy language, the value will never be evaluated.

Associated with each user-defined data type a, we have two unique functions inn (because in is a Haskell keyword for let expressions) and out that are each other inverse and witness the isomorphism between a and the fixed point of its base functor $\mathsf{F}\ a$. They allow us to encode and inspect values of the given type, respectively. For example, lists have the following instance:

$$\begin{array}{l}
\mathbf{class}\ Mu\ a\ \mathbf{where} \\
\quad inn :: Rep\ (\mathsf{F}\ a)\ a \to a \\
\quad out :: a \to Rep\ (\mathsf{F}\ a)\ a \\
\mathbf{instance}\ Mu\ [a]\ \mathbf{where} \\
\quad inn\ (Left\ \_) = [\,] \\
\quad inn\ (Right\ (x, xs)) = x : xs \\
\quad out\ [\,] = Left\ \bot \\
\quad out\ (x : xs) = Right\ (x, xs)
\end{array}$$

Here, instead of using the Haskell standard unit type $()$, we define the unit type as a datatype One with no constructors and $\bot$ as its sole value. For a discussion on this encoding see the implementation of the Pointless Haskell library in (Cunha, 2005, Chapter 6).

Encoding recursive lenses We now have the required machinery to encode the fold and unfold recursion patterns for arbitrary inductive types. Like the formal notation for recursion patterns, that are annotated with a subscript functor to which the argument function should be instantiated, their encodings require an explicit annotation of the recursive type:

$$\begin{aligned}
&cata :: (Mu\ a, Functor\ (\mathsf{F}\ a)) \Rightarrow a \to (Rep\ (\mathsf{F}\ a)\ b \to b) \to a \to b\\
&cata\ a\ f = f\ .\ pmap\ a\ (cata\ a\ f)\ .\ out\\
&ana :: (Mu\ b, Functor\ (\mathsf{F}\ b)) \Rightarrow b \to (a \to Rep\ (\mathsf{F}\ b)\ a) \to a \to b\\
&ana\ b\ f = inn\ .\ pmap\ b\ (ana\ b\ f)\ .\ f
\end{aligned}$$

Lens combinators are also defined directly from their formal definitions presented in Chapter 4. For example, we encode the lens fold combinator as follows:

$$\begin{aligned}
&cata\_lns :: (Mu\ a, Functor\ (\mathsf{F}\ a)) \Rightarrow a \to (Lens\ (\mathsf{F}\ a\ b)\ b) \to Lens\ a\ b\\
&cata\_lns\ a\ l = Lens\ get'\ put'\ create'\\
&\quad \mathbf{where}\ get' \quad = cata\ a\ (get\ l)\\
&\qquad\qquad put' \quad = \mathbf{let}\ g = put\ l\ .\ (id >\!< pmap\ a\ get')\ /\!\backslash\ \pi_2\\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \ ana\ a\ (pzip\ a\ create'\ .\ g\ .\ (id >\!< out))\\
&\qquad\qquad create' = ana\ a\ (create\ l)
\end{aligned}$$

It is now possible to write Haskell recursive programs in a truly point-free style. For instance, some of the lenses over lists presented in Chapter 4 can be directly transcribed to Haskell as follows:

$$\begin{aligned}
&length\_lns :: a \to Lens\ [a]\ Nat\\
&length\_lns\ a = cata\_lns\ \bot\ (id\_lns\ +\ snd\_lns\ ((pnt\ a)\ .\ bang))\\
&map\_lns :: Lens\ c\ a \to Lens\ [c]\ [a]\\
&map\_lns\ f = cata\_lns\ \bot\ (id\_lns\ +\ f\ \times\ id\_lns)
\end{aligned}$$

Here, pnt and $bang$ are point-free combinators denoting points $\underline{\cdot}$ and bang !, while $\times$ and $+$ are the product and sum lens combinators.

6.1.3 Delta lenses

Similarly to regular state-based lenses, a horizontal delta lens (Chapter 5) can be encoded in Haskell as a record of six functions: the three functions for state-based lenses plus three additional horizontal deltas:

$$\begin{aligned}
&\mathbf{data}\ DLens\ s\ a\ v\ b = DLens\ \{\\
&\quad get \qquad :: s\ a \to v\ b\\
&\quad , get_{\blacktriangle} \quad :: s\ a \to Delta\ (v\ b)\ (s\ a)\\
&\quad , put \qquad :: (v\ b, s\ a) \to Delta\ (v\ b)\ (v\ b) \to s\ a
\end{aligned}$$

```
  ,put▲    :: (v b, s a) → Delta (v b) (v b) → Delta (s a) (v b, s a)
  , create :: v b → s a
  ,create▲ :: v b → Delta (s a) (v b)}
```

The state-based functions transform between polymorphic Haskell source and target types s and v, applied to data elements of types a and b. As dependent functions, the horizontal deltas $get_{\blacktriangle}$, $put_{\blacktriangle}$ and $create_{\blacktriangle}$ receive the same input values as state-based transformations, and return deltas from output to input positions. Deltas (partial relations) are encoded as sets of pairs. Since Haskell is a non-dependently typed language, we encode the types of positions in source and view values as plain integers:

```
type Delta a b = Rel Int Int
type Rel a b = Set (a, b)
```

As an example, the embedding of a horizontal delta lens into a normal lens, by resorting to a difference operation that returns a delta between the modified and original view values, can be implemented as follows:

```
type Diff v = Shapely v ⇒ v → v → Delta v v
embed_lns :: Shapely v ⇒ Diff (v b) → DLens s a v b → Lens (s a) (v b)
embed_lns diff l = Lens (get l) put′ (create l)
  where put′ (v, s) = put l (v, s) (diff v (get l s))
```

The *Shapely* type class implements the shapely type operations for polymorphic inductive data types (we use the name *data_* because **data** is a reserved Haskell keyword):

```
class Shapely t where
  shape   :: t a → t One
  data_   :: t a → Rel Int a
  recover :: (t One, Rel Int a) → t a
```

6.2 A Strategic Lens Library

Modeling real-world problems in a functional language typically leads to a large set of recursive data types, each with many different constructors. That is the case, for example, when developing language processing tools, where grammars are represented using a different data type for each non-terminal and a different constructor for each production rule. Similarly, schema-aware XML processing usually involves mapping a huge schema to an equivalent data type, with each of the many elements mapped to a different type. Such proliferation of data types makes it hard to implement even conceptually simple functions (or lenses), that manipulate a very small subset of the data constructors. To illustrate this problem, remember the XML schema from Figure 1.3 modeling a movie database with information about several shows and actors. A part of this schema could be represented by the hereunder Haskell data type. In Haskell, left-nested products and sums correspond to `xs:sequence` and `xs:choice` elements in XML Schema notation. Primitives include base types, such as the unit type 1, integers Int or strings String, and lists $[A]$ of values of type A that model XML sequences (represented in XML Schema as unbounded $\mathit{maxOccurs}$ repetitions). User-defined data types model XML elements and attributes:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Imdb} = \mathit{Imdb}\ [\mathit{Either}\ \mathit{Movie}\ \mathit{Series}]\ [\mathit{Actor}] \\
\mathbf{data}\ \mathit{Movie} = \mathit{Movie}\ \mathit{Year}\ \mathit{Title}\ [\mathit{Review}]\ \mathit{Director}\ \mathit{Boxoffice} \\
\mathbf{data}\ \mathit{Series} = \mathit{Series}\ \mathit{Year}\ \mathit{Title}\ [\mathit{Review}]\ [\mathit{Season}] \\
\mathbf{data}\ \mathit{Actor} = \mathit{Actor}\ \mathit{Name}\ [\mathit{Played}] \\
\mathbf{data}\ \mathit{Name} = \mathit{Name}\ \mathit{String} \\
\mathbf{data}\ \mathit{Played} = \mathit{Played}\ \mathit{Year}\ \mathit{Title}\ \mathit{Role}\ [\mathit{Award}] \\
\mathbf{data}\ \mathit{Role} = \mathit{Role}\ \mathit{String} \\
\mathbf{data}\ \mathit{Award} = \mathit{Award}\ \mathit{Name}\ \mathit{Result} \\
\mathbf{data}\ \mathit{Comment} = \mathit{Comment}\ \mathit{String} \\
\ldots
\end{array}
$$

Suppose one also wants to define a lens to collect all names of awards won by a certain actor. Below is a possible definition of this lens using the combinators from the previous section:

$$
\begin{array}{l}
\mathit{awardnames} :: \mathit{Lens}\ \mathit{Actor}\ [\mathit{Name}] \\
\mathit{awardnames} = \mathit{map\_lns}\ (\mathit{snd\_lns}\ \mathit{dresult} \circ \mathit{out\_lns}) \circ \mathit{concat\_lns} \\
\quad \circ\ \mathit{map\_lns}\ (\mathit{snd\_lns}\ \mathit{drole} \circ \mathit{snd\_lns}\ \mathit{dtitle} \circ \mathit{snd\_lns}\ \mathit{dyear} \circ \mathit{out\_lns}) \\
\quad \circ\ \mathit{snd\_lns}\ \mathit{dname} \circ \mathit{out\_lns} \\
\qquad \mathbf{where}\ (\mathit{dresult}, \mathit{drole}) = (\mathit{pnt}\ (\mathit{Result}\ \texttt{""})\ .\ \mathit{bang}, \mathit{pnt}\ (\mathit{Role}\ \texttt{""})\ .\ \mathit{bang}) \\
\qquad\qquad\quad (\mathit{dtitle}, \mathit{dyear}) = (\mathit{pnt}\ (\mathit{Title}\ \texttt{""})\ .\ \mathit{bang}, \mathit{pnt}\ (\mathit{Year}\ 0)\ .\ \mathit{bang}) \\
\qquad\qquad\quad \mathit{dname} = \mathit{pnt}\ (\mathit{Name}\ \texttt{""})\ .\ \mathit{bang}
\end{array}
$$

Even this rather simple definition is filled with boilerplate code, whose only purpose is to perform a standard traversal of the $Actor$ data type to find names to collect. Apart from aesthetical reasons, this kind of boilerplate has some major drawbacks: (1) it makes code understanding rather difficult, since the only interesting functions are lost amid bucketloads of uninteresting code, including in our case default parameters for the backward direction of the lens; (2) and it rapidly becomes a maintenance problem, since each schema evolution implies changes to many functions that are only concerned with parts of the schema not affected by the evolution.

Many generic strategic programming libraries have been proposed to address this issue for functional programs. Most of them allow the user to specify the behavior just for the interesting bits of the structure, and provide traversal combinators to encode the remaining boilerplate. One of the most successful libraries is the conveniently named "Scrap you Boilerplate" (SYB), first introduced by Lämmel and Peyton Jones (2003) and subsequently extended with additional functionalities (Lämmel and Peyton Jones, 2005). Using this library, the forward direction of the $awardnames$ lens can be redefined as the following generic query:

$$
\begin{array}{l}
awardnames :: Actor \to [Name] \\
awardnames = everything\ (+\!\!+)\ (mkQ\ [\,]\ awname) \\
\quad \textbf{where}\ awname\ (Award\ name\ result) = [name]
\end{array}
$$

The $everything$ combinator traverses a data structure in a bottom-up fashion, applying its argument generic query to all nodes and collecting the results using the given append operation. The mkQ combinator builds a generic query from a type-specific one: given an empty value of type b and a function $f :: b \to r$, it behaves like f for all values of type b and returns the empty value otherwise. Besides being much easier to understand what the query does, its definition will stay the same even if the $Actor$ data type changes.

The same rationale applies to type-level transformations. Imagine that we want to delete all actor roles, according to the following lens transformation:

$$
\begin{array}{l}
deleteroles :: Lens\ Actor\ NewActor \\
deleteroles = inn\_lns \circ (id\_lns\ \times\ map\_lns\ played) \circ out\_lns \\
\quad \textbf{where}\ played = inn\_lns \circ (id\_lns\ \times\ (id\_lns\ \times\ role)) \circ out\_lns \\
\qquad\qquad\ role \quad\ = snd\_lns\ (pnt\ (Role\ \texttt{""})\ .\ bang) \\
\textbf{data}\ NewActor = NewActor\ Name\ [NewPlayed]
\end{array}
$$

$$\mathbf{data}\ \mathit{NewPlayed} = \mathit{NewPlayed}\ \mathit{Year}\ \mathit{Title}\ [\mathit{Award}]$$

Again, we have to write a transformation that traverses the source schema until it finds roles to delete. Also, this time we have to encode the result types *a priori*, since Haskell is a typed language and the lens transformation must be defined for specific source and target types. By developing a two-level transformation framework in the style of (Cunha et al., 2006a), we could overcome both limitations and specify a transformation like the following:

$$\begin{array}{l} \mathit{deleteroles} :: \mathit{Type} \to \mathit{Type} \\ \mathit{deleteroles} = \mathit{everywhereT}\ (\mathit{tryT}\ (\mathit{atT}\ \texttt{"Role"}\ \mathit{eraseT})) \end{array}$$

When applied to a representation of the Actor data type, this two-level transformation between arbitrary schema representations, written in our pseudo two-level language, would return a dynamically generated $\mathit{NewActor}$ data type with the new structure and a corresponding lens transformation between both types. Here, $\mathit{everywhereT}$ is a type-level combinator that traverses the argument type and applies its argument transformation to all subtypes, deleting those with name "Role". The remaining combinators will be later introduced and explained.

In this section, we discuss solutions for the above examples in the context of strategic bidirectional lenses. In particular, we construct a language of strategic two-level bidirectional transformations for schema abstraction, whose underlying value-level transformations are total well-behaved lenses. To tackle the $\mathit{awardnames}$ and $\mathit{deleteroles}$ examples, our proposed language will enable the specification of two-level abstraction transformations: as generic value-level queries that collect values of a specific type inside a source database; and as generic type traversals that apply type-level transformations at arbitrary levels inside a schema. Unlike (Cunha et al., 2006a), our two-level language is tailored for data abstraction rather than for data refinement, and supports the traversal and transformation of recursive types.

6.2.1 Representing Types and Expressions

In order to enable the type-safe rewriting of types and lenses (for the execution of two-level lens transformations and their subsequent optimization), we need representations of types, functions and lenses at the value level. For this purpose, instead of the *shallow embedding* used in our lens library from the previous section, we make use of a *deep* *embedding* (Wildmoser and Nipkow, 2004) of types and point-free expressions, where the objects and arrows of our lens language are encoded as Haskell data types.

As proposed by Cunha and Visser (2011), our representations of types and functors use *generalized algebraic data types* (GADTs) (Jones et al., 2006):

```
data Type a where
   Int    :: Type Int
   One    :: Type One
   ...
   Prod   :: Type a → Type b → Type (a, b)
   Either :: Type a → Type b → Type (Either a b)
   List   :: Type a → Type [a]
   Data   :: (Mu a, Functor (F a)) ⇒ String → Fctr (F a) → Type a
   Fun    :: Type a → Type b → Type (a → b)
   Lns    :: Type a → Type b → Type (Lens a b)
data Fctr (f :: ∗ → ∗) where
   I      :: Fctr Id                              -- Identity
   L      :: Fctr []                              -- List functor
   K      :: Type c → Fctr (Const c)              -- Constant
   (⊗)    :: Fctr f → Fctr g → Fctr (f ⊗ g)       -- Product
   (⊕)    :: Fctr f → Fctr g → Fctr (f ⊕ g)       -- Sum
   (⊙)    :: Fctr f → Fctr g → Fctr (f ⊙ g)       -- Application
```

These definitions provide type-safe representations for base types, products, sums, lists, user-defined types, functions, lenses, and polynomial functors (extended with the list type functor)[2]. Note how the a parameter of the $Type\ a$ GADT constrains the type representation to values of type a. For example, the value $Prod\ Int\ Int$ represents the type (Int, Int). For user-defined types, the $Fctr$ value in the definition of $Data$ is not arbitrary, and we use the based functor of the user-defined type a. For example, $Data$ `"Nat"` $(K\ One \oplus \mathsf{I})$ is the representation of the type of natural numbers. As a ubiquitous structure in functional programming and XML programming, it is useful in practice to treat the type of lists as a primitive instead of representing it as a user-defined type with the base functor $K\ One \oplus K\ a \otimes \mathsf{I}$. The data type presented in the beginning of this section to model the XML Schema from Figure 1.3 can be represented as

[2] It is easy to generalize this encoding to support representations of polymorphic data types and regular bifunctors, by introducing a new constructor for type functors in $Fctr$ and a new $BiFctr$ GADT.

$$\begin{array}{l} \mathit{Data}\ \texttt{"imdb"}\ ((L \odot (K\ (\mathit{Data}\ \texttt{"movie"}\ \mathit{fm}) \oplus K\ (\mathit{Data}\ \texttt{"series"}\ \mathit{fs}))) \\ \qquad\qquad \otimes (L \odot K\ (\mathit{Data}\ \texttt{"actor"}\ \mathit{fa}))) \end{array}$$

where the variables *fm*,*fs* and *fa* denote the base functors of the `movie`, `series` and `actor` elements.

We also define a function *rep* that applies a functor to a type, returning a flat type representation:

$$\begin{array}{l} \mathit{rep} :: \mathit{Fctr}\ f \to \mathit{Type}\ a \to \mathit{Type}\ (\mathit{Rep}\ f\ a) \\ \mathit{rep}\ \mathsf{I}\ a = a \\ \mathit{rep}\ (f \otimes g)\ a = \mathit{Prod}\ (\mathit{rep}\ f\ a)\ (\mathit{rep}\ g\ a) \\ \mathit{rep}\ (f \oplus g)\ a = \mathit{Either}\ (\mathit{rep}\ f\ a)\ (\mathit{rep}\ g\ a) \\ \ldots \end{array}$$

Point-free expressions can also be represented in a type-safe manner using a GADT:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathbf{data}\ \mathit{Pf}\ f\ \mathbf{where}} \\ \quad \mathit{Id} & :: & \mathit{Pf}\ (c \to c) \\ \quad \mathit{Fst} & :: & \mathit{Pf}\ ((a, b) \to a) \\ \quad \mathit{Bang} & :: & \mathit{Pf}\ (a \to \mathit{One}) \\ \quad \ldots & & \\ \quad \mathit{IdLns} & :: & \mathit{Pf}\ (\mathit{Lens}\ c\ c) \\ \quad \mathit{CompLns} & :: & \mathit{Type}\ b \to \mathit{Pf}\ (\mathit{Lens}\ b\ a) \to \mathit{Pf}\ (\mathit{Lens}\ c\ b) \to \mathit{Pf}\ (\mathit{Lns}\ c\ a) \\ \quad \mathit{FstLns} & :: & \mathit{Pf}\ (a \to b) \to \mathit{Pf}\ (\mathit{Lens}\ (a, b)\ a) \\ \quad \mathit{CataLns} & :: & (\mathit{Mu}\ a, \mathit{Functor}\ (\mathsf{F}\ a)) \\ & \Rightarrow & \mathit{Pf}\ (\mathit{Lens}\ (\mathit{Rep}\ (\mathsf{F}\ a)\ b)\ b) \to \mathit{Pf}\ (\mathit{Lens}\ a\ b) \\ \quad \mathit{MapLns} & :: & \mathit{Pf}\ (\mathit{Lens}\ a\ b) \to \mathit{Pf}\ (\mathit{Lens}\ [a]\ [b]) \\ \quad \ldots & & \end{array}$$

Note that the inhabitants of type $\mathit{Pf}\ f$ are the point-free representations of both unidirectional functions and bidirectional lenses. Some typical operations over lists such as mapping are also treated as primitive lenses.

One consequence of using a GADT to encode point-free expressions is the ability to define an evaluation function. For example, the evaluation of the ${\pi_1}^f$ lens calls its definition from our lens library and depends on the evaluation of its default function f:

$$\begin{array}{l} \mathit{eval} :: \mathit{Type}\ a \to \mathit{Pf}\ a \to a \\ \mathit{eval}\ (\mathit{Lns}\ (\mathit{Prod}\ a\ b)\ \_)\ (\mathit{FstLns}\ f) = \mathit{fst\_lns}\ (\mathit{eval}\ (\mathit{Fun}\ a\ b)\ f) \\ \ldots \end{array}$$

Moreover, a GADT allows to simulate a reflexion mechanism that provides the ability to define a type equality function, and is nowadays a classical example of the usefulness of GADTs (Jones et al., 2006):

```
data Equal a b where Eq :: Equal a a
teq :: Type a → Type b → Maybe (Equal a b)
teq Int Int = return Eq
teq (Prod a b) (Prod c d) = do {Eq ← teq a c; Eq ← teq b d; return Eq}
...
teq _ _ = Nothing
```

The constructor *Eq* of the *Equal* GADT can be seen as a proof that the types a and b are indeed equal.

6.2.2 Combinators for Two-level Lenses

In our framework, two-level data transformations are formalized in terms of lenses and can be modeled in Haskell as sequences of type-changing rewrite rules that replace one data type with another (Cunha et al., 2006a).

Encapsulating lenses as rewrite rules A lens defines a particular abstract view of a more concrete type, as defined below:

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

When encoding lenses as rewrite rules, it is essential that they are type-preserving to guarantee type-safe rewriting, but views are type-changing in nature (since they ignore some details from the original format). To resolve this tension, they are masqueraded as type-preserving ones inside the *View* constructor, but similar functions can be implemented to, after rewriting, release them out of their type-preserving shell and obtain the corresponding type-changing bidirectional transformations. Note that only the concrete type c escapes from the *View c* context, while the abstract type a remains encapsulated and is existentially quantified. Also, we do not keep a lens transformation *Lens c a* but a representation of a lens *Pf (Lens c a)*, to allow later inspection and optimization.

A rule generalizes a view 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:

$$\mathbf{type}\ RuleT = \forall c.\ Type\ c \rightarrow RuleM\ (View\ c)$$

$RuleM$ is a stateful monad that keeps a context of applied rules and dynamically generated types and is an instance of the $MonadPlus$ Haskell standard type class, thus modeling partiality in rule application: $return$ denotes successful rule application; failure is signaled with $mzero$; and $mplus$ encodes optional rule application.

Strategic combinators for two-level lenses The core of our rewrite system is built using strategic term rewriting (Lämmel, 2003), where the combination of a standard set of basic rules allows the simple design of flexible rewriting strategies in a compositional style. Some standard strategic combinators for identity ($nopT$), sequential composition ($\rhd$) and left-biased choice ($\|$) are defined below and, derived from these, other combinators such as optional rule application ($tryT$) and repetition ($manyT$) can be encoded:

$$\begin{aligned}
&nopT :: RuleT\\
&nopT\ x = return\ (View\ IdLns\ x)\\
&(\rhd), (\|) :: RuleT \rightarrow RuleT \rightarrow RuleT\\
&(r \rhd s)\ a = \mathbf{do}\ View\ f\ b \leftarrow r\ a\\
&\qquad\qquad\qquad View\ g\ c \leftarrow s\ b\\
&\qquad\qquad\qquad return\ (View\ (CompLns\ b\ g\ f)\ c)\\
&(r \parallel s)\ x = r\ x\ \text{`}mplus\text{`}\ s\ x\\
&tryT, manyT :: RuleT \rightarrow RuleT\\
&tryT\ r = r \parallel nopT\\
&manyT\ r = tryT\ (r \rhd manyT\ r)
\end{aligned}$$

While this small set of combinators suffices for applying transformations at a single level, combinators that descend into the structure of types are more challenging to define. Like (Lämmel and Peyton Jones, 2003) and other strategic programming languages, instead of defining generic traversals by induction on the structure of types, we define a small set of traversal combinators. One such combinator that traverses the functorial structure of types is $allT$, by applying an argument rule to all immediate children of a data type:

```
allT :: RuleT → RuleT
allT r One = return (View IdLns One)
allT r (List a) = do View f b ← r a
                     return (View (MapLns f) (List b))
allT r t@(Data s f) = do
  ViewF l g ← allF r f
  new ← newData s g
  return (View (CataLns (CompLns (rep g new) InnLns (l new))) new)
...
allF :: RuleT → RuleF
allF r I = return (ViewF (λt → IdLns) I)
allF r L = return (ViewF (λt → IdLns) L)
allF r (K a) = do View l b ← r a
                  return (ViewF (λt → l) (K b))
allF r (f ⊗ g) = do ViewF l₁ h ← allF r f
                    ViewF l₂ i ← allF r g
                    return (ViewF (λt → ProdLns (l₁ t) (l₂ t)) (h ⊗ i))
...
```

The behavior of *allT* for user-defined types is the most interesting: it invokes the auxiliary rule *allF* that propagates a rule application down to the constants of the base functor of the data type, where it applies the argument rule, and returns a natural lens transformation as a rewrite rule *RuleF* on functor representations; then, it constructs a bottom-up fold lens that recursively applies the lens transformation to all children of the recursive type. Note that such recursive fold lens is always well-behaved due to its particular form (Chapter 4). A natural lens transformation defines a view of a functor, and is encapsulated by a type *ViewF*:

```
type NatLens f g = ∀a. Type a → Pf (Lens (Rep f a) (Rep g a))
data ViewF f where
  ViewF :: (Functor f, Functor g) ⇒ NatLens f g → Fctr g → ViewF f
type RuleF = ∀f. Fctr f → RuleM (ViewF f)
```

As long as the application of a rule may change the structure of a type, we need to be able to dynamically construct new types during rewriting. For this purpose, the monadic function $\mathit{newData} :: \mathit{String} \to \mathit{Fctr}\ f \to \mathit{RuleM}\ (\mathit{Type}\ (\mu\ f))$ receives a type name and a functor representation and returns a new unique type representation. Although not shown in the code of *allT*, if the base functor is not modified we return the original type. The *RuleM* monad is responsible for keeping track of new types and preserving name uniqueness. Also, it allows *newData* to avoid generating duplicate types whenever another new type with the same name and functor already exists. Newly generated types must also be added to our type representation GADT as fixed points of functors:

```
data Type a where
  ...
  NewData :: Functor f ⇒ String → Fctr f → Type (μ f)
```

Note how, unlike user-defined data types, a new type is not an instance of the *Mu* class, because there exists no corresponding data type in the context of the transformation. While the *Data* constructor allows users to perform rewriting over user-defined types, the *NewData* constructor provides the necessary support for representing the results of such transformations.

Using *allT*, we can define the derived *everywhereT* combinator that traverses a type representation in a bottom-up fashion, applying the given rule to all its descendants:

```
everywhereT :: RuleT → RuleT
everywhereT r = allT (everywhereT r) ▷ r
```

Another strategic type traversal is *onceT*, that applies a given rule exactly once somewhere inside a type representation at an arbitrary depth, by traversing the type in a top-down approach:

```
onceT :: RuleT → RuleT
onceT r ▪ = mzero
onceT r (List a) = r (List a) `mplus` (do
  View l b ← onceT r a
  return (View (MapLns f) (List b))
onceT r a@(Data s f) = r a `mplus` (do
  ViewF l b ← onceF r f
  new ← newData s g
  return (View (AnaLns (CompLns (rep f a) (l a) OutLns)) new)
...
```

```
onceF :: RuleT → RuleF
onceF r I = mzero
onceF r L = do View l ga ← r (rep L ■)
               FRep g ← inferFctr ■ ga
               return (ViewF (λt → l) g)
onceF r (f ⊗ g) =
  do View l ha ← r (rep (f ⊗ g) ■)
     FRep h ← inferFctr ■ ha
     return (ViewF (λt → l) h)
  `mplus` do ViewF l h ← onceF r f
             return (ViewF (λt → ProdLns (l t) IdLns) (h ⊗ g))
  `mplus` do ViewF l i ← onceF r g
             return (ViewF (λt → ProdLns IdLns (l t)) (f ⊗ i))
...
```

The *onceT* traversal stops as soon as its argument rule can be applied successfully. This means that, for user-defined types, it does not necessarily descend into the constants of the base functor, but applies the rule anywhere inside the functor representation. This more intricate behavior is crucial to enable the application of type rewriting rules at particular positions inside user-defined types that are not constants of the base functor of the type. This will be used, for example, to identify lists of child elements in our application scenarios from Section 6.4. To be able to apply normal type rules inside a functor, the auxiliary rule *onceF* flattens the functor by applying it to a special type marker ■ that denotes a kind of uninstantiated type variable:

```
data Type a where
  ...
  ■ :: Type a
```

When the argument rule can be successfully applied, it infers a new functor representation from the target type using ■ to remember the recursive invocations. Such type-functor unification is implemented by the procedure $\mathit{inferFctr} : \mathit{MonadPlus}\ m \Rightarrow \mathit{Type}\ a \to \mathit{Type}\ b \to m\ (\mathit{FctrRep}\ a\ b)$. Here, the $\mathit{FctrRep}\ a\ b$ wrapper encapsulates a representation of a functor f together with a proof that f applied to the marker type a is actually equal to the flattened type b[3]:

[3] In Haskell, type contexts can include equality constraints (Schrijvers et al., 2007) of the form $t_1 \sim t_2$ which entail that the types t_1 and t_2 must be equal after reduction, i.e., application of type functions.

$$
\begin{array}{l}
\mathbf{data}\ \mathit{FctrRep}\ a\ b\ \mathbf{where} \\
\quad \mathit{FRep} :: (\mathit{Functor}\ f, \mathit{Rep}\ f\ a\ \sim\ b) \Rightarrow \mathit{Fctr}\ f \to \mathit{FctrRep}\ a\ b
\end{array}
$$

For recursive types, the resulting lens performs a top-down traversal (unfold) that applies the value-level transformations of the argument rule once for each recursive element. Again, note that this recursive unfold lens is always well-behaved due to its shape (Chapter 4).

The $\mathit{outermostT}$ traversal performs top-down exhaustive rule application and is defined at the cost of onceT:

$$
\begin{array}{l}
\mathit{outermostT} :: \mathit{RuleT} \to \mathit{RuleT} \\
\mathit{outermostT}\ r = \mathit{manyT}\ (\mathit{onceT}\ r)
\end{array}
$$

Local combinators for two-level lenses To control the application of certain rules when using generic traversal combinators, it is useful to identify locations inside type representations. For that purpose, we define the whenT combinator that applies an argument rule whenever a generic type-level predicate is satisfied, or fails otherwise:

$$
\begin{array}{l}
\mathbf{type}\ \mathit{PredicateT} = \forall a.\ \mathit{Type}\ a \to \mathit{Bool} \\
\mathit{whenT} :: \mathit{PredicateT} \to \mathit{RuleT} \to \mathit{RuleT} \\
\mathit{whenT}\ p\ r\ a = \mathbf{do}\ \{\mathit{guard}\ (p\ a); r\ a\}
\end{array}
$$

These type-level predicates can be seen as patterns that represent specific type structures. As an example, we can define the atP predicate (and the corresponding atT combinator) that succeeds when the name of the current type matches a given name:

$$
\begin{array}{l}
\mathit{atP} :: \mathit{String} \to \mathit{PredicateT} \\
\mathit{atP}\ \mathit{name}\ (\mathit{Data}\ n\ f) = \mathit{sameName}\ \mathit{name}\ n \\
\mathit{atP}\ \mathit{name}\ \_ = \mathit{False} \\
\mathit{atT} :: \mathit{String} \to \mathit{RuleT} \to \mathit{RuleT} \\
\mathit{atT}\ \mathit{name}\ r = \mathit{whenT}\ (\mathit{atP}\ \mathit{name})\ r
\end{array}
$$

Here, the $\mathit{sameName}$ function checks if two names are equal according to our representation. Due to the proliferation of dynamically generated types imposed by some of our strategic combinators, we assume that different types always have unique identifier names (managed by the RuleM monad) but may represent similar element names (think of the encoding of two distinct XML elements with the same name).

Other combinators for processing user-defined types inspired by *Focal* (Foster et al., 2007) are: $hoistT$ that unpacks a user-defined type by applying out at the value-level; $plungeT$ that constructs a new (non-recursive) user-defined type by applying in at the value-level; and $renameT$ that renames an existing user-defined type, issuing the id lens. Notice that, for recursive types, $hoistT$ followed by $plungeT$ is different from $renameT$ since there is no general method to "tie the recursive knot". Despite the latter would simply rename the type, the former would unpack the recursive type once and return a new non-recursive top-level type with the same name.

Abstraction combinators for two-level lenses Our library of two-level lenses also supports specific abstraction combinators. To delete part of a schema, we simply call $eraseT$ at the appropriate location: it deletes the current top-level type, by replacing it with the unit type:

$$\begin{array}{l} eraseT :: RuleT \\ eraseT\ a = return\ (View\ (BangLns\ (Pnt\ (defvalue\ a)))\ One) \end{array}$$

At the value level, $eraseT$ applies the $!^{f}$ lens, where $f :: One \to a$ is a function that generates a default value of the erased type a. For this purpose, we introduce a function $defvalue :: Type\ a \to a$ that generates a default value for an inductive type representation and define $f = \underline{defvalue\ a}$.

So far, our combinators build generic transformations that describe the explicit changes that are performed on the source type. An alternative way to specify generic programs is to perform queries that traverse arbitrary structures to collect values of a specific type, as the $everything$ combinator presented at the beginning of this section. A generic query returning values of type r can be encoded in Haskell as a function with type $Q\ r = \forall a.\ Type\ a \to a \to r$. A class of such functions will be presented in Section 6.3. We incorporate generic queries as two-level bidirectional rewrite rules in two steps: we first specialize the query for the specific input type to which it is being applied, and then try to lift the specialized point-free function into a point-free lens:

$$\begin{array}{l} liftQ :: Type\ r \to Pf\ (Q\ r) \to RuleT \\ liftQ\ r\ q\ a = \mathbf{do\ let}\ f = reduce\ optimize\_query\ (Fun\ a\ r)\ (ApplyQ\ a\ q) \\ \qquad\qquad\qquad lns \leftarrow lensify\ (Fun\ a\ r)\ f \\ \qquad\qquad\qquad return\ (View\ lns\ r) \end{array}$$

In this code, the $ApplyQ$ point-free combinator converts a generic query into a regular function by applying it to a specific type, and the $optimize\_query$ strategy specializes the generic point-free function to a non-generic point-free function. More details on the specialization and optimization of point-free expressions will be provided later in Section 6.3. To lift the resulting functions into lenses, the $lensify$ combinator checks if their point-free expressions are defined using only the point-free lens combinators from Chapter 4, otherwise rule application fails. For lenses requiring additional parameters such as π_1,π_2, ! and $f \triangledown g$, default values and left-biased choices are assumed.

Given a lens transformation between specific types, it can be lifted to a two-level rewrite rule as follows:

$$\begin{array}{l} liftLns :: Type\ c \to Type\ a \to Pf\ (Lens\ c\ a) \to RuleT \\ liftLns\ c\ a\ lns\ t = \mathbf{do}\ Eq \leftarrow teq\ c\ t \\ \qquad\qquad\qquad\qquad\quad return\ (View\ lns\ b) \end{array}$$

To be able to apply the lens, the rule must test if the input type t matches the concrete type c of the lens. This combinator is particularly useful for extending the set of primitive two-level combinators with user-defined lenses. It also allows users to write more refined alignment-aware code for specific pieces of a larger two-level transformation, using the delta lenses from Chapter 5.

Unleashing the value-level lenses So far, we have camouflaged type-changing rewrite strategies as type-preserving transformations by using the $View$ constructor. Consequently, we cannot use the resulting lenses directly (unless we know the target types *a priori*) because the target type is statically undefined, and is only available after executing the type-level transformation. To overcome this issue, we export the resulting view type and the corresponding value-level lens to a new Haskell module that can be compiled and executed independently:

$$generateHaskell :: [FilePath] \to Type\ a \to RuleT \to IO\ ()$$

The $generateHaskell$ function receives a list of input Haskell modules to import (where the original source types shall be defined), a source type representation and a two-level transformation, and writes as output to a new Haskell module, containing an optimized lens transformation (by applying the rewrite system from the next section) and dynamically generated data type declarations (and conforming instances) for all the elements of the target type.

6.3 A Point-free Rewriting Library

A known disadvantage of strategic programs is their worse performance in comparison to analogous non-strategic ones. For example, Mitchell and Runciman (2007) have reported that the SYB implementation of a standard set of benchmark functions runs 7 times slower in average than the non-strategic implementation. Part of this performance loss is due to the run-time checks needed to determine at each node whether to apply specific or generic behavior. The remaining is due to structural reasons inherent to this style: the traversal combinators must blindly traverse the whole data structure, even if a certain branch does not mention types where the specific behavior applies.

Similarly, after executing a two-level transformation, the composed value-level transformations still traverse whole input data structures. The solution used in the *2LT* framework (Visser, 2008) was to develop a rewrite system for the optimization of the value-level point-free functions using algebraic point-free laws (Cunha and Visser, 2007). After rewriting, the resulting transformations work directly between the source and target types, by fusing and cutting redundant traversals, and are significantly more efficient. A similar rewrite system was used to optimize generic value-level functions, by adding laws for the specialization of generic functions over specific types into non-optimized point-free definitions (Cunha and Visser, 2011).

However, the major drawback of both approaches was the lack of support for user-defined recursive types. In this section, we extend this point-free rewrite system to also cover arbitrary recursive data types. As particular contributions, we show how to specialize generic traversal combinators for recursive types using well-known recursion patterns and propose a technique to mechanize the challenging fusion laws for recursion patterns, that traditionally require a “guessing” step.

Moreover, we extend the system to support the rewriting of point-free lenses. Of course, optimization could be attempted independently at the three components of a lens (as in the *2LT* framework (Visser, 2008)), since they are also defined in the point-free style, but this would lead to a much more expensive optimization of the three components separately, allied to the fact that the complexity of the put function prevents the automatic spotting of many optimization opportunities. We ended up with a mixed approach: the bulk of optimizations is performed directly at the lens level (namely, all fusions involving recursion patterns), with some minor optimizations involving the unpacking of “opaque” lens combinators performed later at each component separately.

6.3.1 Specializing Generic Queries

Generic value-level programs come in two flavors: type-preserving transformations that preserve the type of the input and type-unifying queries that return values of a specific type. In this thesis, we will focus on the latter. Generic queries, that traverse arbitrary data structures to collect values of a specific type, constitute typical examples of transformations that abstract a source type to keep only particular concrete information, and are a good candidate high-level language for expressing generic structure-shy lens programs. In addition to alleviating the usual inefficiency issue that plagues generic programs, the specialization and subsequent optimization of generic queries is instrumental for their successful bidirectionalization into lenses. Remembering the $liftQ$ two-level combinator from Section 6.2, we convert a generic query into a lens by first specializing it from its high-level form into a point-free expression, that can be further simplified using the point-free calculus, and then try to syntactically match the optimized point-free expression with our language of point-free lenses from Chapter 4. In Section 6.4, we will use this same technique to lensify XPath queries. Conversely, type-preserving programs hardly constitute good examples of lens transformations. For completeness, the specialization for this class of programs can be found in (Cunha and Pacheco, 2011).

The following set of combinators captures the essence of generic type-unifying queries:

$$
\begin{array}{lll}
\varnothing & : Q\ R & \text{-- empty result} \\
(\cup) & : Q\ R \to Q\ R \to Q\ R & \text{-- union of results} \\
gmapQ & : Q\ R \to Q\ R & \text{-- fold over children} \\
everything & : Q\ R \to Q\ R & \text{-- fold over every node} \\
mkQ_A & : (A \to R) \to Q\ R & \text{-- creation} \\
apQ_A & : Q\ R \to (A \to R) & \text{-- composition}
\end{array}
$$

As defined above, $Q\ R$ represents the type of generic queries with result type R. To simplify the specialization laws, we will assume that R is a monoid, with a $zero : R$ element and an associative $plus : R \times R \to R$ operator. In practice, this makes little difference since most typical result types, namely lists and integers, are indeed monoids. In the SYB library, the type $Q\ r$ is defined as $\forall a.\ Data\ a \Rightarrow a \to r$. Type classes like $Data$ are extensively used in SYB to infer type representations for data types. Among others, these are necessary in the definition of mkQ to determine where the

$$apQ_A\ \varnothing = \underline{zero} \circ ! \qquad \varnothing\text{-APPLY}$$

$$apQ_A\ (f \cup g) = plus \circ (apQ_A\ f \vartriangle apQ_A\ g) \qquad \cup\text{-APPLY}$$

$$\left.\begin{array}{l} apQ_A\ (gmapQ\ f) = \underline{zero} \circ !\,, \textbf{if}\ A\ base\ type \\ apQ_{A \times B}\ (gmapQ\ f) = plus \circ (apQ_A\ f \times apQ_B\ f) \\ apQ_{A+B}\ (gmapQ\ f) = apQ_A\ f \triangledown apQ_B\ f \\ apQ_{\mu\mathsf{F}}\ (gmapQ\ f) = apQ'_{\mathsf{F}\ \mu\mathsf{F}}\ f \circ out_{\mathsf{F}} \end{array}\right\} \qquad gmapQ\text{-APPLY}$$

$$apQ_A\ (everything\ f) = apQ_A\ (f \cup gmapQ\ (everything\ f)) \qquad everything\text{-APPLY}$$

$$\left.\begin{array}{l} apQ_A\ (mkQ_A\ f) = f \\ apQ_A\ (mkQ_B\ f) = \underline{zero} \circ !\,, \textbf{if}\ A \not\equiv B \end{array}\right\} \qquad mkQ\text{-APPLY}$$

$$\left.\begin{array}{l} apQ'_{\mathsf{F}\ A}\ f = fplus_{\mathsf{F}}, \textbf{if}\ A \equiv R \\ apQ'_{\mathsf{F}\ A}\ f = fplus_{\mathsf{F}} \circ \mathsf{F}\ apQ_A\ f, \textbf{if}\ A \not\equiv R \end{array}\right\} \qquad \text{FUNCTOR-APPLY}$$

Figure 6.1: Laws for the specialization of generic queries.

type-specific transformation should be applied, and the heavy use of type-classes to infer type representations is in fact a large part of SYB's inefficiency. To factor out this penalty and simplify the presentation, instead of type classes we will parameterize mkQ with an explicit type representation. Besides $everything$, that traverses a data structure in a bottom-up fashion to apply a query to all nodes while collecting the results using the $plus$ operation, $gmapQ$ collects the results of applying a query to all direct children, $\cup$ sums the results of two queries, and $\varnothing$ denotes the query that always returns $zero$. To apply a generic query to a particular type we have the explicit combinator apQ. As an example, the $awardnames$ generic query from the previous section can now be redefined as follows:

$$\begin{array}{l} awardnames :: Actor \to [Name] \\ awardnames = apQ_{Actor}\ (everything\ (mkQ_{Award}\ awname)) \\ \quad \textbf{where}\ awname\ (Award\ name\ result) = [name] \end{array}$$

Figure 6.1 presents the laws used to specialize type-unifying combinators into point-free (assuming that R denotes the result type). Specialization proceeds by pushing down the apQ combinator until it gets consumed by the mkQ-APPLY law. Although not generic, the definitions produced by this specialization are very inefficient because they still traverse the whole data structure. However, using point-free program calculation laws they can be optimized in order to eliminate redundant traversals. The auxiliary $apQ\prime$ combinator, used by $gmapQ$ for user-defined types, behaves like apQ but applies the argument query inside a functor. The $fplus$ function generalizes the $plus$ monoid operation and folds an arbitrary regular functor into a monoid result, as follows:

$$
\begin{aligned}
& fplus_{\mathsf{F}} : \mathsf{F}\ R \to R \\
& fplus_{\mathsf{Id}} = id \\
& fplus_{\underline{A}} = \underline{zero} \circ ! \\
& fplus_{\mathsf{F} \otimes \mathsf{G}} = plus \circ (fplus_{\mathsf{F}} \times fplus_{\mathsf{G}}) \\
& fplus_{\mathsf{F} \oplus \mathsf{G}} = fplus_{\mathsf{F}} \triangledown fplus_{\mathsf{G}} \\
& fplus_{\mathsf{F} \odot \mathsf{G}} = fplus_{\mathsf{F}} \circ \mathsf{F}\ fplus_{\mathsf{G}} \\
& fplus_{T} = \llparenthesis fplus_{\mathsf{B}\ R} \rrparenthesis_{\mathsf{B}\ R}
\end{aligned}
$$

Notice that the $everything$-APPLY law describes the recursive definition of the $everything$ traversal combinator using $gmapQ$ and $\cup$. Since the *2LT* framework (Visser, 2008) only handled non-recursive user-defined data types, this law did not pose any termination problems. However, it cannot be used for recursive types because it would lead to an infinite expansion of the definition, due to successive expansions of $everything$ in recursive occurrences of the type.

The key to avoid infinite expansions is to specialize traversal combinators using an alternative definition based on recursion patterns. When applied to an inductive type, the bottom-up $everything$ traversal will be specialized into a paramorphism over that type:

$$
\begin{gathered}
apQ_{\mu\mathsf{F}}\ (everything\ f) = \\
\langle\!| plus \circ (apQ\prime_{\mathsf{F}\ \widehat{R}}\ (everything\ f) \times apQ_{\mu\mathsf{F}}\ f) \circ (\mathsf{F}\ \pi_1 \vartriangle in_{\mathsf{F}} \circ \mathsf{F}\ \pi_2) |\!\rangle_{\mathsf{F}}
\end{gathered}
$$

The behavior of this paramorphism is better understood with the help of the following diagram:

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[rr, "out_{\mathsf{F}}"] \arrow[d, "\langle\!|\cdot|\!\rangle_{\mathsf{F}}"'] & & \mathsf{F}\ \mu\mathsf{F} \arrow[rr, "\mathsf{F}\ (id \vartriangle id)"] & & \mathsf{F}\ (\mu\mathsf{F} \times \mu\mathsf{F}) \arrow[d, "\mathsf{F}\ (\langle\!|\cdot|\!\rangle_{\mathsf{F}} \times id)"] \\
R & R \times R \arrow[l, "plus"] & & \mathsf{F}\ R \times \mu\mathsf{F} \arrow[ll, "apQ\prime_{\mathsf{F}\ \widehat{R}}\ (everything\ f) \times apQ_{\mu\mathsf{F}}\ f"] & \mathsf{F}\ (R \times \mu\mathsf{F}) \arrow[l, "\mathsf{F}\ \pi_1 \vartriangle in_{\mathsf{F}} \circ \mathsf{F}\ \pi_2"]
\end{tikzcd}$$

The intent of the function $apQ\prime_{\mathsf{F}\ \widehat{R}}\ (everything\ f)$ is to apply the query to all content of the functor, apart from its recursive occurrences (which were already processed recursively by the paramorphism itself). This behavior is achieved by adding the following law to the set presented in Figure 6.1:

$$apQ_{\widehat{A}}\ f = \underline{zero} \circ ! \qquad \textit{rec}\text{-}\textsc{Apply}$$

This law guarantees that a typed marked with a curved line is ignored by the apQ combinator. For example, for lists of type A the expression $apQ\prime_{\mathsf{F}\ \widehat{R}}\ (everything\ f)$ would be instantiated to $apQ\prime_{(\underline{1} \oplus \underline{A} \otimes \mathsf{Id})\ \widehat{R}}\ (everything\ f)$, which is equivalent after simplification to $(\underline{zero} \circ !\,) \bigtriangledown apQ_A\ (everything\ f) \circ \pi_1$. Note that $apQ\prime_{\mathsf{Id}\ \widehat{R}}\ f = \underline{zero} \circ !$ since $R \not\equiv \widehat{R}$.

After recursion, the input value is reconstructed using in_{F} in order to feed it to the generic query. Simultaneously, the query is applied to the non-recursive type contents, and finally both are put together with the monoid $plus$ operator.

To exemplify the specialization of a generic query over a recursive type, consider the following example where $f = mkQ_{Award}\ awname : Q\ [Name]$ and $\mathsf{F} = \mathsf{List}_{Award}$ (implementation-wise we consider a special case for lists):

$$
\begin{array}{l}
apQ_{[Award]}\ (everything\ f) \\
\ = \{\, everything\text{-}\textsc{Apply} \,\} \\
\langle\!| plus \circ (apQ\prime_{\mathsf{F}\ \widehat{[Name]}}\ (everything\ f) \times apQ_{[Award]}\ f) \circ (\mathsf{F}\ \pi_1 \bigtriangleup in_{\mathsf{F}} \circ \mathsf{F}\ \pi_2) |\!\rangle_{\mathsf{F}} \\
\ = \{\, mkQ\text{-}\textsc{Apply};\ \textsc{Functor}\text{-}\textsc{Apply};\ rec\text{-}\textsc{Apply};\ plus\text{-}\textsc{Zero}; \times - \textsc{Cancel} \,\} \\
\langle\!| (apQ_1\ (everything\ f) \bigtriangledown plus \circ (apQ_{Award}\ (everything\ f) \times id)) \circ \mathsf{F}\ \pi_1 |\!\rangle_{\mathsf{F}} \\
\ = \{\, everything\text{-}\textsc{Apply}; \cup\text{-}\textsc{Apply};\ gmapQ\text{-}\textsc{Apply};\ plus\text{-}\textsc{Zero} \,\} \\
\qquad apQ_{Award}\ (gmapQ\ (everything\ f)) \\
\qquad\ = \{ ... \} \\
\qquad \underline{zero} \circ ! \\
\langle\!| (apQ_1\ f \bigtriangledown plus \circ (plus \circ (apQ_{Award}\ f \bigtriangleup \underline{zero} \circ !\,) \times id)) \circ \mathsf{F}\ \pi_1 |\!\rangle_{\mathsf{F}} \\
\ = \{\, mkQ\text{-}\textsc{Apply};\ plus\text{-}\textsc{Zero} \,\} \\
\langle\!| (\underline{zero} \circ ! \bigtriangledown plus \circ (awname \times id)) \circ \mathsf{F}\ \pi_1 |\!\rangle_{\mathsf{F}} \\
\ = \{\, \langle\!| \cdot |\!\rangle - \textsc{Cata} \,\} \\
(\!| \underline{zero} \circ ! \bigtriangledown plus \circ (awname \times id) |\!)_{\mathsf{F}}
\end{array}
$$

As expected, the result is empty if f is applied to a type that does not contain awards:

$$
\begin{array}{l}
apQ_{[Comment]}\ (everything\ f) \\
\ = \{ ... \}
\end{array}
$$

$$\begin{array}{l}
\llparenthesis (\underline{\mathit{zero}} \circ ! \triangledown \mathit{plus} \circ (\mathit{apQ}_{\mathit{Comment}}\ f \times \mathit{id})) \rrparenthesis_{\mathsf{List}_{\mathit{Comment}}} \\
= \{ \mathit{mkQ}\text{-}\textsc{Apply} \} \\
\llparenthesis \underline{\mathit{zero}} \circ ! \triangledown \pi_2 \rrparenthesis_{\mathsf{List}_{\mathit{Comment}}} \\
= \{ \llparenthesis \cdot \rrparenthesis\text{-}\textsc{Zero} \} \\
\quad (\underline{\mathit{zero}} \circ ! \triangledown \pi_2) \circ (\underline{1} \oplus \underline{\mathit{Comment}} \otimes \mathsf{Id})\ (\underline{\mathit{zero}} \circ !\,) = \underline{\mathit{zero}} \circ ! \\
\quad \Leftrightarrow \{ \textsc{Functor-Def}; +\text{-}\textsc{Absor}; \times - \textsc{Def}; \times - \textsc{Cancel} \} \\
\quad \underline{\mathit{zero}} \circ ! \triangledown \underline{\mathit{zero}} \circ ! \circ \pi_2 = \underline{\mathit{zero}} \circ ! \\
\quad \Leftrightarrow \{ ! - \textsc{Fusion}; +\text{-}\textsc{Zero} \} \\
\quad \textsc{True} \\
\underline{\mathit{zero}} \circ !
\end{array}$$

6.3.2 Mechanizing Fusion

The main difference between equational reasoning and term rewriting (Baader and Nipkow, 1998) is that bidirectional equations of the form $f = g$ are adapted into unidirectional rewrite rules of the form $f \leadsto g$ (read f leads to g), indicating that a term f can be substituted by a term g, but not otherwise. For the goal of simplification, the general idea is to substitute terms by simpler terms (for most cases). In our point-free rewrite system, this corresponds to viewing the point-free equational laws presented throughout this thesis as rules oriented from left-to-right.

When rewriting transformations over recursive types, the most effective optimizations are often enabled by the fusion laws for recursion patterns, like $\llparenthesis \cdot \rrparenthesis$-FUSION and $[\![\cdot]\!]$-FUSION, that allow collapsing successive data traversals into a single one. However, implementing the full power of these laws in an algebraic rewrite system is a challenging task, since it implies "guessing" the algebra (or coalgebra) of the resulting recursion pattern. To be more specific, remember the fold fusion law:

$$f \circ \llparenthesis g \rrparenthesis_{\mathsf{F}} = \llparenthesis h \rrparenthesis_{\mathsf{F}} \Leftarrow f \circ g = h \circ \mathsf{F}\ f$$

Reading this law as a rewrite rule, in order to perform the reduction $f \circ \llparenthesis g \rrparenthesis_{\mathsf{F}} \leadsto \llparenthesis h \rrparenthesis_{\mathsf{F}}$, one must compute a function h such that $f \circ g = h \circ \mathsf{F}\ f$ holds. For this reason, fusion laws for specific operations over lists or the following instance of the fold fusion law used in (Cunha and Pacheco, 2011) are especially useful because they avoid the hard guessing step:

$$(\!|f|\!)_{\mathsf{F}} \circ (\!|in_{\mathsf{F}} \circ g|\!)_{\mathsf{F}} = (\!|f \circ g|\!)_{\mathsf{F}} \quad \Leftrightarrow \quad \mathsf{F}\ (\!|f|\!)_{\mathsf{F}} \circ g = g \circ \mathsf{F}\ (\!|f|\!)_{\mathsf{F}}$$

Unfortunately, we cannot always avoid the need to use general fusion, and thus some technique must be implemented in order to mechanize it. This research topic has received some attention in the past: one of the most successful implementations is the MAG system (Sittampalam and de Moor, 2003), which views the guessing step as a *higher-order matching* problem. However, MAG is not fully automatic and thus not suitable for our optimization system: the user must have some idea of the steps of the proof to provide sufficient hints to proceed with the derivation.

We reduce the hard guessing step to a simple rewriting problem that, although not as general as MAG, is fully automatic and works in practice for many examples. In the above fold fusion law, if the converse of f could be computed as f°, then h could be trivially defined as $f \circ g \circ \mathsf{F}\ f^{\circ}$. Of course, this is just an alternative formulation of the guessing step and useless *per se*. However, if f° is left opaque (just denoting the tagging of the expression f), and by applying our standard rewrite system, temporarily augmented with the rule $f \circ f^{\circ} \leadsto id$, we manage to get rid of f°, then we get the desired algebra. This idea is embodied in the following rewrite rule, where we test that f° does not occur in the normal form of $f \circ g \circ \mathsf{F}\ f^{\circ}$:

$$f \circ (\!|g|\!)_{\mathsf{F}} \leadsto (\!|h|\!)_{\mathsf{F}} \Leftarrow f \circ g \circ \mathsf{F}\ f^{\circ} \overset{*}{\leadsto} h \wedge f^{\circ} \notin h \qquad (\!|\cdot|\!)\text{-Fusion}$$

Our technique is in a way similar to the point-wise fusion algorithm proposed by Sheard and Fegaras (1993). The same technique can be used for mechanizing laws over anamorphisms and paramorphisms, with rules $f^{\circ} \circ f \leadsto id$ and $f \circ f^{\circ} \leadsto id$, respectively:

$$\begin{aligned} [\![g]\!]_{\mathsf{F}} \circ f \leadsto [\![h]\!]_{\mathsf{F}} &\Leftarrow \mathsf{F}\ f^{\circ} \circ g \circ f \overset{*}{\leadsto} h \wedge f^{\circ} \notin h && [\![\cdot]\!]\text{-Fusion} \\ f \circ \langle\!|g|\!\rangle_{\mathsf{F}} \leadsto \langle\!|h|\!\rangle_{\mathsf{F}} &\Leftarrow f \circ g \circ \mathsf{F}\ (f^{\circ} \times id) \overset{*}{\leadsto} h \wedge f^{\circ} \notin h && \langle\!|\cdot|\!\rangle\text{-Fusion} \\ \langle\!|f|\!\rangle_{\mathsf{F}} \leadsto (\!|g|\!)_{\mathsf{F}} &\Leftarrow f \circ \mathsf{F}\ {\pi_1}^{\circ} \overset{*}{\leadsto} g \wedge {\pi_1}^{\circ} \notin g && \langle\!|\cdot|\!\rangle\text{-Cata} \end{aligned}$$

For an example of this technique, consider the following trace demonstrating the optimization of a composed lens over lists (indentation in the trace indicates the rewriting of side-conditions):

$length \circ filter\_left$

$$\begin{array}{l}
\rightsquigarrow \{\, \mathit{filter\_left} - \textsc{Def};\ (\![\cdot]\!)\text{-}\textsc{Fusion};\ +\text{-}\textsc{Functor-Comp};\ +\text{-}\textsc{Functor-Id} \,\} \\
\quad \mathit{length} \circ (\mathit{in}_{\mathsf{List}_A} \mathbin{\bullet\!\nabla} {\pi_2}^{\underline{b} \circ\, !}) \circ \mathit{coassocl} \circ (\mathit{id} + \mathit{distl}) \circ (\mathit{id} + \mathit{id} \times \mathit{length}^{\circ})) \\
\quad \rightsquigarrow \{\, +\text{-}\textsc{Functor-Comp};\ +\text{-}\textsc{Functor-Id}^{-1};\ \mathit{distl} - \textsc{Nat} \,\} \\
\quad \mathit{length} \circ (\mathit{in}_{\mathsf{List}_A} \mathbin{\bullet\!\nabla} {\pi_2}^{\underline{b} \circ\, !}) \circ \mathit{coassocl} \\
\quad \circ\, (\mathit{id} + (\mathit{id} \times \mathit{length}^{\circ} + \mathit{id} \times \mathit{length}^{\circ})^{(\pi_1 \times \mathit{put}_{\mathit{length}^{\circ}}) \circ \mathit{distp}, (\pi_1 \times \mathit{put}_{\mathit{length}^{\circ}}) \circ \mathit{distp}}) \\
\quad \circ\, (\mathit{id} + \mathit{distl}) \\
\quad \rightsquigarrow \{\, \mathit{coassocl} - \textsc{Nat};\ + - \textsc{Absor} \,\} \\
\quad \mathit{length} \circ ((\mathit{in}_{\mathsf{List}_A} \circ (\mathit{id} + \mathit{id} \times \mathit{length}^{\circ}) \mathbin{\bullet\!\nabla} {\pi_2}^{\underline{b} \circ\, !} \circ (\mathit{id} \times \mathit{length}^{\circ})) \circ \mathit{coassocl} \\
\quad \circ\, (\mathit{id} + \mathit{distl}) \\
\quad \rightsquigarrow \{\, + -\textsc{Fusion};\ \mathit{length}\text{-}\textsc{Def};\ (\![\cdot]\!) - \textsc{Cancel};\ +\text{-}\textsc{Functor-Comp} \\
\qquad ;\ \times -\textsc{Functor-Comp} \,\} \\
\quad (\mathit{in}_{\mathsf{Nat}} \circ (\mathit{id} + {\pi_2}^{!} \circ (\mathit{id} \times \mathit{length} \circ \mathit{length}^{\circ})) \mathbin{\bullet\!\nabla} \mathit{length} \circ {\pi_2}^{\underline{b} \circ\, !} \circ (\mathit{id} \times \mathit{length}^{\circ})) \\
\quad \circ\, \mathit{coassocl} \circ (\mathit{id} + \mathit{distl}) \\
\quad \rightsquigarrow \{\, \times - \textsc{Functor-Comp};\ \mathit{length} \circ \mathit{length}^{\circ} \rightsquigarrow \mathit{id} \,\} \\
\quad (\mathit{in}_{\mathsf{Nat}} \circ (\mathit{id} + {\pi_2}^{!}) \mathbin{\bullet\!\nabla} \mathit{length} \circ {\pi_2}^{\underline{b} \circ\, !} \circ (\mathit{id} \times \mathit{length}^{\circ})) \circ \mathit{coassocl} \circ (\mathit{id} + \mathit{distl}) \\
\quad \rightsquigarrow \{\, \pi_2 - \textsc{Nat};\ \mathit{length} \circ \mathit{length}^{\circ} \rightsquigarrow \mathit{id} \,\} \\
\qquad \mathit{create}_{\mathit{id}} \circ \underline{b} \circ\, ! \circ \mathit{get}_{\mathit{length}^{\circ}} \\
\qquad \rightsquigarrow \{\text{definition of } \mathit{create};\ ! - \textsc{Uniq}\} \\
\qquad \underline{b} \circ\, ! \\
\quad (\mathit{in}_{\mathsf{Nat}} \circ (\mathit{id} + {\pi_2}^{!}) \mathbin{\bullet\!\nabla} {\pi_2}^{\underline{b} \circ\, !}) \circ \mathit{coassocl} \circ (\mathit{id} + \mathit{distl}) \\
(\![(\mathit{in}_{\mathsf{Nat}} \circ (\mathit{id} + {\pi_2}^{!}) \mathbin{\bullet\!\nabla} {\pi_2}^{\underline{b} \circ\, !}) \circ \mathit{coassocl} \circ (\mathit{id} + \mathit{distl})]\!)_{\mathsf{List}_{A+B}}
\end{array}$$

To make the presentation clear, in this trace we mention the inverse of some rules (denoting the respective laws oriented from right-to-left). Obviously, to ensure termination, these rules are not encoded as such in our rewrite system. Instead, we have generalized versions that cover additional cases such as the following for *distl*-NAT (the definitions of x and y can be easily computed, but are omitted to simplify the presentation):

$$\mathit{distl} \circ (\mathit{id} \times f) \rightsquigarrow (\mathit{id} \times f + \mathit{id} \times f)^{x,y} \circ \mathit{distl}$$

$$\mathit{distl} \circ ((f + g) \circ h \times i) \rightsquigarrow (f \times i + g \times i)^{x,y} \circ \mathit{distl} \circ (h \times \mathit{id})$$

6.3.3 Encoding a Point-free Rewrite System

A straightforward method to optimize the generic programs presented in this chapter and the lenses from Chapter 4 would be to specify the point-free laws described in this thesis as GHC rewrite rules (Jones et al., 2001), and allow their use by the GHC compiler. However, such an approach provides little control over the rewrite strategy and is not capable of implementing laws such as *gmapQ*-Apply, !-Uniq or $(\!|\cdot|\!)$-Fusion, since it does not support type-directed rewriting nor side-conditions. In order to harness the full power of our algebraic laws, we instead recover a successful type-preserving, type-directed rewrite system for the transformation of point-free programs (Cunha and Visser, 2011) and extend it to support recursive types and bidirectional lenses.

Point-free rewrite system and rules In our implementation, point-free rewrite rules are represented by monadic type-preserving functions that receive a type representation and a point-free expression and return a new expression of the same type:

```
type Rule = ∀f. Type f → Pf f → RewriteM (Pf f)
```

The *RewriteM* monad fulfills the same task of the *RuleM* for two-level transformations: the monadic function *success* updates the *RewriteM* monad to keep trace of successful reductions, while failure is signaled with *mzero*. The extra type representation passed as an argument to the rule allows it to make type-based rewriting decisions. For example, we can encode the !-Uniq law for lenses as follows:

```
bang_uniq_lns :: Rule
bang_uniq_lns (Lns _ _) (BangLns f) = mzero
bang_uniq_lns (Lns a One) l = do
  let createl = createof (Lns a One) l
  g ← optimise_fun (Fun One a) createl
  success "!-Uniq-Lns" (BangLns g)
bang_uniq_lns _ _ = mzero
```

The first case of this rule avoids a rewriting loop (application of !-Uniq to ! itself. The third catch-all case indicates that the rule fails for any other input. The second case reveals the two-layered architecture of our rewrite system: the strategy *optimise_fun* simplifies function representations, of the form $Pf\ (a \rightarrow b)$, and the strategy *optimise_lens* rewrites lens representations, of the form $Pf\ (Lens\ a\ b)$. To mediate between these two classes, the procedures *getof*,*createof* and *putof* take the representation of a lens and return the representations of the corresponding *get*,*put* and *create* functions. As a general methodology, whenever a unidirectional function is created inside a lens rule, we apply *optimise_fun* to simplify it.

A more intricate rule involving side-conditions is the encoding of $(\!|\cdot|\!)$-CATA, instrumental for a successful conversion of specialized generic queries into lenses (since paramorphisms perform implicit duplication and are not supported by our lens language, despite many cases degenerate into catamorphisms that we support):

```
para_cata :: Rule
para_cata (Fun a@(Data _ fctr) b) (Para f) = do
  let (fb, fba) = (rep fctr b, rep fctr (Prod b a))
      g' = Comp fba f (Fmap fctr (Fun b (Prod b a)) (Conv Fst))
  g ← optimise_fun (Fun fb b) g'
  guard (not (findConv (Pf (Fun fb b)) g))
  success "para-Cata" (Cata g)
para_cata _ _ = mzero
```

Put quickly, the rule composes the argument of the paramorphism with the converse of π_1 and simplifies the composed function (considering the special rule for eliminating converses). After the intermediate rewriting step, the rule uses *findConv* to check if the resulting expression no longer contains temporary converses, and if successful it returns the simplified point-free expression as the argument of a catamorphism. This method is more general than pattern matching directly on the expression $f \circ \mathsf{F}\ \pi_1$.

Our point-free rewrite system is built by re-instantiating a standard set of type-preserving strategic combinators (Lämmel, 2003). Namely, the combinators ⧁, ⊘ *nop*, *try*, *once* and *outermost* encode sequential composition, choice, identity, optional top-level rule application, single rule application at an arbitrary depth and exhaustive top-down rule application. Using these strategic combinators, we can construct our three main strategies: *optimize_query* for the specialization of type-unifying generic programs into point-free functions; and *optimize_fun* and *optimize_lens* for the simplification and optimization of point-free expressions. The last is defined as follows:

```
optimize_lens = outermost opt ⧁ rec
  where opt  = id_nat_lns ⊘ bang_uniq_lns ⊘ ...
        rec  = try (once fuse ⧁ optimize_lens)
        fuse = cata_fusion_lns ⊘ ana_fusion_lns ⊘ ...
```

This strategy exhaustively applies the set of rewrite rules for lenses described across this thesis. Some of these rules (particularly fusion rules) are evidently more expensive,

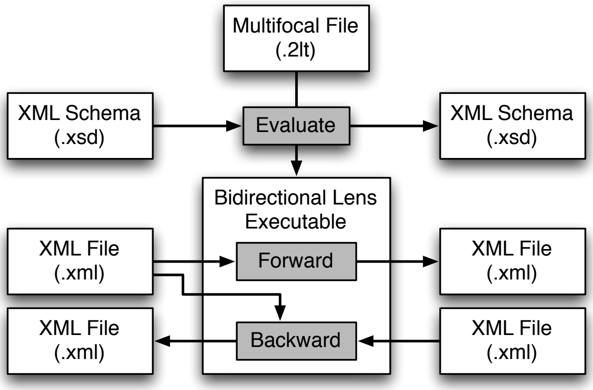


Figure 6.2: Architecture of the *Multifocal* framework.

due to the intermediate rewriting of side-conditions, and are deferred inside the strategy until no other rule can be applied.

6.4 *Multifocal*: A Strategic Bidirectional Transformation Language for XML Schemas

The previous sections have described our Haskell libraries for the specification, execution and optimization of two-level lens transformations over algebraic data types. In this section, we provide an overview of the *Multifocal* language and the corresponding framework for the two-level transformation of XML Schemas.

The general architecture of our framework is illustrated in Figure 6.2. A two-level transformation defined as a *Multifocal* program is executed in two stages: first, it is translated into our strategic lens library and evaluated as a type-level transformation by applying it to a source XML Schema, producing a target XML Schema and a bidirectional lens; second, the lens is compiled into a Haskell executable file that can be used to propagate updates between XML documents conforming to the source and target schemas. In our scenario, optimization is done at the second stage: we can optimize the value-level lenses once for each input schema and generate optimized executables that efficiently propagate updates between XML documents. We discuss particular examples involving the generic evolution of recursive XML Schemas, and compare their performance gains over non-optimized definitions.

6.4.1 Language

Our two-level XML transformation language is defined by embedding the strategic lens combinators from Section 6.2 for the transformation of XML Schemas, together with specific XML transformers. The full syntax of *Multifocal* is defined as follows:

$$
\begin{array}{rcl}
\mathit{strat} & ::= & \mathtt{nop} \mid \mathit{strat} \gg \mathit{strat} \mid \mathit{strat}\ \mathtt{||}\ \mathit{strat} \mid \mathtt{many}\ \mathit{strat} \mid \mathtt{try}\ \mathit{strat} \\
& \mid & \mathtt{all}\ \mathit{strat} \mid \mathtt{once}\ \mathit{strat} \mid \mathtt{everywhere}\ \mathit{strat} \mid \mathtt{outermost}\ \mathit{strat} \\
& \mid & \mathtt{at}\ \mathtt{'"'}\ \mathit{tag}\ \mathtt{'"'}\ \mathit{strat} \mid \mathtt{when}\ \mathtt{'"'}\ \mathit{tag}\ \mathtt{'"'}\ \mathit{strat} \\
& \mid & \mathtt{hoist} \mid \mathtt{plunge}\ \mathtt{'"'}\ \mathit{tag}\ \mathtt{'"'} \mid \mathtt{rename}\ \mathtt{'"'}\ \mathit{tag}\ \mathtt{'"'} \\
& \mid & \mathtt{erase} \mid \mathtt{select}\ \mathtt{'"'}\ \mathit{xpath}\ \mathtt{'"'}
\end{array}
$$

The strategic combinators (in the first two lines) simply apply the Haskell combinators from Section 6.2 over translated schema representations. The intuition for strategic traversals over XML Schemas is to descend into child elements in the sense of XPATH traversals. For example, the `all` combinator applies a transformation to all immediate children of the current schema element (for the `imdb` element from Figure 1.3, these would be all `movie`, `series` and `actor` elements).

The `at` combinator applies an argument rule if the name of the current element matches a given XML element tag. As in XPath, we will consider that XML node names preceded by an ampersat "@" denote attributes. On the other hand, `when` takes the name of an XML Schema element and performs the following pattern matching: if the given element name is defined as a top-level element in the source schema and its structure matches the structure of the current element (by converting its structure into type-level predicate that is passed to whenT), then it applies the argument rule; otherwise, rule application fails. The structure of the top-level element is converted into a type-level predicate by partially applying teq to the its type representation. Polymorphic types are encoded by extending our type representations with unbounded type variables and adapting teq to handle variable matching.

Other tag-based combinators inspired by *Focal* (Foster et al., 2007) are: `hoist` that untags the current element, `plunge` that names a new XML element and `rename` that renames the current element.

As a language for defining views of XML Schemas, *Multifocal* also supports specific abstraction combinators. We can delete part of a schema by calling `erase` at the appropriate location. A well-known feature of XML query languages like XPath is that they allow writing generic queries that traverse arbitrary documents to select particular nodes from XML documents without having to exhaustively specify intermediate element tags. To apply an XPath query to a schema, we invoke the `select` combinator: it internally converts the XPath expression into a point-free generic query that abstracts the schema into the desired result type; and invokes the $liftQ$ strategy that attempts to bidirectionalize the generic query into a lens.

6.4.2 Interface

We now unveil the implementation of the *Multifocal* framework. The embedding of the *Multifocal* language in our framework is supported by front-ends that translate XML Schemas and XML documents into Haskell types and values, and vice-versa. As exemplified in Section 6.2, these front-ends map XML Schemas to equivalent type-safe representations. A more technical description of similar XML-Haskell front-ends can be found in (Berdaguer et al., 2007).

Two-level transformations written in *Multifocal* are translated into our core library of strategic lens combinators (Section 6.2) that operate on Haskell type representations. After translating the source XML Schema into a Haskell type representation, the framework applies the type-level transformation to produce as output a target type and a lens representation as Haskell values. From these, it generates a target XML Schema and a Haskell executable file containing the lens transformation and the data type declarations that represent all the source and target XML elements. The main function of this file parses XML documents complying to the schemas, converts them into internal Haskell values, runs the lens transformation either in the forward or backward direction to propagate source-to-target or target-to-source updates, and finishes by pretty-printing an updated XML document.

Conversion of XPath queries As done in (Cunha and Visser, 2011; Ferreira and Pacheco, 2007), we convert XPath queries into generic point-free programs of type $Q\ [*]$ that return sequences of dynamic values. Since XPath result sets are untyped, $*$ works as a generic type wrapper and allows the encapsulation of multiple values of different types to be returned by the generic query (e.g., values of different XML elements with the same name but different base types). Using this conversion, followed by a successful "lensification" with $lensify$ (that attempts to unwrap result values of type $*$ when they are known to have the same type), our interpreter can specialize XPath queries into lenses over recursive XML schemas.

$$\begin{array}{lll}
{\pi_1}^{!} : A \times 1 \unrhd A & {\pi_2}^{!} : 1 \times A \unrhd A & \text{-- Products} \\
\mathit{id} \triangledown \mathit{nil} : [A] + 1 \unrhd [A] & \mathit{nil} \triangledown \mathit{id} : 1 + [A] \unrhd [A] & \text{-- Sums} \\
\mathit{filter\_left} : [A + 1] \unrhd [A] & \mathit{filter\_right} : [1 + A] \unrhd [A] & \text{-- Lists} \\
\mathit{id} \underset{\bullet}{\triangledown} \mathit{id} : A + A \unrhd A & \mathit{concat} : [[A]] \unrhd [A] & \text{-- Ambiguous types}
\end{array}$$

Figure 6.3: Rules for the normalization of XML Schema representations.

Schema normalization To keep a minimal suite of combinators, our language supports abstractions through the `erase` combinator, that deletes elements locally and thus leaves "dangling" ones in the target schema. However, these empty elements are unintended and may yield XML Schemas that are deemed ambiguous by many XML processors. For example, applying a transformation that erases `series` inside `imdb` elements to the IMDb schema from Figure 1.3 would result in a (flattened) list representation $\mathit{List}\ (\mathit{Either}\ (\mathit{Data}\ \texttt{"movie"}\ \mathit{fm})\ \mathit{One})$. Such dangling unit types have no representation in the XML side and must be deleted from the target schema representation. Such deletion is performed by a *normalize* procedure that removes these and other ambiguities, by exhaustively applying the lens transformations from Figure 6.3. Normalization is silently applied by extending the semantics of the `all` and `once` traversals (denoted by $[\![\cdot]\!]$) so that they apply *normalize* after rewriting:

$$\begin{array}{l}
[\![\texttt{all r}]\!] = \mathit{allT}\ [\![\texttt{r}]\!] \rhd \mathit{normalize} \\
[\![\texttt{once r}]\!] = \mathit{onceT}\ [\![\texttt{r}]\!] \rhd \mathit{normalize}
\end{array}$$

Lens optimization Although the lens transformations generated by our framework are instantiated for particular source and target schemas, they still contain many redundant computations and traverse the whole structures, as a consequence of being a two-level transformation. To improve their efficiency, we employ the rewriting library from Section 6.3 for the optimization of point-free lenses. supporting powerful laws for fusing and cutting redundant traversals. After rewriting, the resulting transformations work directly between the source and target types and are significantly more efficient, as demonstrated below. In our framework, we provide users with the option to optimize the generated bidirectional programs at the time of generation of the Haskell bidirectional executable, if they intend to repeatedly propagate updates between XML documents conforming to the same schemas. This could be the case, for example, when the schemas represent the configuration of a live system that replies to frequent requests.

In such cases, the once-a-time penalty of an additional optimization phase for a specific schema is amortized by a larger number of executions.

6.4.3 Application Scenarios

We now present two XML evolution scenarios and compare the performance of the lenses resulting from the execution of the two-level transformations with their automatically optimized definitions.

The first example consists in summarizing the information about movies and actors stored in the IMDb schema from Figure 1.3, according to the following *Multifocal* transformation:

```
everywhere (try (at "series" erase))
>> everywhere (try (at "movie" (
 outermost (when "reviews" (
  select "count(//comment)" >> plunge "@popularity"))
 >> outermost (when "boxoffices" (
  select "sum(//@value)" >> plunge "@profit")))))
>> everywhere (try (at "actor" (
 outermost (at "played" (
  select "award/@name" >> all (rename "awname"))))))
```

This transformation performs the following steps:

1. Delete all `series` elements by applying an `erase` (constrained by `at`) everywhere in the source schema;

2. For each `movie`, replace its `reviews` by a `popularity` attribute counting the number of `comments` and replace its `boxoffice` elements with a `profit` attribute summing the total `value` elements. These attributes are calculated using XPath queries (constrained by `when`) and tagged with `plunge`;

3. For each `actor`, keep his `name` and compute a list of `award names` using another XPath query. The *name* elements in the result sequence are renamed to `awname` by applying `rename` within the all traversal.

In this transformation, the `reviews` and `boxoffices` names used by the `when` combinator denote top-level XML Schema elements (Figure 6.4) that must be defined in the source XML Schema. They match the type representations $List\ (Data$ `"review"` $fr)$ and $List\ (Data$ `"boxoffice"` $fb)$, for arbitrary functors *fr* and *fb*, i.e., lists of

```
<xs:group name="reviews"><xs:sequence>
  <xs:element name="review" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence></xsd:group>
<xs:group name="boxoffices"><xs:sequence>
  <xs:element name="boxoffice" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence></xs:group>
```

Figure 6.4: XML Schema top-level elements modeling specific type patterns.

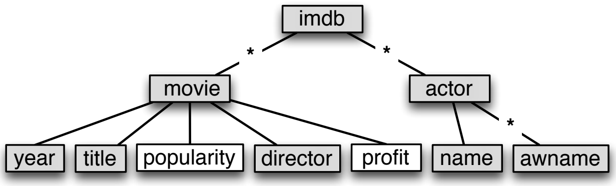


Figure 6.5: A view of the movie database schema from Figure 1.3.

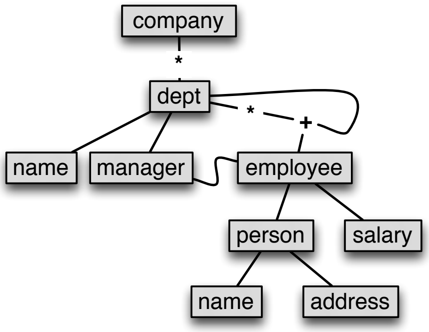


Figure 6.6: A company hierarchized payroll XML schema.

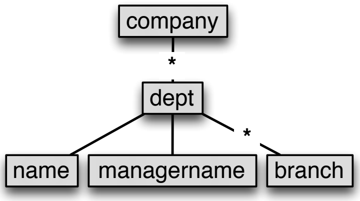


Figure 6.7: A view of the company schema.

elements named `review` and `boxoffice`, respectively. The resulting schema is depicted in Figure 6.5.

A classical schema used to demonstrate strategic programming systems is the so called "paradise benchmark" (Lämmel and Peyton Jones, 2003). Suppose one has a recursive XML Schema to model a company with several departments, each having a name, a manager and a collection of employees or sub-departments, illustrated in Figure 6.6. Our second evolution example consists in creating a view of this schema according to the following transformation:

```
everywhere (try (at "manager" (
 all (select "(//name)[1]") >> rename "managername")))
>> everywhere (try (at "employee" erase))
>> once (at "dept" (hoist >> outermost (at "dept" (
```

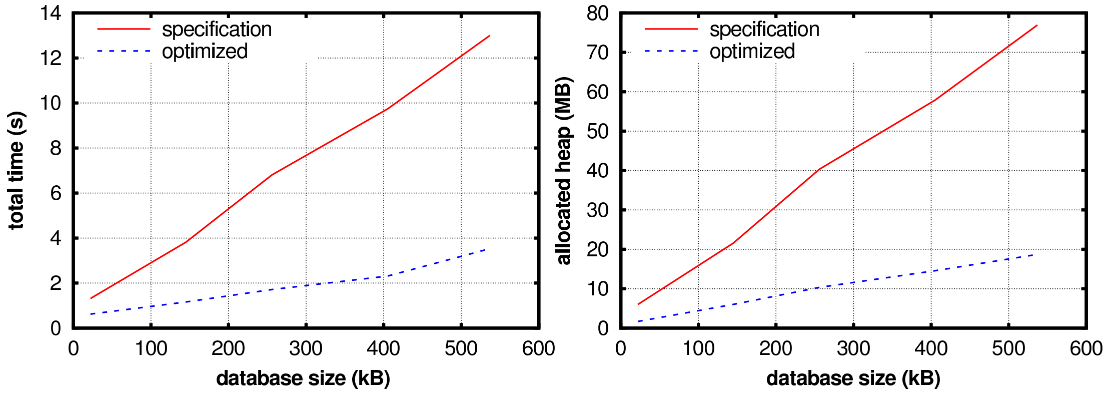


Figure 6.8: Benchmark results for the *IMDb* example.

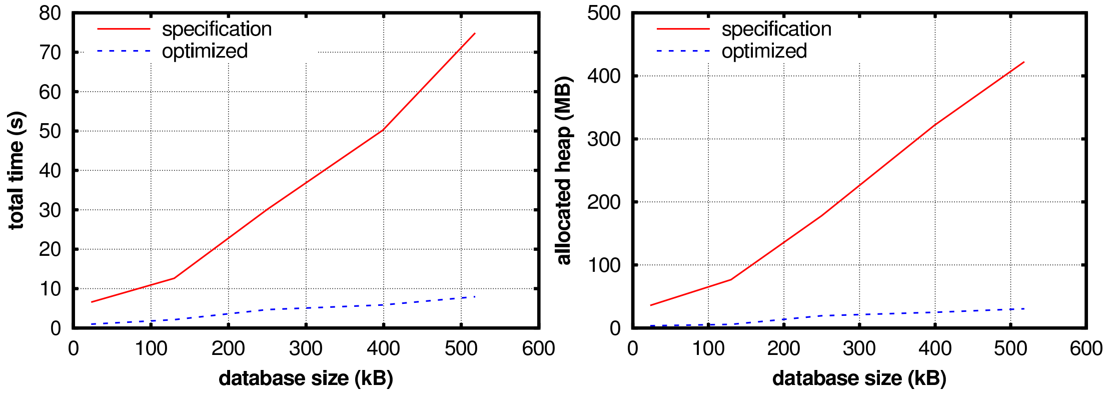


Figure 6.9: Benchmark results for the *paradise* example.

```
    select "name" >> rename "branch")) >> plunge "dept"))
```

For each top-level department, this transformations keeps only the `names` of `managers` (renamed to `managername`), deletes all `employees` and collects the names of direct sub-departments renamed to `branch`. The resulting non-recursive schema is depicted in Figure 6.7. There are some details worth noticing. First, it is easier to keep only manager names using a generic query instead of a transformation that would need to specify how to drop the remaining structure. The XPath filter "[1]" guarantees a sole result if multiple names existed under `managers`. Second, since `dept` is a recursive type, we unpack its top-level recursive structure once using `hoist` to be able to process sub-branches, and create a new non-recursive `dept` element with `plunge`.

Performance Analysis Unfortunately, as discussed in Section 6.3, the lenses resulting from the above two-level transformations are not very efficient. For instance, for the IMdb example the traversals over `series`, `movies` and `actors` are independent and can be done in parallel. Also, the transformations of `reviews` and `boxoffices` and the extra (hidden) normalizing step that filters out unit types (originating from erased `series` elements) can be fused into a single traversal. For the paradise example, all the three steps and the extra normalization step (for erased `employee` elements) can be fused into a single traversal. Also, the first two steps, that traverse all department values recursively due to the semantics of `all` (invoked by `everywhere`) for recursive types, are deemed redundant for sub-departments by the last step.

All these optimizations can be performed by an optional lens optimization phase. We have measured space and time consumption of the lenses generated by our two examples, and the results are presented in Figure 6.8 and Figure 6.9. Most of a lens' inefficiency comes from the complex synchronizing behavior of its put function. Therefore, to quantify the speedup achieved by the optimizations, we have compared the runtime behavior of put functions for non-optimized (`specification`) and optimized lens definitions (`optimized`). Note that the lens expressions corresponding to XPath queries are already optimized in the non-optimized lens, since their successful "lensification" depends on their specialization. To factor out the cost of parsing and pretty-printing XML documents from and to our internal Haskell representation, we have tested the generated put functions of the lenses with pre-compiled input databases of increasing size (measured in MBytes needed to store their Haskell definitions), randomly generated with the *QuickCheck* testing suite (Claessen and Hughes, 2000). We compiled each function using `GHC 7.2.2` with optimization flag `O2`. As expected, the original specification performs much worse than the optimized lens, and the loss factor grows with the database size. Considering the biggest sample, the loss factors are of 3.7 in time and 4.1 in space for the IMDb example and of 9.4 in time and 13.4 in space for the paradise example. The more significant results (and the worse overall performance) for the paradise example are justified by the elimination of the recursive traversals over sub-departments.

To better quantify the degree of our optimizations, we should also include in our benchmark a comparison with an handwritten definition. However, it is extremely complex to hand-code the put functions of the lenses used in our examples, let alone efficient versions. For that purpose, we consider a simple example of a lens that counts the number of women in a list of people containing names and gender, defined in our point-free lens language as follows:

$$\mathbf{type}\ \mathit{Person} = (\mathit{Name}, \mathit{Gender}) \qquad \mathbf{type}\ \mathit{Name} = \mathit{String}$$

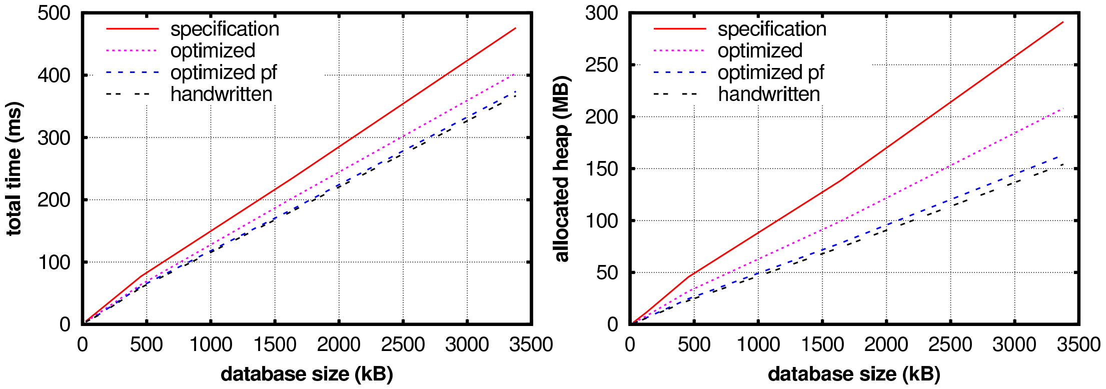


Figure 6.10: Benchmark results for the *women* example.

$$\mathbf{data}\ \mathit{Gender} = \mathit{Male} \mid \mathit{Female}$$

$$\mathit{women} : [\mathit{Person}] \trianglerighteq \mathit{Nat}$$

$$\mathit{women} = \mathit{length} \circ \mathit{filter\_left} \circ \mathit{map}\ (\mathit{out}_{\mathsf{Gender}} \circ \pi_2{}^{\underline{\texttt{"Eve"}}\circ !})$$

The benchmark results for this example are presented in Figure 6.10. Below the optimized lens, we introduce another measure for the output of a second optimization phase performed on the point-free definition of the *put* function (`optimized pf`). Even for this simple example, the optimized *put* allocates nearly half the memory and performs very close to an handwritten definition (`handwritten`), both in time and space. This additional speedup reported in `optimized pf` is mainly due to the optimization of expressions involving “opaque” lens isomorphisms, such as *assocl* or *distl*.

6.5 Summary

This chapter proposed *Multifocal*, a strategic two-level bidirectional transformation language for XML Schema evolution with document-level migrations based on bidirectional lenses. By using strategic programming techniques, these coupled transformations can be specified in a concise and generic way, mimicking the typical coding pattern of XML transformation languages such as XSLT, that allow to easily specify how to modify only selected nodes via specific templates. When applied to input schemas, our schema-level transformations produce new schemas, as well as bidirectional lens transformations that propagate updates between old and new documents. In the *Multifocal* framework, we release such bidirectional transformations as independent programs that can be used to translate updates for particular source and target schemas. We also provide users with an optional optimization phase that improves the efficiency of the generated lens programs for intensive usage scenarios.

At the core of this framework, we have developed a library of point-free lenses between inductive data types (supporting both state-based and delta-based lenses), a library of two-level strategic lenses for the evolution of arbitrary inductive data type representations, and a library for the automated rewriting of point-free lenses and for the specialization of generic queries over recursive types.

Our framework has been fully implemented in Haskell, and is available through the Hackage package repository (`http://hackage.haskell.org`) under the name `multifocal`. The three core Haskell libraries are available under the names `pointless-lenses`, `pointless-2lt` and `pointless-rewrite`, honoring a common joke about the point-free style.

Chapter 7

Conclusion

This dissertation discusses the design and implementation of the *Multifocal* framework for defining generic views of XML Schemas and algebraic data type representations. In response to the research challenges posed in Chapter 1, our framework is: (1) two-level, in the sense that each transformation, for a given source schema, provides a view schema and data conversion functions between documents that conform to the schemas; (2) bidirectional, by supporting both forward and backward document-level functions, obeying a strong bidirectional semantics that is based on total well-behaved lenses and formalizes the schema-level abstractions; (3) and suited to the automatic optimization of the lens transformations resulting from the evaluation of the two-level stage, underpinned by a rich algebraic calculus of point-free lenses. Within the development of the *Multifocal* framework, our main contributions are:

- A point-free lens language (Chapter 4). We showed that most of the standard point-free combinators can be lifted to well-behaved lenses. To express generic lenses over arbitrary inductive data types, we identified precise termination conditions required to lift recursion patterns to well-behaved lenses. We believe that, using suitable techniques, these conditions are easier to verify than the ones stated in (Foster et al., 2007) concerning general recursion. In Chapter 6, we implemented this lens language as a Haskell library to aid the construction of functional bidirectional programs by composition.

- An algebraic theory of point-free lenses (Chapter 4). In particular, we studied that most algebraic laws characterizing the point-free combinators can also be lifted to lenses. This theory allows to reason directly about lenses defined in our point-free language and permits to write conventional proofs at the lens level, using only their forward point-free specification. To prove the usefulness of this calculus, we employed it at the kernel of an automatic optimization library for point-free lenses, thus mitigating the inefficiency of bidirectional transformations while preserving the simplicity and elegance of a combinatorial approach. The implementation of this library (Chapter 6) extends a previous point-free rewrite system (Cunha and Visser, 2011) to support lenses. A key result of our theory was that uniqueness (and therefore fusion) laws also holds for lens recursion patterns. We proposed a technique to mechanize the challenging fusion laws.

- A point-free language of alignment-aware delta lenses (Chapter 5). This language was constructed by recasting the point-free combinators as delta lenses with an explicit notion of shape and data, whose transformations process not only states but also deltas representing the issued updates. We made use of such deltas to refine the positional behavior of our original lenses, namely by identifying mapping lenses that only modify data and by instrumenting the recursion pattern combinators with alignment mechanisms that infer and propagate edit operations on shapes. The resulting lenses perform better than the previous ones due to the delta information, in the sense that they produce smaller source updates for certain view updates. Moreover, they are able to solve alignment for intricate reshaping scenarios not considered before. An implementation of this delta lenses language, using a simple minimal edit sequence differencing algorithm (Tichy, 1984), is distributed as part of our Haskell lens programming library (Chapter 6).

- A Haskell library of strategic two-level lenses (Chapter 6). The combinators provided by this language can be used to encode generic views over arbitrary inductive data types, allowing users to specify only the particular evolution steps that modify the structure of the original type. As generic programming idioms, our library supports both: generic value-level queries that collect values of a specific type; and strategic type traversals that apply type-level transformations at arbitrary levels inside a type. The value-level semantics of the two-level combinators was defined according to our point-free lens language.

- The *Multifocal* XML transformation language (Chapter 6). This language can be used to write structure-shy two-level transformations between XML Schemas in a concise and compositional way, by combining flexible generic strategies with specific XML transformers. Coupled XML transformations written in *Multifocal* are translated into our Haskell library, by converting the source and target XML Schemas to equivalent algebraic data type representations. When applied to an input XML Schema, a *Multifocal* program produces a new XML Schema, as well as a bidirectional lens compiled to a Haskell executable file that propagates updates between XML documents conforming to old and new schemas. We demonstrated the practical application of the *Multifocal* language by using it to describe the evolution of two particular XML Schemas. In particular, we showed that an automatic optimization phase greatly improves the efficiency of the generated lens transformations.

Another contribution of this thesis is a detailed picture of the state of the art of the field of bidirectional transformations, with particular emphasis on lenses. In Chapter 3, we presented a taxonomy for the classification of the most relevant features found in bidirectional transformation frameworks, and surveyed the most influential papers in the field by instantiating them in our taxonomy. Hopefully, this revision can help situating the work developed in this thesis among the vast number of existing bidirectional approaches, and be used as a reference for readers who are not experts in bidirectional transformations and wish to acquire a broad knowledge on the field.

7.1 Final Remarks

Our research on an algebra of point-free lenses was carried out in the SET category, where all functions are total and data types for finite and infinite data structures constitute two separate worlds. However, lazy functional languages like Haskell are not so well-defined and permit partial and arbitrary recursive functions, which reveals some discrepancies between our theoretical semantic domain and our practical Haskell implementation. In order to define (terminating) hylomorphisms and be able to build recursive lenses, we have restricted ourselves the notion of a recursive anamorphism that builds a value of an inductive types and is always a total and terminating function. In practice this implies that, apart from some particularly identified cases, developers must prove precise termination conditions to guarantee that user-defined recursive lenses are well-behaved. Note that, nevertheless, all lenses underlying *Multifocal* transformations fall into such particular class and therefore are guaranteed to be terminating, and consequently well-behaved. An interesting solution would be to link our library with existing static termination checkers like (Sereni, 2007). Although an elegant relational definition of well-foundedness is well-known (Bird and de Moor, 1997), and (Backhouse and Doornbos, 2001; Capretta et al., 2006) report studies towards a calculus of well-founded relations, the proofs of termination for particular coalgebras except trivial ones using these theories are still highly manual and require a significant amount of creative input from the programmer. A more challenging but engaging path (at least for a not-disinterested point-free advocate) would be to investigate agile mechanizable techniques to support a static termination checker based on the relational calculus, likewise our point-free automatic optimization tool. In the semantic domain of most lazy functional languages, partial functions and non-terminating functions are modeled by introducing a least bottom element $\perp_A$, for each type A. The calculational counterpart of such additional structure is that many algebraic laws have to either be relaxed to work only modulo bottoms or consider extra strictness side conditions ensuring the preservation of bottoms.

The bidirectional scheme assumed in this thesis is that of total well-behaved lenses. This formulation provides strong semantic properties and guarantees that all source and target updates can be successfully propagated, but our choice of a standard (and decidable) type system, implemented in functional languages like Haskell, has restricted the class of lens programs supported by *Multifocal* to that of perfect data abstractions. That said, some data transformations like data insertion and value-level filtering are not supported, unlike in other bidirectional approaches. Nevertheless, most of our results and techniques are not bound to total well-behaved lenses and could be generalized to other bidirectional formulations supporting a wider class of transformations. This would require, however, some tradeoff with the remaining defining features of the bidirectional framework.

Regarding our point-free delta lens language, now and then we have disclosed the algebraic laws that rule some of our combinators. In general, it is reasonable to assume that the algebraic laws from Chapter 4 are preserved in the delta lens world, with the natural exception being the combinators that have a more refined delta-based backward semantics. For example, in Chapter 5 we refrained from expressing fusion laws for delta lens recursion patterns, because these would involve side conditions stating that the fused lens is compatible with the shape alignment procedures. Explicit shape programming also introduced the $\overset{\leftrightarrow}{\cdot}$-NAT naturality law entailing that sequences of mapping and restructuring lenses can be shifted to separate groups and fused together into a single mapping-restructuring pair. Unfortunately, when embedded as state-based lenses and combined with regular point-free lenses, our point-free delta lenses do not enjoy interesting algebraic laws. This is because the semantics of the resulting lenses is directly dependent on the differencing procedures used to infer the view-update deltas, and the interaction of these differencing functions with the lenses must be taken into account by the algebraic laws. An example of this side effect is the MAP-DIFF-COMP law found in Section 5.3.2. It remains as an open interesting topic to investigate formulations of hybrid state-based and delta lenses with nice combined algebraic properties.

All the tools and libraries developed in this thesis have been implemented in Haskell, and make heavy use of advanced functional programming features such as GADTs and type-indexed type families to support the embedding of our domain-specific transformation languages. Although being able to inherit the type system and high-level features of Haskell is advantageous for developing our prototype framework, their extensive use may hinder the overall efficiency of the resulting framework. Another disadvantage is that some limitations of the Haskell language make it difficult to encode certain theoretically simple features such as n-ary polymorphic data types and true structural type equivalence. A dedicated implementation, with its own type system, would require significantly more effort but could provide a native support for such features.

7.2 Future Work

The work developed in this dissertation suggests several open directions that can be explored in future work. We enunciate some of them in this section.

***Multifocal* Framework.** Although *Multifocal* already supports interesting XML transformation scenarios, the expressiveness of the underlying bidirectional transformations is naturally limited by the language of point-free lenses in use. In future work, we plan to extend our language to support more XPath features and to leverage the underlying bidirectional scheme to other bidirectional formulations supporting different kinds of transformations that do not impose an abstraction data flow.

Currently, the *Multifocal* language does not support alignment-aware lenses. Naturally, a valid solution would be to include a backdoor for applying “black box” user-defined lenses at a particular location, as in the underlying Haskell library. A more interesting solution would be to provide language-based techniques in the style of (Barbosa et al., 2010) to annotate generic two-level programs with alignment information without compromising their structure-shyness. In this context, we are investigating the "deltification" of ordinary point-free lenses through the processing of annotations in the input XML Schemas such as `xs:key` to identify reorderable chunks in the source document and guide the translation of view updates.

Multifocal proposes a way of replacing three unidirectional XML transformations (a schema-level transformation and two transformations between XML documents) with a single two-level bidirectional transformation. In order to bridge the gap to standard XML transformation tools, translations from XSLT-like idioms to *Multifocal* should be developed. A successful integration would also encompass a comparative study on the usefulness, expressiveness and efficiency of *Multifocal* transformations.

Invariants. In our language of point-free lenses, the non-existence of splits and injections triggered the definition of several plumbing isomorphism combinators to regain some expressiveness. In order to alleviate this problem, we are considering to extend our lens framework with invariants that capture the exact domain and range of the transformations, so that any transformation written in a point-free functional language can be lifted to a total well-behaved lens between particular domains. A first effort towards this is reported in (Macedo et al., 2012), by considering a language of point-free lenses with invariants whose backward transformations generate all possible source updates satisfying the lens laws and the source/target invariants. However, a suitable mechanism to choose from such various solutions (and more refined lens laws as discussed below) is still imperative for a more practical application. We also intend to investigate other approaches to derive backward transformations that are correct by construction and satisfy the respective invariants.

Bidirectional Properties. In general, all bidirectional approaches try to convey a combination of properties and semantics that ensure some degree of reasonability on the behavior of bidirectional transformations. Ideally, this reasonable behavior should be postulated by a *Principle of Least Change* (Meertens, 1998), entailing that updates are as small as possible, but unfortunately it is hard to formalize and to find such minimal updates in practice.

Therefore, the traditional bidirectional properties only establish "first principles" and still leave a lot of room for unpredictable and sometimes unreasonable behavior.

Even for operation-based frameworks, the more refined bidirectional properties simply entail round-tripping modulo updates and also do not impose any quality measure on update translation. For instance, our delta-based lens laws guarantee that deltas are correctly propagated alongside transformations, but not that all (possible) alignment information is preserved. In the future, this should be solved with the proposal of new generic laws that better characterize reasonable and minimal behavior, for particular application domains.

Equally important is to discover flexible mechanisms to derive suitable backward transformations satisfying a more refined set of reasonable update translation properties. Even according to such criteria, update translation is inherently ambiguous. To disambiguate, Keller (1986) runs a dialog with the developer of the forward transformation to collect sufficient information to make the backward transformation deterministic at forward definition time. An interesting research topic would be to study related language-based techniques to allow users to control such non-deterministic choices at the time of defining the bidirectional transformations.

Design Space. Not less important than other topics is the still pressing need in the community for a complete, precise and self-contained survey on the design space of bidirectional transformations. Although our taxonomy and state of the art from Chapter 3 contribute positively towards such a goal, there is still a lot of room to explore in such design space.

On the one side, our taxonomy and classification point to some "holes" in the design space corresponding to hitherto combinations of features. An interesting question is to discover which of those are impossible or meaningless and which are promising opportunities that have simply not been instantiated yet. Additionally, our taxonomy could be extended with more axes and features characterizing other bidirectional design features, for example, to assess the support for particular verification or implementation capabilities of the different approaches.

As discussed above, the usual bidirectional properties guaranteed by most frameworks do not seriously constrain the behavior of the permitted transformations, making it harder to compare the effectiveness of two bidirectional frameworks based solely on their supported properties. In the continuation of (Czarnecki et al., 2009), this problem could be alleviated by developing a suite of paradigmatic "benchmark scenarios" for each application domain, so that users can compare the behavior of different frameworks within that domain.

Finally, it is essential for a classification effort to be validated and consolidated according to input from the envisaged community. We believe that the accuracy and completeness of our survey could be greatly improved by enrolling the bidirectional transformation community. Such debate could be proposed to take place in future scientific forums in the followup of previous events such as the *GRACE International Meeting on Bidirectional Transformations*, the *Dagstuhl Seminar on Bidirectional Transformations* or the *First International Workshop on Bidirectional Transformations*.

Appendix A

Additional Point-free Laws and Proofs

A.1 Functional Point-free Laws and Proofs

Basic Laws

$$f \circ h = g \circ h \;\Leftarrow\; f = g \qquad \textsc{Leibniz}$$

$$(f \times g) \circ (h \vartriangle i) = f \circ h \vartriangle g \circ i \qquad \times\text{-}\textsc{Absor}$$

$$(f \vartriangle g) = (h \vartriangle i) \;\Leftrightarrow\; f = h \;\wedge\; g = i \qquad \times\text{-}\textsc{Equal}$$

$$(f \triangledown g) \circ (h + i) = f \circ h \triangledown g \circ i \qquad +\text{-}\textsc{Absor}$$

$$(f \triangledown g) = (h \triangledown i) \;\Leftrightarrow\; f = h \;\wedge\; g = i \qquad +\text{-}\textsc{Equal}$$

Isomorphism Laws

$$swap = \pi_2 \vartriangle \pi_1 \qquad swap\text{-}\textsc{Def}$$

$$(f \times g) \circ swap = swap \circ (g \times f) \qquad swap\text{-}\textsc{Nat}$$

$$swap \circ swap = id \qquad swap\text{-}\textsc{Iso}$$

$$assocl = (id \times \pi_1) \vartriangle (\pi_2 \circ \pi_2) \qquad assocl\text{-}\textsc{Def}$$

$$((f \times g) \times h) \circ assocl = assocl \circ (f \times (g \times h)) \qquad assocl\text{-}\textsc{Nat}$$

$$assocr = (\pi_1 \circ \pi_1) \vartriangle (\pi_2 \times id) \qquad assocr\text{-}\textsc{Def}$$

$$(f \times (g \times h)) \circ assocr = assocr \circ ((f \times g) \times h) \qquad assocr\text{-}\textsc{Nat}$$

$$assocl \circ assocr = id \quad \wedge \quad assocr \circ assocl = id \qquad assocl\text{-}assocr\text{-}\textsc{Iso}$$

$$coswap = i_2 \triangledown i_1 \qquad coswap\text{-}\textsc{Def}$$

$$(f+g)\circ \mathit{coswap} = \mathit{coswap}\circ(g+f) \qquad \mathit{coswap}\text{-}\textsc{Nat}$$

$$\mathit{coswap}\circ \mathit{coswap} = \mathit{id} \qquad \mathit{coswap}\text{-}\textsc{Iso}$$

$$\mathit{coassocl} = (i_1\circ i_1)\,\triangledown(i_2+\mathit{id}) \qquad \mathit{coassocl}\text{-}\textsc{Def}$$

$$\mathit{coassocl}\circ(f+(g+h)) = ((f+g)+h)\circ \mathit{coassocl} \qquad \mathit{coassocl}\text{-}\textsc{Nat}$$

$$\mathit{coassocr} = (\mathit{id}+i_1)\,\triangledown(i_2\circ i_2) \qquad \mathit{coassocr}\text{-}\textsc{Def}$$

$$\mathit{coassocr}\circ((f+g)+h) = (f+(g+h))\circ \mathit{coassocr} \qquad \mathit{coassocr}\text{-}\textsc{Nat}$$

$$\mathit{coassocl}\circ \mathit{coassocr} = \mathit{id} \quad\wedge\quad \mathit{coassocr}\circ \mathit{coassocl} = \mathit{id} \qquad \mathit{coassocl}\text{-}\mathit{coassocr}\text{-}\textsc{Iso}$$

$$\mathit{distl} = \mathit{ap}\circ((\overline{i_1}\,\triangledown\,\overline{i_2})\times \mathit{id}) \qquad \mathit{distl}\text{-}\textsc{Def}$$

$$\mathit{distl}\circ((f+g)\times h) = (f\times h+g\times h)\circ \mathit{distl} \qquad \mathit{distl}\text{-}\textsc{Nat}$$

$$\mathit{distl}\circ(i_1\times \mathit{id}) = i_1 \quad\wedge\quad \mathit{distl}\circ(i_2\times \mathit{id}) = i_2 \qquad \mathit{distl}\text{-}\textsc{Cancel}$$

$$(\mathit{id}\,\triangledown\,\mathit{id})\circ \mathit{distl} = (\mathit{id}\,\triangledown\,\mathit{id})\times \mathit{id} \qquad \mathit{distl}\text{-}\mathit{id}\text{-}\textsc{Cancel}$$

$$(\pi_1+\pi_1)\circ \mathit{distl} = \pi_1 \qquad \mathit{distl}\text{-}\pi_1\text{-}\textsc{Cancel}$$

$$(\pi_2\,\triangledown\,\pi_2)\circ \mathit{distl} = \pi_2 \qquad \mathit{distl}\text{-}\pi_2\text{-}\textsc{Cancel}$$

$$\mathit{undistl} = (i_1\times \mathit{id})\,\triangledown(i_2\times \mathit{id}) \qquad \mathit{undistl}\text{-}\textsc{Def}$$

$$\mathit{distl}\circ \mathit{undistl} = \mathit{id} \quad\wedge\; \mathit{undistl}\circ \mathit{distl} = \mathit{id} \qquad \mathit{distl}\text{-}\mathit{undistl}\text{-}\textsc{Iso}$$

$$\mathit{distr} = (\mathit{swap}+\mathit{swap})\circ \mathit{distl}\circ \mathit{swap} \qquad \mathit{distr}\text{-}\textsc{Def}$$

$$\mathit{distr}\circ(f\times(g+h)) = (f\times g+f\times h)\circ \mathit{distr} \qquad \mathit{distr}\text{-}\textsc{Nat}$$

$$\mathit{distr}\circ(\mathit{id}\times i_1) = i_1 \quad\wedge\quad \mathit{distr}\circ(\mathit{id}\times i_2) = i_2 \qquad \mathit{distr}\text{-}\textsc{Cancel}$$

$$(\mathit{id}\,\triangledown\,\mathit{id})\circ \mathit{distr} = \mathit{id}\times(\mathit{id}\,\triangledown\,\mathit{id}) \qquad \mathit{distr}\text{-}\mathit{id}\text{-}\textsc{Cancel}$$

$$(\pi_1\,\triangledown\,\pi_1)\circ \mathit{distr} = \pi_1 \qquad \mathit{distr}\text{-}\pi_1\text{-}\textsc{Cancel}$$

$$(\pi_2+\pi_2)\circ \mathit{distr} = \pi_2 \qquad \mathit{distr}\text{-}\pi_2\text{-}\textsc{Cancel}$$

$$\mathit{undistr} = (\mathit{id}\times i_1)\,\triangledown(\mathit{id}\times i_2) \qquad \mathit{undistr}\text{-}\textsc{Def}$$

$$\mathit{distr}\circ \mathit{undistr} = \mathit{id} \quad\wedge\quad \mathit{undistr}\circ \mathit{distr} = \mathit{id} \qquad \mathit{distr}\text{-}\mathit{undistr}\text{-}\textsc{Iso}$$

Shapely Type Laws

$$\mathit{shape}\circ \mathit{recover} = \pi_1 \quad\wedge\quad \mathit{data}\circ \mathit{recover} = \pi_2 \qquad \mathit{recover}\text{-}\textsc{Cancel}$$

$$\mathsf{F}\,f\circ \mathit{recover} = \mathit{recover}\circ(\mathit{id}\times f^{\bullet}) \qquad \mathit{recover}\text{-}\textsc{Data}$$

$$\eta\circ \mathit{recover} = \mathit{recover}\circ(\eta\circ\pi_1\vartriangle\hat{\circ}\circ(\pi_2\vartriangle\overleftarrow{n}\circ\pi_1)) \qquad \mathit{recover}\text{-}\textsc{Shape}$$

$$\mathit{shape}\circ\mathsf{F}\,f = \mathit{shape} \qquad \mathit{shape}\text{-}\textsc{Data}$$

$$shape \circ \eta = \eta \circ shape \qquad \textit{shape}\text{-}\textsc{Shape}$$

$$data \circ \mathsf{F}\, f = f^{\bullet} \circ data \qquad \textit{data}\text{-}\textsc{Data}$$

$$data \circ \eta = \hat{\circ} \circ (data \bigtriangleup \overleftarrow{\eta}) \qquad \textit{data}\text{-}\textsc{Shape}$$

$$\overleftarrow{\eta} \circ shape = \overleftarrow{\eta} \qquad \overleftarrow{\cdot}\text{-}\textsc{Shape}$$

$$\overleftarrow{\pi_1} = \underline{i_1} \circ ! \qquad \overleftarrow{\cdot}\text{-}\textsc{Fst}$$

$$\overleftarrow{id} = \underline{id} \circ ! \qquad \overleftarrow{\cdot}\text{-}\textsc{Id}$$

$$\overleftarrow{f \circ g} = \hat{\circ} \circ (\overleftarrow{g} \bigtriangleup \overleftarrow{f} \circ g) \qquad \overleftarrow{\cdot}\text{-}\textsc{Comp}$$

$$\overleftarrow{f \bigtriangleup g} = \hat{\nabla} \circ (\overleftarrow{f} \bigtriangleup \overleftarrow{g}) \qquad \overleftarrow{\cdot}\text{-}\textsc{Split}$$

Monoid Laws

$$plus \circ (\underline{zero} \circ ! \bigtriangleup f) = f \quad \wedge \quad plus \circ (f \bigtriangleup \underline{zero} \circ !) = f \qquad \textit{plus}\text{-}\textsc{Zero}$$

$$\underline{zero} \circ ! \;\nabla\; \underline{zero} \circ ! = \underline{zero} \circ ! \qquad +\text{-}\textsc{Zero}$$

$$(\!|f|\!)_{\mathsf{F}} = \underline{zero} \circ ! \quad \Leftarrow \quad f \circ \mathsf{F}\; \underline{zero} \circ ! = \underline{zero} \circ ! \qquad (\!|\cdot|\!)\text{-}\textsc{Zero}$$

Functor Laws

$$\mathsf{B}\; \pi_1\; \pi_1 \circ bzip_{\mathsf{B}}\; f\; g = \pi_1 \qquad \textit{bzip}\text{-}\textsc{Cancel}$$

$$bzip_{\mathsf{B}}\; f\; g \circ (\mathsf{B}\; h\; i \bigtriangleup \mathsf{B}\; j\; k) = \mathsf{B}\; (h \bigtriangleup j)\; (i \bigtriangleup k) \qquad \textit{bzip}\text{-}\textsc{Split}$$

$$bzip_{\mathsf{B}}\; f\; g \circ (\mathsf{B}\; h\; i \times id) = \mathsf{B}\; (h \times id)\; (i \times id) \circ bzip_{\mathsf{B}}\; (f \circ h)\; (g \circ i) \qquad \textit{bzip}\text{-}\textsc{Nat}$$

$$\left.\begin{array}{l}
\sigma_{\mathsf{F}} : \mathsf{F}\; A \times B \to \mathsf{F}\; (A \times B) \\
\sigma_{\mathsf{Id}} = id \\
\sigma_{\underline{A}} = \pi_1 \\
\sigma_{\mathsf{F} \otimes \mathsf{G}} = (\sigma_{\mathsf{F}} \circ (\pi_1 \times id) \bigtriangleup \sigma_{\mathsf{G}} \circ (\pi_2 \times id)) \\
\sigma_{\mathsf{F} \oplus \mathsf{G}} = (\sigma_{\mathsf{F}} + \sigma_{\mathsf{G}}) \circ distl \\
\sigma_{\mathsf{F} \odot \mathsf{G}} = \mathsf{F}\; \sigma_{\mathsf{G}} \circ \sigma_{\mathsf{F}} \\
\sigma_T = [\![\sigma_{\mathsf{B}} \circ (out_{\mathsf{B}\; A} \times id)]\!]_{\mathsf{B}\; (A \times B)}
\end{array}\right\} \qquad \sigma\text{-}\textsc{Def}$$

$$\left.\begin{aligned}
&\sigma_{\mathsf{B}} : \mathsf{B}\ A\ B \times C \to \mathsf{B}\ (A \times C)\ (B \times C)\\
&\sigma_{\mathsf{Id}} = id\\
&\sigma_{\mathsf{Par}} = id\\
&\sigma_{\underline{A}} = \pi_1\\
&\sigma_{\mathsf{F} \otimes \mathsf{G}} = (\sigma_{\mathsf{F}} \circ (\pi_1 \times id) \vartriangle \sigma_{\mathsf{G}} \circ (\pi_2 \times id))\\
&\sigma_{\mathsf{F} \oplus \mathsf{G}} = (\sigma_{\mathsf{F}} + \sigma_{\mathsf{G}}) \circ distl\\
&\sigma_{\mathsf{F} \odot \mathsf{B}} = \mathsf{F}\ \sigma_{\mathsf{B}} \circ \sigma_{\mathsf{F}}
\end{aligned}\right\} \qquad \sigma\text{-}\textsc{Def}$$

*Proof (fzip-*CANCEL*).*

$$\begin{aligned}
&\mathsf{Id}\ \pi_1 \circ fzip_{\mathsf{Id}}\ f\\
&\ = \{fzip\text{-}\textsc{Def};\ id - \textsc{Nat}\}\\
&\pi_1
\end{aligned}$$

$$\begin{aligned}
&\underline{C}\ \pi_1 \circ fzip_{\underline{C}}\ f\\
&\ = \{fzip\text{-}\textsc{Def};\ id - \textsc{Nat}\}\\
&\pi_1
\end{aligned}$$

$$\begin{aligned}
&(\mathsf{F} \otimes \mathsf{G})\ \pi_1 \circ fzip_{\mathsf{F} \otimes \mathsf{G}}\ f\\
&\ = \{fzip\text{-}\textsc{Def}\}\\
&(\mathsf{F}\ \pi_1 \times \mathsf{G}\ \pi_1) \circ (fzip_{\mathsf{F}}\ f \times fzip_{\mathsf{G}}\ f) \circ distp\\
&\ = \{\times - \textsc{Functor-Comp}; fzip - \textsc{Cancel}\}\\
&(\pi_1 \times \pi_1) \circ distp\\
&\ = \{\ distp - \textsc{Def}; \times\text{-}\textsc{Absor}; \times - \textsc{Cancel}\}\\
&\pi_1 \circ \pi_1 \vartriangle \pi_2 \circ \pi_1\\
&\ = \{\times - \textsc{Fusion}; \times - \textsc{Reflex};\ id - \textsc{Nat}\}\\
&\pi_1
\end{aligned}$$

$$\begin{aligned}
&(\mathsf{F} \oplus \mathsf{G})\ \pi_1 \circ fzip_{\mathsf{F} \oplus \mathsf{G}}\ f\\
&\ = \{fzip\text{-}\textsc{Def}\}\\
&(\mathsf{F}\ \pi_1 + \mathsf{G}\ \pi_1) \circ (fzip_{\mathsf{F}}\ f \triangledown \mathsf{F}\ (id \vartriangle f) \circ \pi_1 + \mathsf{G}\ (id \vartriangle f) \circ \pi_1 \triangledown fzip_{\mathsf{G}}\ f) \circ dists\\
&\ = \{+ - \textsc{Functor-Comp}; +\text{-}\textsc{Absor}\}\\
&(\mathsf{F}\ \pi_1 \circ fzipF\ f \triangledown \mathsf{F}\ \pi_1 \circ \mathsf{F}\ (id \vartriangle f) \circ \pi_1 + \mathsf{G}\ \pi_1 \circ \mathsf{G}\ (id \vartriangle f) \circ \pi_1 \triangledown \mathsf{G}\ \pi_1 \circ fzip_{\mathsf{G}})
\end{aligned}$$

$\circ\ \mathit{dists}$

$= \{\mathit{fzip} - \textsc{Cancel}; \textsc{Functor-Comp}\}$

$(\pi_1 \triangledown \mathsf{F}\ (\pi_1 \circ (\mathit{id} \vartriangle f)) \circ \pi_1 + \mathsf{G}\ (\pi_1 \circ (\mathit{id} \vartriangle f)) \circ \pi_1 \triangledown \pi_1) \circ \mathit{dists}$

$= \{\textsc{Functor-Id}\}$

$((\pi_1 \triangledown \pi_1) + (\pi_1 \triangledown \pi_1)) \circ \mathit{dists}$

$= \{\mathit{dists} - \textsc{Def}; \mathit{distr} - \pi_1\text{-}\textsc{Cancel}\}$

$(\pi_1 + \pi_1) \circ \mathit{distl}$

$= \{\mathit{distl} - \pi_1\text{-}\textsc{Cancel}\}$

π_1

$(\mathsf{F} \odot \mathsf{G})\ \pi_1 \circ \mathit{fzip}_{\mathsf{F} \odot \mathsf{G}}\ f$

$= \{\textsc{Functor-Def}; \mathit{fzip}\text{-}\textsc{Def}\}$

$\mathsf{F}\ (\mathsf{G}\ \pi_1) \circ \mathsf{F}\ (\mathit{fzip}_{\mathsf{G}}\ f) \circ \mathit{fzip}_{\mathsf{F}}\ (\mathsf{G}\ f)$

$= \{\textsc{Functor-Comp}; \mathit{fzip} - \textsc{Cancel}\}$

$\mathsf{F}\ \pi_1 \circ \mathit{fzip}_{\mathsf{F}}\ (\mathsf{G}\ f)$

$= \{\mathit{fzip} - \textsc{Cancel}\}$

π_1

$T\ \pi_1 \circ \llbracket \mathit{bzip}_{\mathsf{B}}\ f\ (T\ f) \circ (\mathit{out}_{\mathsf{B}\ A} \times \mathit{out}_{\mathsf{B}\ C}) \rrbracket_{\mathsf{B}\ (A \times C)} = \pi_1$

$\Leftrightarrow \{\textsc{Map} - \textsc{Def}; \llbracket \cdot, \cdot \rrbracket - \textsc{Uniq}\}$

$\mathit{in}_{\mathsf{B}\ A} \circ \mathsf{B}\ \pi_1\ \mathit{id} \circ \mathsf{B}\ \mathit{id}\ \pi_1 \circ \mathit{bzip}_{\mathsf{B}}\ f\ (T\ f) \circ (\mathit{out}_{\mathsf{B}\ A} \times \mathit{out}_{\mathsf{B}\ C}) = \pi_1$

$\Leftrightarrow \{\textsc{Bifunctor-Comp}; \mathit{bzip} - \textsc{Cancel}\}$

$\mathit{in}_{\mathsf{B}\ A} \circ \pi_1 \circ (\mathit{out}_{\mathsf{B}\ A} \times \mathit{out}_{\mathsf{B}\ C}) = \pi_1$

$\Leftrightarrow \{\times - \textsc{Def}; \times - \textsc{Cancel}\}$

$\mathit{in}_{\mathsf{B}\ A} \circ \mathit{out}_{\mathsf{B}\ A} \circ \pi_1 = \pi_1$

$\Leftrightarrow \{\mathit{in} - \mathit{out} - \textsc{Iso}\}$

$\textsc{True}$

□

Recursive Laws

$\llbracket g, h \rrbracket_{\mathsf{F}} = g \circ \mathsf{F}\ \llbracket g, h \rrbracket_{\mathsf{F}} \circ h$ $\qquad \llbracket \cdot, \cdot \rrbracket$-CANCEL

$g \circ \llbracket h, i \rrbracket_{\mathsf{F}} \circ j = \llbracket k, l \rrbracket_{\mathsf{F}} \ \Leftarrow\ g \circ h = k \circ \mathsf{F}\ g \ \wedge\ i \circ j = \mathsf{F}\ j \circ l$ $\qquad \llbracket \cdot, \cdot \rrbracket$-FUSION

Lens Laws

$$put \circ (id \vartriangle create) = create \qquad \textsc{CreatePut}$$

$$put \circ (\pi_1 \vartriangle put) = put \qquad \textsc{PutTwice}$$

Proof (CREATEPUT).

$$
\begin{aligned}
& put \circ (id \vartriangle create) \\
& = \{\textsc{CreateGet}\} \\
& put \circ (get \circ create \vartriangle create) \\
& = \{id - \textsc{Nat}; \times - \textsc{Fusion}\} \\
& put \circ (get \vartriangle id) \circ create \\
& = \{\textsc{GetPut}; id - \textsc{Nat}\} \\
& create
\end{aligned}
$$

□

Proof (PUTTWICE).

$$
\begin{aligned}
& put \circ (\pi_1 \vartriangle put) \\
& = \{\textsc{PutGet}\} \\
& put \circ (get \circ put \vartriangle put) \\
& = \{id - \textsc{Nat}; \times - \textsc{Fusion}\} \\
& put \circ (get \vartriangle id) \circ put \\
& = \{\textsc{GetPut}; id - \textsc{Nat}\} \\
& put
\end{aligned}
$$

□

A.2 Relational Point-free Laws and Proofs

$$id \circ R = R = R \circ id \qquad id\text{-}\textsc{Nat}$$

$$R \circ (S \circ T) = (R \circ S) \circ T \qquad \circ\text{-}\textsc{Assoc}$$

$\pi_1 \circ (R \vartriangle S) = R \circ \delta S \quad \wedge \quad \pi_2 \circ (R \vartriangle S) = S \circ \delta R$	×-Cancel
$(R \vartriangle S) \circ T = R \circ T \vartriangle S \circ T \quad \Leftarrow \quad R \circ T \circ T^{\circ} \subseteq R \vee S \circ T \circ T^{\circ} \subseteq S$	×-Fusion
$R \times S = R \circ \pi_1 \vartriangle S \circ \pi_2$	×-Def
$(R \times S) \circ (T \vartriangle U) = R \circ T \vartriangle S \circ U$	×-Absor
$(R \times S) \circ (T \times U) = R \circ T \times S \circ U$	×-Functor-Comp
$(R \triangledown S) \circ i_1 = R \quad \wedge \quad (R \triangledown S) \circ i_2 = S$	+-Cancel
$U \circ (R \triangledown S) = U \circ R \triangledown U \circ S$	+-Fusion
$R + S = i_1 \circ R \triangledown i_2 \circ S$	+-Def
$(R \triangledown S) \circ (T + U) = R \circ T \triangledown S \circ U$	+-Absor
$(R + S) \circ (T + U) = R \circ T + S \circ U$	+-Functor-Comp
$R^{\circ\circ} = R$	$\cdot^{\circ}$-Inv
$(R \circ S)^{\circ} = S^{\circ} \circ R^{\circ}$	$\cdot^{\circ}$-Comp
$(R \cup S)^{\circ} = R^{\circ} \cup S^{\circ} \quad \wedge \quad (R \cap S)^{\circ} = R^{\circ} \cap S^{\circ}$	$\cdot^{\circ}$-Dist
$(R \times S)^{\circ} = R^{\circ} \times S^{\circ}$	$\cdot^{\circ}$-Prod
$(R + S)^{\circ} = R^{\circ} + S^{\circ}$	$\cdot^{\circ}$-Sum
$\bot \cup R = R = R \cup \bot$	$\bot$-Neutral
$\bot \cap R = \bot = R \cap \bot$	$\bot$-Absor
$\bot \circ R = \bot = R \circ \bot$	$\bot$-Fusion
$R \circ (S \cup T) = R \circ S \cup R \circ T \quad \wedge \quad (S \cup T) \circ R = S \circ R \cup T \circ R$	$\cup$-Fusion
$R \circ (S \cap T) \subseteq R \circ S \cap R \circ T \quad \wedge \quad (S \cap T) \circ R \subseteq S \circ T \cap T \circ R$	$\cap$-Fusion
$\left.\begin{aligned} (R \cup S) \cap T = (R \cap T) \cup (S \cap T) \\ T \cap (R \cup S) = (T \cap R) \cup (T \cap S) \end{aligned}\right\}$	$\cup$-Dist
$\left.\begin{aligned} (R \cap S) \cup T = (R \cup T) \cap (S \cup T) \\ T \cup (R \cap S) = (T \cup R) \cap (T \cup S) \end{aligned}\right\}$	$\cap$-Dist
$\Phi \subseteq id$	Φ-Def
$\Phi \circ \Psi = \Phi \cap \Psi$	Φ-Comp
$\Phi \circ \Phi = \Phi$	Φ-Refl
$\Phi^{\circ} = \Phi$	Φ-Conv
$(R \triangledown S) \circ \Phi? \circ T = (R \circ T \triangledown S \circ T) \circ (T^{\circ} \circ \Phi \circ T \cap id)?$	?-Fusion

$$(R \triangledown R) \circ \Phi? = R \qquad \textsc{?-Cancel}$$

$$id? = i_1 \qquad \textsc{?-True}$$

$$\bot? = i_2 \qquad \textsc{?-False}$$

$$! \circ R = ! \;\; \Leftarrow \;\; \bot \subset R \qquad \textsc{!-Fusion}$$

$$[\top] = id \qquad [\cdot]\textsc{-Top}$$

$$[\bot] = \bot \qquad [\cdot]\textsc{-Bot}$$

$$\pi_1 \circ [R] = (id \vartriangle R)^\circ \quad \wedge \quad \pi_2 \circ [R] = (R^\circ \vartriangle id)^\circ \qquad [\cdot]\textsc{-Cancel}$$

$$[\pi_2 \circ \Phi \circ {\pi_1}^\circ] = \Phi \qquad [\cdot]\textsc{-Cor}$$

$$\in_{1\mathsf{B}} \circ \mathsf{B}\ f\ g = f \circ \in_{1\mathsf{B}} \qquad \in_1\textsc{-Nat}$$

$$\in_{2\mathsf{B}} \circ \mathsf{B}\ f\ g = g \circ \in_{2\mathsf{B}} \qquad \in_2\textsc{-Nat}$$

$$\in_{\mathsf{F}} \circ \sigma_{\mathsf{F}} = \in_{\mathsf{F}} \times id \qquad \in\textsc{-Strength}$$

$$\in_{1\mathsf{B}} \circ \sigma_{\mathsf{B}} = \in_{1\mathsf{B}} \times id \qquad \in_1\textsc{-Strength}$$

$$\in_{2\mathsf{B}} \circ \sigma_{\mathsf{B}} = \in_{2\mathsf{B}} \times id \qquad \in_2\textsc{-Strength}$$

Proof (∈-STRENGTH).

$\in_{\mathsf{Id}} \circ \sigma_{\mathsf{Id}}$

$= \{\in - \textsc{Def}; \sigma\textsc{-Def}\}$

$id \circ id$

$= \{id - \textsc{Nat}; \times - \textsc{Reflex}\}$

$id \times id$

$= \{\in - \textsc{Def}\}$

$\in_{\mathsf{Id}} \times id$

$\in_{\underline{A}} \circ \sigma_{\underline{A}}$

$= \{\in - \textsc{Def}; \sigma\textsc{-Def}\}$

$\bot \circ \pi_1$

$= \{\bot\textsc{-Fusion}\}$

$\bot$

$= \{\bot\textsc{-Absor}; \bot\textsc{-Fusion}; \vartriangle - \textsc{Def}\}$

${\pi_1}^\circ \circ \bot \circ \pi_1 \;\cap\; {\pi_2}^\circ \circ \pi_2$

$$\begin{array}{l}
\quad = \{\vartriangle -\textsc{Def}; \times\text{-}\textsc{Def}\} \\
\bot \times \mathit{id} \\
\quad = \{\in - \textsc{Def}\} \\
\in_{\underline{A}} \times \mathit{id}
\end{array}$$

$$\begin{array}{l}
\in_{\mathsf{F}\otimes\mathsf{G}} \circ \sigma_{\mathsf{F}\otimes\mathsf{G}} \\
\quad = \{\in - \textsc{Def}; \sigma\text{-}\textsc{Def}\} \\
(\in_{\mathsf{F}} \circ \pi_1 \ \cup\ \in_{\mathsf{G}} \circ \pi_2) \circ (\sigma_{\mathsf{F}} \circ (\pi_1 \times \mathit{id}) \vartriangle \sigma_{\mathsf{G}} \circ (\pi_2 \times \mathit{id})) \\
\quad = \{\cup\text{-}\textsc{Fusion}; \times - \textsc{Cancel}\} \\
\in_{\mathsf{F}} \circ \sigma_{\mathsf{F}} \circ (\pi_1 \times \mathit{id}) \ \cup\ \in_{\mathsf{G}} \circ \sigma_{\mathsf{G}} \circ (\pi_2 \times \mathit{id}) \\
\quad = \{\in\text{-}\textsc{Strength}\} \\
(\in_{\mathsf{F}} \times \mathit{id}) \circ (\pi_1 \times \mathit{id}) \ \cup\ (\in_{\mathsf{G}} \times \mathit{id}) \circ (\pi_2 \times \mathit{id}) \\
\quad = \{\times\text{-}\textsc{Functor-Comp}\} \\
(\in_{\mathsf{F}} \circ \pi_1 \times \mathit{id}) \ \cup\ (\in_{\mathsf{G}} \circ \pi_2 \times \mathit{id}) \\
\quad = \{\times\text{-}\textsc{Def}\} \\
(\in_{\mathsf{F}} \circ \pi_1 \circ \pi_1 \vartriangle \mathit{id} \circ \pi_2) \ \cup\ (\in_{\mathsf{G}} \circ \pi_2 \circ \pi_1 \vartriangle \mathit{id} \circ \pi_2) \\
\quad = \{\vartriangle -\textsc{Def}; \cap\text{-}\textsc{Dist}; \cup\text{-}\textsc{Fusion}\} \\
(\in_{\mathsf{F}} \circ \pi_1 \circ \pi_1 \ \cup\ \in_{\mathsf{G}} \circ \pi_2 \circ \pi_1) \vartriangle \mathit{id} \circ \pi_2 \\
\quad = \{\cup\text{-}\textsc{Fusion}; \times\text{-}\textsc{Def}\} \\
(\in_{\mathsf{F}} \circ \pi_1 \ \cup\ \in_{\mathsf{G}} \circ \pi_2) \times \mathit{id} \\
\quad = \{\in - \textsc{Def}\} \\
\in_{\mathsf{F}\otimes\mathsf{G}} \times \mathit{id}
\end{array}$$

$$\begin{array}{l}
\in_{\mathsf{F}\oplus\mathsf{G}} \circ \sigma_{\mathsf{F}\oplus\mathsf{G}} \\
\quad = \{\in - \textsc{Def}; \sigma\text{-}\textsc{Def}\} \\
(\in_{\mathsf{F}} \triangledown \in_{\mathsf{G}}) \circ (\sigma_{\mathsf{F}} + \sigma_{\mathsf{G}}) \circ \mathit{distl} \\
\quad = \{+\text{-}\textsc{Absor}\} \\
(\in_{\mathsf{F}} \circ \sigma_{\mathsf{F}} \triangledown \in_{\mathsf{G}} \circ \sigma_{\mathsf{G}}) \circ \mathit{distl} \\
\quad = \{\in\text{-}\textsc{Strength}\} \\
((\in_{\mathsf{F}} \times \mathit{id}) \triangledown (\in_{\mathsf{G}} \times \mathit{id})) \circ \mathit{distl} \\
\quad = \{+\text{-}\textsc{Absor}; \mathit{distl} \circ ((R + S) \times T) = (R \times T + S \times T) \circ \mathit{distl}\} \\
(\mathit{id} \triangledown \mathit{id}) \circ \mathit{distl} \circ ((\in_{\mathsf{F}} + \in_{\mathsf{G}}) \times \mathit{id}) \\
\quad = \{\mathit{distl} - \mathit{id}\text{-}\textsc{Cancel}\} \\
((\mathit{id} \triangledown \mathit{id}) \times \mathit{id}) \circ ((\in_{\mathsf{F}} + \in_{\mathsf{G}}) \times \mathit{id}) \\
\quad = \{\times\text{-}\textsc{Functor-Comp}; +\text{-}\textsc{Absor}\}
\end{array}$$

$$
\begin{array}{l}
(\in_{\mathsf{F}} \nabla \in_{\mathsf{G}}) \times id \\
\;= \{\in - \textsc{Def}\} \\
\in_{\mathsf{F} \oplus \mathsf{G}} \times id
\end{array}
$$

$$
\begin{array}{l}
\in_{\mathsf{F} \odot \mathsf{G}} \circ \sigma_{\mathsf{F} \odot \mathsf{G}} \\
\;= \{\in - \textsc{Def}; \sigma\text{-}\textsc{Def}\} \\
\in_{\mathsf{G}} \circ \in_{\mathsf{F}} \circ \mathsf{F}\ \sigma_{\mathsf{G}} \circ \sigma_{\mathsf{F}} \\
\;= \{\in - \textsc{Nat}\} \\
\in_{\mathsf{G}} \circ \sigma_{\mathsf{G}} \circ \in_{\mathsf{F}} \circ \sigma_{\mathsf{F}} \\
\;= \{\in\text{-}\textsc{Strength}\} \\
(\in_{\mathsf{G}} \times id) \circ (\in_{\mathsf{F}} \times id) \\
\;= \{\times\text{-}\textsc{Functor}\text{-}\textsc{Comp}\} \\
(\in_{\mathsf{F}} \circ \in_{\mathsf{G}}) \times id \\
\;= \{\in - \textsc{Def}\} \\
\in_{\mathsf{F} \odot \mathsf{G}} \times id
\end{array}
$$

$$
\begin{array}{l}
\in_T \circ \sigma_T = \in_T \times id \\
\Leftrightarrow \{R = S \Leftrightarrow R \subseteq S \;\wedge\; S \subseteq R\} \\
\quad \in_T \circ \sigma_T \subseteq \in_T \times id \\
\quad \Leftrightarrow \{\times\text{-}\textsc{Def}; \times - \textsc{Univ}\} \\
\quad \pi_1 \circ \in_T \circ \sigma_T \subseteq \pi_1 \circ (\in_T \times id) \quad \wedge \quad \pi_2 \circ \in_T \circ \sigma_T \subseteq \pi_2 \circ (\in_T \times id) \\
\quad \Leftrightarrow \{\in - \textsc{Nat}; \sigma\text{-}\textsc{Cancel}; \pi_2 \circ \in_T \circ \sigma_T = \pi_2 \circ (\in_T \times id); \times\text{-}\textsc{Cancel}; \delta\pi_2 = id\} \\
\quad \textsc{True} \\
\\
\quad \in_T \times id \subseteq \in_T \circ \sigma_T \\
\quad \Leftrightarrow \{id\text{-}\textsc{Nat}; \times - \textsc{Reflex}; \times\text{-}\textsc{Fusion}\} \\
\qquad \pi_2 \circ (\in_T \circ \sigma_T) \circ (\in_T \circ \sigma_T)^\circ \subseteq \pi_2 \\
\qquad \Leftrightarrow \{R \circ f^\circ \subseteq S \Leftrightarrow R \subseteq S \circ f; \cdot^\circ\text{-}\textsc{Comp}\} \\
\qquad \pi_2 \circ (\in_T \circ \sigma_T) \circ (\pi_2 \circ \in_T \circ \sigma_T)^\circ \subseteq id \\
\qquad \Leftrightarrow \{\pi_2 \circ \in_T \circ \sigma_T = \pi_2 \circ (\in_T \times id); \times\text{-}\textsc{Cancel}; \} \\
\qquad \pi_2 \circ \delta(\in_T \circ \pi_1) \circ (\pi_2 \circ \delta(\in_T \circ \pi_1))^\circ \subseteq id \\
\qquad \Leftrightarrow \{\cdot^\circ\text{-}\textsc{Comp}; \Phi\text{-}\textsc{Conv}; \Phi\text{-}\textsc{Refl}\} \\
\qquad \pi_2 \circ \delta(\in_T \circ \pi_1) \circ \pi_2{}^\circ \subseteq id \\
\qquad \Leftrightarrow \{R \circ f^\circ \subseteq S \Leftrightarrow R \subseteq S \circ f; \Phi\text{-}\textsc{Def}\} \\
\qquad \textsc{True} \\
\quad \in_T \times id \subseteq (\pi_1 \circ \in_T \circ \sigma_T \vartriangle \pi_2 \circ \in_T \circ \sigma_T)
\end{array}
$$

$\Leftrightarrow \{\times - \textsc{Univ}\}$

$\pi_1 \circ (\in_T \times id) \subseteq \pi_1 \circ \in_T \circ \sigma_T \quad \wedge \quad \pi_2 \circ (\in_T \times id) \subseteq \pi_2 \circ \in_T \circ \sigma_T$

$\Leftrightarrow \{\in - \textsc{Nat}; \sigma\text{-}\textsc{Cancel}; \pi_2 \circ \in_T \circ \sigma_T = \pi_2 \circ (\in_T \times id); \times\text{-}\textsc{Cancel}; \delta\pi_2 = id\}$

TRUE

TRUE

Consider two entire relational catamorphisms $X_A = (\!|i_1 \circ \in_{1\mathsf{B}} \ \cup \ \in_{2\mathsf{B}} \ \cup \ i_2 \circ !\,|\!)_{\mathsf{B}\ A}$ and $Y_A = (\!|i_1 \circ \pi_2 \circ \in_{1\mathsf{B}} \ \cup \ \in_{2\mathsf{B}} \ \cup \ i_2 \circ !\,|\!)_{\mathsf{B}\ A}$:

$\pi_2 \circ \in_T \circ \sigma_T$

$= \{\in - \textsc{Def}; R \circ i_1{}^{\circ} = i_1{}^{\circ} \circ (R + S)\}$

$i_1{}^{\circ} \circ (\pi_2 + id) \circ X_{A \times B} \circ \sigma_T = \pi_2 \circ (\in_T \times id)$

$= \{\text{assumption} : (\pi_2 + id) \circ X_{A \times B} = Y_{A \times B}\}$

$(\pi_2 + id) \circ X_{A \times B} = Y_{A \times B}$

$\Leftarrow \{(\!|\cdot|\!)\text{-}\textsc{Fusion}\}$

$(\pi_2 + id) \circ (i_1 \circ \in_{1\mathsf{B}} \ \cup \ \in_{2\mathsf{B}} \ \cup \ i_2 \circ !\,)$

$= (i_1 \circ \pi_2 \circ \in_{1\mathsf{B}} \ \cup \ \in_{2\mathsf{B}} \ \cup \ i_2 \circ !\,) \circ \mathsf{B}\ id\ (\pi_2 + id)$

$\Leftrightarrow \{\cup\text{-}\textsc{Fusion}\}$

$(\pi_2 + id) \circ i_1 \circ \in_{1\mathsf{B}} \ \cup \ (\pi_2 + id) \circ \in_{2\mathsf{B}} \ \cup \ (\pi_2 + id) \circ i_2 \circ !$

$= i_1 \circ \pi_2 \circ \in_{1\mathsf{B}} \circ \mathsf{B}\ id\ (\pi_2 + id) \ \cup \ \in_{2\mathsf{B}} \circ \mathsf{B}\ id\ (\pi_2 + id) \ \cup \ i_2 \circ !$

$\circ\, \mathsf{B}\ id\ (\pi_2 + id)$

$\Leftrightarrow \{+\text{-}\textsc{Def}; +\text{-}\textsc{Cancel}; \in_1\text{-}\textsc{Nat}; \textsc{Bifunctor-Id}; \in_2\text{-}\textsc{Nat}; \ ! - \textsc{Fusion}\}$

TRUE

$i_1{}^{\circ} \circ Y_{A \times B} \circ \sigma_T$

$= \{\text{assumption} : Y_{A \times B} \circ \sigma_T = (\pi_2 + \pi_1) \circ distl \circ (X_A \times id)\}$

$Y_{A \times B} \circ \sigma_T = (\pi_2 + \pi_1) \circ distl \circ (X_A \times id)$

$\Leftrightarrow \{\sigma\text{-}\textsc{Def}; [\![\cdot, \cdot]\!]\text{-}\textsc{Uniq}\}$

$(i_1 \circ \pi_2 \circ \in_{1\mathsf{B}} \ \cup \ \in_{2\mathsf{B}} \ \cup \ i_2 \circ !\,) \circ \mathsf{B}\ id\ ((\pi_2 + \pi_1) \circ distl \circ (X_A \times id))$

$\circ\, \sigma_{\mathsf{B}} \circ (out_{\mathsf{B}\ A} \times id)$

$= (\pi_2 + \pi_1) \circ distl \circ (X_A \times id)$

$\Leftrightarrow \{\cup\text{-}\textsc{Fusion}; \textsc{Bifunctor-Comp}; \in_1\text{-}\textsc{Nat}; \in_2\text{-}\textsc{Nat}; \ !\,\text{-}\textsc{Fusion}\}$

$i_1 \circ \pi_2 \circ \in_{1\mathsf{B}} \circ \mathsf{B}\ id\ (X_A \times id) \circ \sigma_{\mathsf{B}} \circ (out_{\mathsf{B}\ A} \times id)$

$\cup\ (\pi_2 + \pi_1) \circ distl \circ \in_{2\mathsf{B}} \circ \mathsf{B}\ id\ (X_A \times id) \circ \sigma_{\mathsf{B}} \circ (out_{\mathsf{B}\ A} \times id) \ \cup \ i_2 \circ !$

$= (\pi_2 + \pi_1) \circ distl \circ (X_A \times id)$

$\Leftrightarrow \{\mathsf{B}\ id\ (R \times id) \circ \sigma_{\mathsf{B}} = \sigma_{\mathsf{B}} \circ (\mathsf{B}\ id\ R \times id); \times\text{-}\textsc{Functor-Comp}$

$; \in_1\text{-}\textsc{Strength}; \in_2\text{-}\textsc{Strength}\}$

$i_1 \circ \pi_2 \circ (\in_{1\mathsf{B}} \circ \mathsf{B}\ id\ X_A \circ out_{\mathsf{B}\ A} \times id)$

$$\begin{array}{l}
\quad \cup\ (\pi_2 + \pi_1) \circ \mathit{distl} \circ (\in_{2\mathsf{B}} \circ \mathsf{B}\ \mathit{id}\ X_A \circ \mathit{out}_{\mathsf{B}\ A} \times \mathit{id})\ \cup\ i_2 \circ !\\
\quad = (\pi_2 + \pi_1) \circ \mathit{distl} \circ (X_A \times \mathit{id})\\
\quad \Leftrightarrow \{\mathit{in} - \mathit{out} - \textsc{Iso};\ (\!|\cdot|\!)\text{-}\textsc{Cancel};\ \cup\text{-}\textsc{Fusion};\ \times\text{-}\textsc{Functor-Comp};\ \mathit{distl}\text{-}\textsc{Cancel}\}\\
\quad i_1 \circ \pi_2 \circ (\in_{1\mathsf{B}} \circ \mathsf{B}\ \mathit{id}\ X_A \circ \mathit{out}_{\mathsf{B}\ A} \times \mathit{id})\\
\quad \cup\ (\pi_2 + \pi_1) \circ \mathit{distl} \circ (\in_{2\mathsf{B}} \circ \mathsf{B}\ \mathit{id}\ X_A \circ \mathit{out}_{\mathsf{B}\ A} \times \mathit{id})\ \cup\ i_2 \circ !\\
\quad = (\pi_2 + \pi_1) \circ i_1 \circ (\in_{1\mathsf{B}} \circ B\ \mathit{id}\ X \circ \mathit{out}_{\mathsf{B}\ A} \times \mathit{id})\\
\quad \cup\ (\pi_2 + \pi_1) \circ \mathit{distl} \circ (\in_{2\mathsf{B}} \circ B\ \mathit{id}\ X \circ \mathit{out}_{\mathsf{B}\ A} \times \mathit{id})\\
\quad \cup\ (\pi_2 + \pi_1) \circ i_2 \circ (! \circ B\ \mathit{id}\ X \circ \mathit{out}_{\mathsf{B}\ A} \times \mathit{id})\\
\quad \Leftrightarrow \{+ - \textsc{Def};\ + - \textsc{Cancel};\ \times\text{-}\textsc{Cancel};\ !\text{-}\textsc{Fusion}\}\\
\quad \textsc{True}\\
i_1{}^{\circ} \circ (\pi_2 + \pi_1) \circ \mathit{distl} \circ (X_A \times \mathit{id})\\
\ = \{R \circ i_1{}^{\circ} = i_1{}^{\circ} \circ (R + S);\ \mathit{distl}\text{-}\textsc{Cancel};\ \cdot^{\circ}\text{-}\textsc{Comp};\ \cdot^{\circ}\text{-}\textsc{Prod}\}\\
\pi_2 \circ (i_1{}^{\circ} \times \mathit{id}) \circ \mathit{distl}^{\circ} \circ \mathit{distl} \circ (X_A \times \mathit{id})\\
\ = \{\mathit{distl} - \mathit{undistl}\text{-}\textsc{Iso};\ \times\text{-}\textsc{Functor-Comp};\ \in - \textsc{Def}\}\\
\pi_2 \circ (\in_T \times \mathit{id})
\end{array}$$

□

Appendix B

Additional Lens Laws and Proofs

B.1 State-based Lens Laws

$$((f \times g) \times h) \circ \mathit{assocl} = \mathit{assocl} \circ (f \times (g \times h)) \qquad \mathit{assocl}\text{-}\textsc{Nat}$$

$$\left.\begin{aligned} &{\pi_1}^{f \circ \pi_2} \circ \mathit{assocl} = \mathit{id} \times {\pi_1}^{f} \\ &({\pi_1}^{f} \times \mathit{id}) \circ \mathit{assocl} = \mathit{id} \times {\pi_2}^{f} \end{aligned}\right\} \qquad \mathit{assocl}\text{-}\pi_1\text{-}\textsc{Cancel}$$

$$\left.\begin{aligned} &{\pi_2}^{f} \circ \mathit{assocl} = {\pi_2}^{\pi_2 \circ f} \circ {\pi_2}^{\pi_1 \circ f \circ \pi_2} \\ &({\pi_2}^{f} \times \mathit{id}) \circ \mathit{assocl} = {\pi_2}^{f \circ \pi_1} \end{aligned}\right\} \qquad \mathit{assocl}\text{-}\pi_2\text{-}\textsc{Cancel}$$

$$(f \times (g \times h)) \circ \mathit{assocr} = \mathit{assocr} \circ ((f \times g) \times h) \qquad \mathit{assocr}\text{-}\textsc{Nat}$$

$$\left.\begin{aligned} &{\pi_1}^{f} \circ \mathit{assocr} = {\pi_1}^{\pi_1 \circ f} \circ {\pi_1}^{\pi_2 \circ f \circ \pi_1} \\ &(\mathit{id} \times {\pi_1}^{f}) \circ \mathit{assocr} = {\pi_1}^{f \circ \pi_2} \end{aligned}\right\} \qquad \mathit{assocr}\text{-}\pi_1\text{-}\textsc{Cancel}$$

$$\left.\begin{aligned} &{\pi_2}^{f \circ \pi_1} \circ \mathit{assocr} = {\pi_2}^{f} \times \mathit{id} \\ &(\mathit{id} \times {\pi_2}^{f}) \circ \mathit{assocr} = {\pi_1}^{f} \times \mathit{id} \end{aligned}\right\} \qquad \mathit{assocr}\text{-}\pi_2\text{-}\textsc{Cancel}$$

$$\mathit{assocl} \circ \mathit{assocr} = \mathit{id} \quad \wedge \quad \mathit{assocr} \circ \mathit{assocl} = \mathit{id} \qquad \mathit{assocl}\text{-}\mathit{assocr}\text{-}\textsc{Iso}$$

$$\mathit{coassocl} \circ (f + (g + h)) = ((f + g) + h) \circ \mathit{coassocl} \qquad \mathit{coassocl}\text{-}\textsc{Nat}$$

$$((f \triangledown g)^{p} \triangledown h)^{q} \circ \mathit{coassocl} = (f \triangledown (g \triangledown h)^{q})^{\hat{\wedge} \circ (p \,\vartriangle\, q)} \qquad \mathit{coassocl}\text{-}\textsc{Cancel}$$

$$\mathit{coassocr} \circ ((f + g) + h) = (f + (g + h)) \circ \mathit{coassocr} \qquad \mathit{coassocr}\text{-}\textsc{Nat}$$

$$(f \triangledown (g \triangledown h)^{p})^{q} \circ \mathit{coassocr} = ((f \triangledown g)^{q} \triangledown h)^{\hat{\wedge} \circ (p \,\vartriangle\, q)} \qquad \mathit{coassocr}\text{-}\textsc{Cancel}$$

$$\mathit{coassocl} \circ \mathit{coassocr} = \mathit{id} \qquad \mathit{coassocl}\text{-}\mathit{coassocr}\text{-}\textsc{Iso}$$

B.2 State-based Lens Proofs

B.2.1 Composition

Proof ($f \circ g$ is a well-behaved lens).

$$
\begin{aligned}
& get_{f \circ g} \circ create_{f \circ g} \\
& \;= \{\text{definition of } get; \text{definition of } create\} \\
& get_f \circ get_g \circ create_g \circ create_f \\
& \;= \{\textsc{CreateGet}\} \\
& id
\end{aligned}
$$

$$
\begin{aligned}
& get_{f \circ g} \circ put_{f \circ g} \\
& \;= \{\text{definition of } get; \text{definition of } put\} \\
& get_f \circ get_g \circ put_g \circ (put_f \circ (id \times get_g) \vartriangle \pi_2) \\
& \;= \{\textsc{PutGet}; \times - \textsc{Cancel}\} \\
& get_f \circ put_f \circ (id \times get_g) \\
& \;= \{\textsc{PutGet}; \times - \textsc{Def}; \times - \textsc{Cancel}; id - \textsc{Nat}\} \\
& \pi_1
\end{aligned}
$$

$$
\begin{aligned}
& put_{f \circ g} \circ (get_{f \circ g} \vartriangle id) \\
& \;= \{\text{definition of } put; \text{definition of } get\} \\
& put_g \circ (put_f \circ (id \times get_g) \vartriangle \pi_2) \circ (get_f \circ get_g \vartriangle id) \\
& \;= \{\times - \textsc{Fusion}; \times\text{-}\textsc{Absor}\} \\
& put_g \circ (put_f \circ (get_f \vartriangle id) \circ get_g \vartriangle id) \\
& \;= \{\textsc{GetPut}\} \\
& put_g \circ (id \circ get_g \vartriangle id) \\
& \;= \{id - \textsc{Nat}; \textsc{GetPut}\} \\
& id
\end{aligned}
$$

□

B.2.2 Catamorphisms

Proof ($put_{(\![f]\!)_{\mathsf{F}}}$ is a recursive anamorphism).

To prove that the put of our catamorphism lens terminates, we will reformulate it by tupling the forward transformation $get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} : \mu\mathsf{F} \to A$ with a curried backward transformation $put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} : \mu\mathsf{F} \to A \to \mu\mathsf{F}$, such that:

$$put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = \overline{put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ swap}$$

The Mutu Tupling Theorem, devised by Fokkinga (1989), allows to combine two different function traversing the same data structure (in a particular regular way) into a single catamorphism that traverses the data structure only once:

$$f \vartriangle g = \llparenthesis h \vartriangle i \rrparenthesis_{\mathsf{F}} \;\Leftarrow\; \begin{cases} f = h \circ \mathsf{F}\ (f \vartriangle g) \circ out_{\mathsf{F}} \\ g = i \circ \mathsf{F}\ (f \vartriangle g) \circ out_{\mathsf{F}} \end{cases} \qquad \text{MUTUTUPLING}$$

We can show that $get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}$ and $put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}$ can be tupled into the following catamorphism:

$$\begin{aligned} &get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \vartriangle put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \ :\ \mu\mathsf{F} \to A \times (A \to \mu\mathsf{F}) \\ &get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \vartriangle put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = \llparenthesis (get_f \circ \pi_1 \vartriangle \overline{aux}) \circ (\mathsf{F}\ \pi_1 \vartriangle \mathsf{F}\ \pi_2) \rrparenthesis_{\mathsf{F}} \end{aligned}$$

where $aux = in_{\mathsf{F}} \circ \mathsf{F}\ (ap \circ swap \triangledown create) \circ zip_{\mathsf{F}} \circ (put_f \times id) \circ assocl \circ swap$. Here, $zip_{\mathsf{F}} : \mathsf{F}\ A \times \mathsf{F}\ B \to \mathsf{F}\ (A \times B + A)$ is a polymorphic function that exposes the causes for the non-naturality of $fzip_{\mathsf{F}}\ f$, such that:

$$\mathsf{F}\ (id \triangledown (id \vartriangle f)) \circ zip_{\mathsf{F}} = fzip_{\mathsf{F}}\ f \qquad fzip\text{-}\mathrm{ZIP}$$

$$zip_{\mathsf{F}} \circ (\mathsf{F}\ f \times \mathsf{F}\ g) = \mathsf{F}\ (f \times g + f) \circ zip_{\mathsf{F}} \qquad zip\text{-}\mathrm{NAT}$$

We write the following proof:

$$get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \vartriangle put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = \llparenthesis (get_f \circ \pi_1 \vartriangle \overline{aux}) \circ (\mathsf{F}\ \pi_1 \vartriangle \mathsf{F}\ \pi_2) \rrparenthesis_{\mathsf{F}}$$

$\Leftrightarrow \{\times - \mathrm{UNIQ}; \mathrm{MUTUTUPLING}\}$

$$get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = \pi_1 \circ (get_f \circ \pi_1 \vartriangle \overline{aux}) \circ (\mathsf{F}\ \pi_1 \vartriangle \mathsf{F}\ \pi_2) \circ \mathsf{F}\ (get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \vartriangle put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}) \circ out_{\mathsf{F}}$$

$\Leftrightarrow \{\times - \mathrm{CANCEL}; \mathrm{FUNCTOR\text{-}COMP}; \times - \mathrm{CANCEL}\}$

$$get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = get_f \circ \mathsf{F}\ get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ out_{\mathsf{F}}$$

$\Leftrightarrow \{in - out - \mathrm{ISO}; \llparenthesis \cdot \rrparenthesis - \mathrm{CANCEL}\}$

TRUE

$$put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = \pi_2 \circ (get_f \circ \pi_1 \vartriangle \overline{aux}) \circ (\mathsf{F}\ \pi_1 \vartriangle \mathsf{F}\ \pi_2) \circ \mathsf{F}\ (get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \vartriangle put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}) \circ out_{\mathsf{F}}$$

$\Leftrightarrow \{\times - \mathrm{CANCEL}; \mathrm{EXP\text{-}FUSION}; \mathrm{EXP\text{-}UNIQ}; \mathrm{EXP\text{-}CANCEL}\}$

$$put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ swap = aux \circ ((\mathsf{F}\ \pi_1 \vartriangle \mathsf{F}\ \pi_2) \circ \mathsf{F}\ (get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \vartriangle put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}) \circ out_{\mathsf{F}} \times id)$$

$\Leftrightarrow \{\times - \textsc{Fusion}; \textsc{Functor-Comp}; \times - \textsc{Cancel}\}$

$put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ swap = aux \circ ((\mathsf{F}\ get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \vartriangle \mathsf{F}\ put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}) \circ out_{\mathsf{F}} \times id)$

$\Leftarrow \{swap - \textsc{Nat}; assocl\text{-}\textsc{Def}; \times - \textsc{Fusion}; \times - \textsc{Functor-Comp}$
$; \times - \textsc{Cancel}; \textsc{Leibniz}\}$

$put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = in_{\mathsf{F}} \circ \mathsf{F}\ (ap \circ swap \triangledown create_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}) \circ zip_{\mathsf{F}}$
$\circ (put_f \circ (id \times \mathsf{F}\ get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}) \vartriangle \mathsf{F}\ put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ \pi_2) \circ (id \times out_{\mathsf{F}})$

$\Leftarrow \{\text{definition of } put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}; in - out - \textsc{Iso}; \llbracket \cdot \rrbracket - \textsc{Cancel}; \times - \textsc{Functor-Comp}$
$; \textsc{Leibniz}\}$

$in_{\mathsf{F}} \circ \mathsf{F}\ put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ fzip_{\mathsf{F}}\ create_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}$
$= in_{\mathsf{F}} \circ \mathsf{F}\ (ap \circ swap \triangledown create_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}) \circ zip_{\mathsf{F}} \circ (id \times \mathsf{F}\ put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}})$

$\Leftrightarrow \{fzip\text{-}\textsc{Zip}; + - \textsc{Fusion}; \textsc{CreatePut}; \textsc{Functor-Id}; zip\text{-}\textsc{Nat}\}$

$in_{\mathsf{F}} \circ \mathsf{F}\ (put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \triangledown create_{\llparenthesis f \rrparenthesis_{\mathsf{F}}})$
$= in_{\mathsf{F}} \circ \mathsf{F}\ (ap \circ swap \triangledown create_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}) \circ \mathsf{F}\ (id \times put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} + id)$

$\Leftrightarrow \{\textsc{Functor-Comp}; +\text{-}\textsc{Absor}; swap - \textsc{Nat}; \textsc{Exp-Cancel}; swap - \textsc{Iso}\}$

TRUE

TRUE

Now, we can redefine $put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}$ as the expression $\hat{put'}_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ swap$, with $put'_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = \pi_1 \circ \llparenthesis (get_f \circ \pi_1 \vartriangle \overline{aux}) \circ (\mathsf{F}\ \pi_1 \vartriangle \mathsf{F}\ \pi_2) \rrparenthesis_{\mathsf{F}}$. Since a catamorphism is always a terminating function, $put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}}$ also terminates.

□

Proof ($\llparenthesis f \rrparenthesis_{\mathsf{F}}$ is a well-behaved lens).

$get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ create_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = id$

$\Leftrightarrow \{\text{definition of } get; \text{definition of } create\}$

$\llparenthesis get_f \rrparenthesis_{\mathsf{F}} \circ \llbracket create_f \rrbracket_{\mathsf{F}} = id$

$\Leftarrow \{\llbracket \cdot, \cdot \rrbracket - \textsc{Split}; \llbracket \cdot, \cdot \rrbracket - \textsc{Uniq}\}$

$get_f \circ \mathsf{F}\ id \circ create_f = id$

$\Leftrightarrow \{\textsc{Functor-Id}; id - \textsc{Nat}; \textsc{CreateGet}\}$

TRUE

$get_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ put_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} = \pi_1$

$\Leftrightarrow \{\text{definition of } get; \text{definition of } put\}$

$\llparenthesis get_f \rrparenthesis_{\mathsf{F}} \circ \llbracket fzip_{\mathsf{F}}\ create_{\llparenthesis f \rrparenthesis_{\mathsf{F}}} \circ (put_f \circ (id \times \mathsf{F}\ get) \vartriangle \pi_2) \circ (id \times out_{\mathsf{F}}) \rrbracket_{\mathsf{F}} = \pi_1$

$\Leftarrow \{\llbracket \cdot, \cdot \rrbracket - \textsc{Split}; \llbracket \cdot, \cdot \rrbracket - \textsc{Uniq}\}$

$$get_f \circ \mathsf{F}\ \pi_1 \circ fzip_{\mathsf{F}}\ create_{(\!|f|\!)_{\mathsf{F}}} \circ (put_f \circ (id \times \mathsf{F}\ get) \vartriangle \pi_2) \circ (id \times out_{\mathsf{F}}) = \pi_1$$

$\Leftrightarrow \{ fzip - \textsc{Cancel}; \times - \textsc{Cancel}; \textsc{PutGet} \}$

$$\pi_1 \circ (id \times \mathsf{F}\ get) \circ (id \times out_{\mathsf{F}}) = \pi_1$$

$\Leftrightarrow \{ \times - \textsc{Functor-Comp}; \times - \textsc{Def}; \times - \textsc{Cancel} \}$

TRUE

$$put_{(\!|f|\!)_{\mathsf{F}}} \circ (get_{(\!|f|\!)_{\mathsf{F}}} \vartriangle id) = id$$

$\Leftrightarrow \{ \text{definition of } put; \llbracket \cdot \rrbracket - \textsc{Reflex} \}$

$$\llbracket fzip_{\mathsf{F}}\ create_{(\!|f|\!)_{\mathsf{F}}} \circ (put_f \circ (id \times \mathsf{F}\ get) \vartriangle \pi_2) \circ (id \times out_{\mathsf{F}}) \rrbracket_{\mu\mathsf{F}} \circ (get \vartriangle id)$$
$$= \llbracket out_{\mathsf{F}} \rrbracket_{\mathsf{F}}$$

$\Leftarrow \{ \llbracket \cdot \rrbracket - \textsc{Fusion} \}$

$$fzip_{\mathsf{F}}\ create_{(\!|f|\!)_{\mathsf{F}}} \circ (put_f \circ (id \times \mathsf{F}\ get) \vartriangle \pi_2) \circ (id \times out_{\mathsf{F}}) \circ (get \vartriangle id)$$
$$= \mathsf{F}\ (get \vartriangle id) \circ out_{\mathsf{F}}$$

$\Leftrightarrow \{ \times\text{-}\textsc{Absor}; \times - \textsc{Fusion}; \times\text{-}\textsc{Absor}; \times - \textsc{Cancel} \}$

$$fzip_{\mathsf{F}}\ create_{(\!|f|\!)_{\mathsf{F}}} \circ (put_f \circ (get \vartriangle \mathsf{F}\ get \circ out_{\mathsf{F}}) \vartriangle out_{\mathsf{F}}) = \mathsf{F}\ (get \vartriangle id) \circ out_{\mathsf{F}}$$

$\Leftrightarrow \{ in - out - \textsc{Iso}; (\!| \cdot |\!) - \textsc{Cancel}; \times - \textsc{Fusion} \}$

$$fzip_{\mathsf{F}}\ create_{(\!|f|\!)_{\mathsf{F}}} \circ (put_f\ (get_f \vartriangle id) \circ \mathsf{F}\ get \circ out_{\mathsf{F}} \vartriangle out_{\mathsf{F}})$$
$$= \mathsf{F}\ (get \vartriangle id) \circ out_{\mathsf{F}}$$

$\Leftrightarrow \{ \textsc{GetPut}; id - \textsc{Nat}; \times - \textsc{Fusion} \}$

$$fzip_{\mathsf{F}}\ create_{(\!|f|\!)_{\mathsf{F}}} \circ (\mathsf{F}\ get \vartriangle id) \circ out_{\mathsf{F}} = \mathsf{F}\ (get \vartriangle id) \circ out_{\mathsf{F}}$$

$\Leftrightarrow \{ fzip - \textsc{Split} \}$

TRUE

□

Proof ($(\!|\cdot|\!)$-UNIQ).

$$create_f = create_{(\!|g|\!)_{\mathsf{F}}}$$

$\Leftrightarrow \{ \text{definition of } create \}$

$$create_f = \llbracket create_g \rrbracket_{\mathsf{F}}$$

$\Leftrightarrow \{ \llbracket \cdot \rrbracket - \textsc{Uniq} \}$

$$out_{\mathsf{F}} \circ create_f = \mathsf{F}\ create_f \circ create_g$$

$\Leftrightarrow \{ \text{definition of } create \}$

$$create_{in_{\mathsf{F}}} \circ create_f = create_{\mathsf{F}\ f} \circ create_g$$

$\Leftrightarrow \{ \text{definition of } create \}$

$$create_{f \circ in_{\mathsf{F}}} = create_{g \circ \mathsf{F}\ f}$$

$$
\begin{array}{l}
put_f = put_{(\!|g|\!)_{\mathsf{F}}} \\
\Leftrightarrow \{ \text{definition of } put \} \\
put_f = \llbracket(fzip_{\mathsf{F}}\ create_{(\!|g|\!)_{\mathsf{F}}} \circ (put_g \circ (id \times \mathsf{F}\ get_{(\!|g|\!)_{\mathsf{F}}}) \mathbin{\triangle} \pi_2) \circ (id \times out_{\mathsf{F}}))\rrbracket_{\mathsf{F}} \\
\Leftrightarrow \{ \llbracket(\cdot)\rrbracket - \textsc{Uniq} \} \\
out_{\mathsf{F}} \circ put_f \\
\quad = \mathsf{F}\ put_f \circ fzip_{\mathsf{F}}\ create_{(\!|g|\!)_{\mathsf{F}}} \circ (put_g \circ (id \times \mathsf{F}\ get_{(\!|g|\!)_{\mathsf{F}}}) \mathbin{\triangle} \pi_2) \circ (id \times out_{\mathsf{F}}) \\
\Leftrightarrow \{ get_f = get_{(\!|g|\!)_{\mathsf{F}}} ; create_f = create_{(\!|g|\!)_{\mathsf{F}}} \} \\
out_{\mathsf{F}} \circ put_f \\
\quad = \mathsf{F}\ put_f \circ fzip_{\mathsf{F}}\ create_f \circ (put_g \circ (id \times \mathsf{F}\ get_f) \mathbin{\triangle} \pi_2) \circ (id \times out_{\mathsf{F}}) \\
\Leftrightarrow \{ fzip\text{-}\textsc{Put}; \text{definition of } get \} \\
out_{\mathsf{F}} \circ put_f = put_{\mathsf{F}\ f} \circ (put_g \circ (id \times get_{\mathsf{F}\ f}) \mathbin{\triangle} \pi_2) \circ (id \times out_{\mathsf{F}}) \\
\Leftrightarrow \{ \text{definition of } put ; \times - \textsc{Reflex} \} \\
out_{\mathsf{F}} \circ put_f = put_{g \circ \mathsf{F}\ f} \circ (id \times out_{\mathsf{F}}) \\
\Leftrightarrow \{ in - out - \textsc{Iso} ; \times - \textsc{Functor-Comp} ; \textsc{Leibniz} \} \\
out_{\mathsf{F}} \circ put_f \circ (id \times in_{\mathsf{F}}) = put_{g \circ \mathsf{F}\ f} \\
\Leftrightarrow \{ \times - \textsc{Cancel} ; \text{definition of } put \} \\
put_{in_{\mathsf{F}}} \circ (put_f \circ (id \times in_{\mathsf{F}}) \mathbin{\triangle} \pi_2) = put_{g \circ \mathsf{F}\ f} \\
\Leftrightarrow \{ \text{definition of } get ; \text{definition of } put \} \\
put_{f \circ in_{\mathsf{F}}} = put_{g \circ \mathsf{F}\ f}
\end{array}
$$

□

B.2.3 Anamorphisms

Proof ($put_{\llbracket f \rrbracket_{\mathsf{G}}}$ is a recursive hylomorphism).

To prove that the put of our anamorphism lens terminates, we will reformulate it using an alternative version $put'_{\llbracket f \rrbracket_{\mathsf{G}}} : \mu\mathsf{G} \times A + \mu\mathsf{G} \rightarrow A$ that exposes the causes for the non-naturality of $fzip_{\mathsf{G}}\ f$ by partitioning it into a polymorphic function zip_{G} and the application of f inside the functor G. Formally, we write:

$$put'_{\llbracket f \rrbracket_{\mathsf{G}}} = \llbracket put_f \mathbin{\nabla} create_{\llbracket f \rrbracket_{\mathsf{G}}}, zip_{\mathsf{G}} \circ (out_{\mathsf{G}} \times get_f) \mathbin{\triangle} \pi_2 + id \rrbracket_{\mathsf{G} \otimes \underline{A} \oplus \underline{\mu\mathsf{G}}}$$

Since $zip_{\mathsf{F}} : \mathsf{F}\ A \times \mathsf{F}\ B \rightarrow \mathsf{F}\ (A \times B + A)$ is polymorphic on A and B, we can prove the following property ruling its interaction with membership:

$$\in_{\mathsf{F}} \circ\, zip_{\mathsf{F}} \subseteq (id \mathbin{\nabla} \pi_1)^{\circ} \circ (\in_{\mathsf{F}} \times \in_{\mathsf{F}}) \qquad \in\text{-}\textsc{Zip}$$

Informally, this law says that any pair (a, b) that is a member of the zipped F-structure is so that a is a member of the left F-structure and b is a member of the right F-structure (and similarly for lone a values).

After writing the auxiliary proof

$$\begin{array}{l}
\mathit{put}'_{\llbracket f \rrbracket_{\mathsf{G}}} = \mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}} \\
\Leftrightarrow \{\text{definition of } \mathit{put}'_{\llbracket f \rrbracket_{\mathsf{G}}} ; \llbracket \cdot, \cdot \rrbracket - \textsc{Uniq}\} \\
(\mathit{put}_f \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}}) \circ ((\mathsf{G} \otimes \underline{A} \oplus \underline{\mu \mathsf{G}})\ (\mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}})) \\
\circ\ (\mathit{zip}_{\mathsf{G}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2 + \mathit{id}) = \mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}} \\
\Leftrightarrow \{\textsc{Functor-Def}; + - \textsc{Functor-Comp}; \times\text{-}\textsc{Absor}\} \\
(\mathit{put}_f \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}}) \circ (\mathsf{G}\ (\mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}}) \circ \mathit{zip}_{\mathsf{G}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2 + \mathit{id}) \\
= \mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}} \\
\Leftrightarrow \{+\text{-}\textsc{Absor}; +\text{-}\textsc{Equal}\} \\
\mathit{put}_f \circ (\mathsf{G}\ (\mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}}) \circ \mathit{zip}_{\mathsf{G}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2) = \mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \\
\Leftrightarrow \{\text{definition of } \mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} ; \llbracket \cdot, \cdot \rrbracket\text{-}\textsc{Cancel}\} \\
\mathit{put}_f \circ (\mathsf{G}\ (\mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}}) \circ \mathit{zip}_{\mathsf{G}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2) \\
= \mathit{put}_f \circ (\mathsf{G} \otimes \underline{A})\ \mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (\mathit{fzip}_{\mathsf{G}}\ \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2) \\
\Leftrightarrow \{\textsc{Functor-Def}; \times\text{-}\textsc{Absor}; \mathit{fzip}\text{-}\textsc{Zip}\} \\
\mathit{put}_f \circ (\mathsf{G}\ (\mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}}) \circ \mathit{zip}_{\mathsf{G}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2) \\
= \mathit{put}_f \circ (\mathsf{G}\ \mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \circ \mathsf{G}\ (\mathit{id} \nabla (\mathit{id} \triangle \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}})) \circ \mathit{zip}_{\mathsf{G}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2) \\
\Leftrightarrow \{\textsc{Functor-Comp}; + - \textsc{Fusion}; \textsc{CreatePut}\} \\
\textsc{True}
\end{array}$$

, we can trivially show the following equivalence:

$$\begin{array}{l}
\mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} = \mathit{put}'_{\llbracket f \rrbracket_{\mathsf{G}}} \circ i_1 \\
\Leftrightarrow \{\mathit{put}'_{\llbracket f \rrbracket_{\mathsf{G}}} = \mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}} \nabla \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}} ; + - \textsc{Cancel}\} \\
\textsc{True}
\end{array}$$

Now, proving that $\mathit{put}_{\llbracket f \rrbracket_{\mathsf{G}}}$ is a recursive hylomorphism amounts to proving that $\mathit{put}'_{\llbracket f \rrbracket_{\mathsf{G}}} \circ i_1$ terminates. More specifically, we need to prove that the respective anamorphism is recursive:

$$\begin{array}{l}
\llbracket \mathit{zip}_{\mathsf{G}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2 + \mathit{id} \rrbracket_{\mathsf{G} \otimes \underline{A} \oplus \underline{\mu \mathsf{G}}} \text{ recursive} \\
\equiv\ \{\llbracket g \rrbracket_{\mathsf{F}} \text{ recursive} \Leftrightarrow \in_{\mathsf{F}} \circ\, g \text{ well} - \text{founded}\} \\
\in_{\mathsf{G} \otimes \underline{A} \oplus \underline{\mu \mathsf{G}}} \circ\, (\mathit{zip}_{\mathsf{G}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_f) \triangle \pi_2 + \mathit{id}) \text{ well} - \text{founded} \\
\equiv\ \{\in - \textsc{Def}\}
\end{array}$$

$((\in_{\mathsf{G}} \circ \pi_1 \;\cup\; \bot \circ \pi_2) \triangledown \bot) \circ (zip_{\mathsf{G}} \circ (out_{\mathsf{G}} \times get_f) \vartriangle \pi_2 + id)$ well − founded

$\equiv$ { $\triangledown$ −DEF; $\cup$-FUSION; $\bot$-FUSION; $\bot$-NEUTRAL }

$\in_{\mathsf{G}} \circ \pi_1 \circ i_1{}^{\circ} \circ (zip_{\mathsf{G}} \circ (out_{\mathsf{G}} \times get_f) \vartriangle \pi_2 + id)$ well − founded

$\equiv$ { $i_1{}^{\circ} \circ (R + S) = R \circ i_1{}^{\circ}$; $\times$ − CANCEL }

$\in_{\mathsf{G}} \circ (zip_{\mathsf{G}} \circ (out_{\mathsf{G}} \times get_f)) \circ i_1{}^{\circ}$ well − founded

$\Leftarrow$ { $\in$-ZIP; $\times$-FUNCTOR-COMP; $S \subseteq R \;\wedge\; R$ well − founded $\Rightarrow S$ well − founded }

$(id \triangledown \pi_1)^{\circ} \circ (\in_{\mathsf{G}} \circ out_{\mathsf{G}} \times \in_{\mathsf{G}} \circ get_f) \circ i_1{}^{\circ}$ well − founded

$\equiv$ { $\triangledown$ −DEF; $\cdot^{\circ}$-DIST; $\cdot^{\circ}$-COMP; $\cdot^{\circ}$-INV }

$(i_1 \;\cup\; i_2 \circ \pi_1) \circ (\in_{\mathsf{G}} \circ out_{\mathsf{G}} \times \in_{\mathsf{G}} \circ get_f) \circ i_1{}^{\circ}$ well − founded

$\equiv$ { $\cup$-FUSION }

$i_1 \circ (\in_{\mathsf{G}} \circ out_{\mathsf{G}} \times \in_{\mathsf{G}} \circ get_f) \circ i_1{}^{\circ} \;\cup\; i_2 \circ \in_{\mathsf{G}} \circ out_{\mathsf{G}} \circ \pi_1 \circ i_1{}^{\circ}$ well − founded

$\Leftarrow$ { R, S well − founded $\wedge\; R \circ S \subseteq R \Rightarrow R \;\cup\; S$ well − founded }

> $i_1 \circ (\in_{\mathsf{G}} \circ out_{\mathsf{G}} \times \in_{\mathsf{G}} \circ get_f) \circ i_1{}^{\circ}$ well − founded
>
> $\Leftrightarrow$ { out_{G} recursive; get_f recursive; $\llbracket g \rrbracket_{\mathsf{F}}$ recursive $\Leftrightarrow \in_{\mathsf{F}} \circ g$ well − founded }
>
> TRUE
>
> $i_2 \circ \in_{\mathsf{G}} \circ out_{\mathsf{G}} \circ \pi_1 \circ i_1{}^{\circ}$ well − founded
>
> $\Leftrightarrow$ { out_{G} recursive; $\llbracket g \rrbracket_{\mathsf{F}}$ recursive $\Leftrightarrow \in_{\mathsf{F}} \circ g$ well − founded }
>
> TRUE
>
> $i_1 \circ (\in_{\mathsf{G}} \circ out_{\mathsf{G}} \times \in_{\mathsf{G}} \circ get_f) \circ i_1{}^{\circ} \circ i_2 \circ \in_{\mathsf{G}} \circ out_{\mathsf{G}} \circ \pi_1 \circ i_1{}^{\circ}$
>
> $\subseteq i_1 \circ (\in_{\mathsf{G}} \circ out_{\mathsf{G}} \times \in_{\mathsf{G}} \circ get_f) \circ i_1{}^{\circ}$
>
> $\Leftrightarrow$ { $i_1{}^{\circ} \circ i_2 = \bot$; $\bot$-FUSION }
>
> $\bot \subseteq i_1 \circ (\in_{\mathsf{G}} \circ out_{\mathsf{G}} \times \in_{\mathsf{G}} \circ get_f) \circ i_1{}^{\circ}$
>
> $\Leftrightarrow$ { $\bot \subseteq R$ }
>
> TRUE

TRUE

□

Proof ($\llbracket f \rrbracket_{\mathsf{G}}$ is a well-behaved lens).

$get_{\llbracket f \rrbracket_{\mathsf{G}}} \circ create_{\llbracket f \rrbracket_{\mathsf{G}}} = id$

$\Leftrightarrow$ { definition of get; $\llbracket \cdot \rrbracket$ − REFLEX }

$\llbracket get_f \rrbracket_{\mathsf{G}} \circ create_{\llbracket f \rrbracket_{\mathsf{G}}} = \llbracket out_{\mathsf{G}} \rrbracket_{\mathsf{G}}$

$\Leftarrow$ { $\llbracket \cdot \rrbracket$ − FUSION }

$get_f \circ create_{\llbracket f \rrbracket_{\mathsf{G}}} = \mathsf{G}\; create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ out_{\mathsf{G}}$

$\Leftrightarrow$ { definition of $create$; $(\!|\cdot|\!)$ − CANCEL }

$$get_f \circ create_f \circ \mathsf{G}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ out_{\mathsf{G}} = \mathsf{G}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ out_{\mathsf{G}}$$

$$\Leftarrow \{\textsc{Leibniz}\}$$

$$get_f \circ create_f = id$$

$$\Leftrightarrow \{\textsc{CreateGet}\}$$

$$\textsc{True}$$

$$get_{\llbracket f \rrbracket_{\mathsf{G}}} \circ put_{\llbracket f \rrbracket_{\mathsf{G}}}$$

$$= \{\text{definition of } get\}$$

$$\llbracket get_f \rrbracket_{\mathsf{G}} \circ put_{\llbracket f \rrbracket_{\mathsf{G}}}$$

$$= \{\llbracket \cdot \rrbracket - \textsc{Fusion}; \text{definition of } put\}$$

$$\quad get_f \circ \llbracket put_f, fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (out_{\mathsf{G}} \times get_f) \vartriangle \pi_2 \rrbracket_{\mathsf{G} \otimes \underline{A}} = \mathsf{G}\ put \circ h$$

$$\quad \Leftrightarrow \{\llbracket \cdot, \cdot \rrbracket\text{-}\textsc{Cancel}\}$$

$$\quad get_f \circ put_f \circ (\mathsf{G} \otimes \underline{A})\ put \circ (fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (out_{\mathsf{G}} \times get_f) \vartriangle \pi_2)$$

$$\quad = \mathsf{G}\ put \circ h$$

$$\quad \Leftrightarrow \{\textsc{PutGet}; \times - \textsc{Functor-Comp}\}$$

$$\quad \pi_1 \circ (\mathsf{G}\ put \circ fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (out_{\mathsf{G}} \times get_f) \vartriangle \pi_2) = \mathsf{G}\ put \circ h$$

$$\quad \Leftrightarrow \{\times - \textsc{Cancel}\}$$

$$\quad \mathsf{G}\ put \circ fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (out_{\mathsf{G}} \times get_f)$$

$$\quad = \mathsf{G}\ put \circ fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (out_{\mathsf{G}} \times get_f)$$

$$\llbracket fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (out_{\mathsf{G}} \times get_f) \rrbracket_{\mathsf{G}}$$

$$= \{\llbracket \cdot \rrbracket - \textsc{Fusion}\}$$

$$\quad \mathsf{G}\ \pi_1 \circ fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (out_{\mathsf{G}} \times get_f) = out_{\mathsf{G}} \circ \pi_1$$

$$\quad \Leftrightarrow \{fzip - \textsc{Cancel}; \times - \textsc{Cancel}\}$$

$$\quad out_{\mathsf{G}} \circ \pi_1 = out_{\mathsf{G}} \circ \pi_1$$

$$\llbracket out_{\mathsf{G}} \rrbracket_{\mathsf{G}} \circ \pi_1$$

$$= \{\llbracket \cdot \rrbracket - \textsc{Reflex}\}$$

$$\pi_1$$

$$put_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (get_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle id)$$

$$= \{\text{definition of } put\}$$

$$\llbracket put_f, (fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (id \times get_f) \vartriangle \pi_2) \circ (out_{\mathsf{G}} \times id) \rrbracket_{\mathsf{G} \otimes \underline{A}} \circ (get_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle id)$$

$$= \{\llbracket \cdot, \cdot \rrbracket\text{-}\textsc{Fusion}\}$$

$$\quad (fzip_{\mathsf{G}}\ create_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (id \times get_f) \vartriangle \pi_2) \circ (out_{\mathsf{G}} \times id) \circ (get_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle id)$$

$$\quad = (\mathsf{G} \otimes \underline{A})\ (get_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle id) \circ (get_f \vartriangle id)$$

$$\quad \Leftrightarrow \{\times\text{-}\textsc{Absor}; \times\text{-}\textsc{Absor}; \times - \textsc{Cancel}\}$$

$$\mathit{fzip}_{\mathsf{G}}\ \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (\mathit{out}_{\mathsf{G}} \circ \mathit{get}_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle \mathit{get}_f) \vartriangle \mathit{id}$$
$$= (\mathsf{G}\ (\mathit{get}_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle \mathit{id}) \times \mathit{id}) \circ (\mathit{get}_f \vartriangle \mathit{id})$$
$$\Leftrightarrow \{ \llbracket\cdot\rrbracket - \textsc{Cancel}; \times - \textsc{Fusion} \}$$
$$\mathit{fzip}_{\mathsf{G}}\ \mathit{create}_{\llbracket f \rrbracket_{\mathsf{G}}} \circ (\mathsf{G}\ \mathit{get}_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle \mathit{id}) \circ \mathit{get}_f \vartriangle \mathit{id}$$
$$= (\mathsf{G}\ (\mathit{get}_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle \mathit{id}) \times \mathit{id}) \circ (\mathit{get}_f \vartriangle \mathit{id})$$
$$\Leftrightarrow \{ \mathit{fzip} - \textsc{Split}; \times\text{-}\textsc{Absor} \}$$
$$(\mathsf{G}\ (\mathit{get}_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle \mathit{id}) \circ \mathit{get}_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle \mathit{id}) = (\mathsf{G}\ (\mathit{get}_{\llbracket f \rrbracket_{\mathsf{G}}} \vartriangle \mathit{id}) \circ \mathit{get}_f \vartriangle \mathit{id})$$
$$\llbracket \mathit{put}_f, \mathit{get}_f \vartriangle \mathit{id} \rrbracket_{\mathsf{G} \otimes \underline{A}} = \mathit{id}$$
$$= \{ \llbracket \cdot, \cdot \rrbracket - \textsc{Uniq}; \textsc{GetPut} \}$$
$$\mathit{id}$$

□

Proof ($\llbracket\cdot\rrbracket$-Uniq).

$$\mathit{create}_f = \mathit{create}_{\llbracket g \rrbracket_{\mathsf{G}}}$$
$$\Leftrightarrow \{ \text{definition of } \mathit{create} \}$$
$$\mathit{create}_f = \llparenthesis \mathit{create}_g \rrparenthesis_{\mathsf{G}}$$
$$\Leftrightarrow \{ \llparenthesis\cdot\rrparenthesis - \textsc{Uniq} \}$$
$$\mathit{create}_f \circ \mathit{in}_{\mathsf{G}} = \mathit{create}_g \circ \mathsf{G}\ \mathit{create}_f$$
$$\Leftrightarrow \{ \text{definition of } \mathit{create} \}$$
$$\mathit{create}_f \circ \mathit{create}_{\mathit{out}_{\mathsf{G}}} = \mathit{create}_g \circ \mathit{create}_{\mathsf{G}\ f}$$
$$\Leftrightarrow \{ \text{definition of } \mathit{create} \}$$
$$\mathit{create}_{\mathit{out}_{\mathsf{G}} \circ f} = \mathit{create}_{\mathsf{G}\ f \circ g}$$

$$\mathit{put}_f = \mathit{put}_{\llbracket g \rrbracket_{\mathsf{G}}}$$
$$\Leftrightarrow \{ \text{definition of } \mathit{put} \}$$
$$\mathit{put}_f = \llbracket \mathit{put}_g, \mathit{fzip}_{\mathsf{G}}\ \mathit{create}_{\llbracket g \rrbracket_{\mathsf{G}}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_g) \vartriangle \pi_2 \rrbracket_{\mathsf{G} \otimes \underline{A}}$$
$$\Leftrightarrow \{ \llbracket \cdot, \cdot \rrbracket - \textsc{Uniq} \}$$
$$\mathit{put}_f = \mathit{put}_g \circ (\mathsf{G} \otimes \underline{A})\ \mathit{put}_f \circ (\mathit{fzip}_{\mathsf{G}}\ \mathit{create}_{\llbracket g \rrbracket_{\mathsf{G}}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_g) \vartriangle \pi_2)$$
$$\Leftrightarrow \{ \textsc{Functor-Def}; \times\text{-}\textsc{Absor} \}$$
$$\mathit{put}_f = \mathit{put}_g \circ (\mathsf{G}\ \mathit{put}_f \circ \mathit{fzip}_{\mathsf{F}}\ \mathit{create}_{\llbracket g \rrbracket_{\mathsf{G}}} \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_g) \vartriangle \pi_2)$$
$$\Leftrightarrow \{ \mathit{create}_f = \mathit{create}_{\llbracket g \rrbracket_{\mathsf{G}}} \}$$
$$\mathit{put}_f = \mathit{put}_g \circ (\mathsf{G}\ \mathit{put}_f \circ \mathit{fzip}_{\mathsf{G}}\ \mathit{create}_f \circ (\mathit{out}_{\mathsf{G}} \times \mathit{get}_g) \vartriangle \pi_2)$$
$$\Leftrightarrow \{ \mathit{fzip}\text{-}\textsc{Put}; \times - \textsc{Functor-Comp}; \times - \textsc{Cancel} \}$$
$$\mathit{put}_f = \mathit{put}_g \circ (\mathit{put}_{\mathsf{G}\ f} \circ (\mathit{id} \times \mathit{get}_g) \circ (\mathit{out}_{\mathsf{G}} \times \mathit{id}) \vartriangle \pi_2 \circ (\mathit{out}_{\mathsf{G}} \times \mathit{id}))$$

$\Leftrightarrow \{\times - \textsc{Fusion}; \text{definition of } put\}$

$put_f = put_{\mathsf{G}\, f \circ g} \circ (out_{\mathsf{G}} \times id)$

$\Leftrightarrow \{in - out - \textsc{Iso}; \times - \textsc{Functor-Comp}; \textsc{Leibniz}\}$

$put_f \circ (in_{\mathsf{G}} \times id) = put_{\mathsf{G}\, f \circ g}$

$\Leftrightarrow \{\times - \textsc{Def}; \times - \textsc{Cancel}; \text{definition of } put\}$

$put_f \circ (put_{out_{\mathsf{G}}} \circ (id \times get_f) \bigtriangleup \pi_2) = put_{\mathsf{G}\, f \circ g}$

$\Leftrightarrow \{\text{definition of } put\}$

$put_{out_{\mathsf{G}} \circ f} = put_{\mathsf{G}\, f \circ g}$

□

B.2.4 Natural Transformations

Proof ($(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}$ *is a well-behaved lens* $\Leftarrow$ $get_f : \mathsf{F} \stackrel{\cdot}{\to} \mathsf{G}$).

To write this proof (without having to separately prove that the anamorphisms are recursive), we only need to redo the auxiliary proofs without resorting to the $[\![\cdot, \cdot]\!]$-$\textsc{Uniq}$ law (that requires the anamorphisms to be recursive).

$get_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} = id$

$\Leftrightarrow \{\text{definition of } get; (\!|\cdot|\!) - [\!(\cdot)\!]\text{-}\textsc{Shift}\}$

$[\!(get_f \circ out_{\mathsf{F}})\!]_{\mathsf{F}} \circ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} = id$

$\Leftrightarrow \{[\!(\cdot)\!] - \textsc{Reflex}; [\!(\cdot)\!] - \textsc{Fusion}\}$

$\quad get_f \circ out_{\mathsf{F}} \circ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} = \mathsf{G}\ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ out_{\mathsf{G}}$

$\quad \Leftrightarrow \{\text{definition of } create; [\!(\cdot)\!] - \textsc{Cancel}\}$

$\quad get_f \circ \mathsf{F}\ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ create_f \circ out_{\mathsf{G}} = \mathsf{G}\ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ out_{\mathsf{G}}$

$\quad \Leftrightarrow \{\eta - \textsc{Nat}; \textsc{CreateGet}\}$

$\quad \mathsf{G}\ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ out_{\mathsf{G}} = \mathsf{G}\ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ out_{\mathsf{G}}$

$\textsc{True}$

$get_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ put_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} = \pi_1$

$\Leftrightarrow \{\text{definition of } get; (\!|\cdot|\!) - [\!(\cdot)\!]\text{-}\textsc{Shift}\}$

$[\!(get_f \circ out_{\mathsf{F}})\!]_{\mathsf{G}} \circ put_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} = \pi_1$

$\Leftrightarrow \{[\!(\cdot)\!] - \textsc{Fusion}\}$

$\quad get_f \circ out_{\mathsf{F}} \circ put_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} = \mathsf{G}\ put_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ h$

$\quad \Leftrightarrow \{\text{definition of } put; [\!(\cdot)\!] - \textsc{Cancel}; \eta - \textsc{Nat}\}$

$\quad \mathsf{G}\ put_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}} \circ get_f \circ fzip_{\mathsf{F}}\ create_{(\!|in_{\mathsf{G}} \circ f|\!)_{\mathsf{F}}}$

$$\begin{aligned}
&\quad \circ\,(put_f \circ (id \times \mathsf{F}\ get_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}}) \vartriangle \pi_2) \circ (out_{\mathsf{G}} \times out_{\mathsf{F}}) \\
&\quad = \mathsf{G}\ put_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}} \circ get_f \circ fzip_{\mathsf{F}}\ create_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}} \\
&\quad \circ\,(put_f \circ (id \times \mathsf{F}\ get_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}}) \vartriangle \pi_2) \circ (out_{\mathsf{G}} \times out_{\mathsf{F}}) \\
&\llbracket get_f \circ fzip_{\mathsf{F}}\ create_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}} \circ (put_f \circ (id \times \mathsf{F}\ get_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}}) \vartriangle \pi_2) \circ (out_{\mathsf{G}} \times out_{\mathsf{F}})\rrbracket_{\mathsf{G}} \\
&= \pi_1 \\
&\Leftrightarrow \{\llbracket\cdot\rrbracket - \textsc{Uniq}; \eta - \textsc{Nat}\} \\
&get_f \circ \mathsf{F}\ \pi_1 \circ fzip_{\mathsf{F}}\ create_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}} \circ (put_f \circ (id \times \mathsf{F}\ get_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}}) \vartriangle \pi_2) \\
&\circ\,(out_{\mathsf{G}} \times out_{\mathsf{F}}) = out_{\mathsf{G}} \circ \pi_1 \\
&\Leftrightarrow \{fzip - \textsc{Cancel}; \times - \textsc{Cancel}\} \\
&get_f \circ put_f \circ (id \times \mathsf{F}\ get_{\llparenthesis in_{\mathsf{G}} \circ f \rrparenthesis_{\mathsf{F}}}) \circ (out_{\mathsf{G}} \times out_{\mathsf{F}}) = out_{\mathsf{G}} \circ \pi_1 \\
&\Leftrightarrow \{\textsc{PutGet}; \times - \textsc{Def}; id - \textsc{Nat}; \times - \textsc{Cancel}\} \\
&\pi_1 \circ (out_{\mathsf{G}} \times out_{\mathsf{F}}) = out_{\mathsf{G}} \circ \pi_1 \\
&\Leftrightarrow \{\times - \textsc{Def}; id - \textsc{Nat}; \times - \textsc{Cancel}\} \\
&out_{\mathsf{G}} \circ \pi_1 = out_{\mathsf{G}} \circ \pi_1
\end{aligned}$$

□

Proof ($\llbracket f \circ out_{\mathsf{F}} \rrbracket_{\mathsf{G}}$ *is a well-behaved lens* $\Leftarrow$ $get_f : \mathsf{F} \stackrel{.}{\to} \mathsf{G}$).

To write this proof, we only need to prove that the forward anamorphism $[\!(get_f \circ out_{\mathsf{F}})\!]_{\mathsf{G}}$ is recursive. Given that get_f is a natural transformation and out_{F} is a recursive F-coalgebra, then the G-coalgebra $get_f \circ out_{\mathsf{F}}$ is recursive (Capretta et al., 2006, Proposition 10.(b)).

□

B.3 Horizontal Delta Lens Proofs

B.3.1 Horizontal Delta Lens

Proof (Theorem 8).

$$\begin{aligned}
&get_{\blacktriangle}\ (put_{\blacktriangle}\ d) \\
&= \{\text{definition of } get_{\blacktriangle}\} \\
&get_{\blacktriangle}\ ((get_{\vartriangle} \circ d \triangledown id) \circ put_{\vartriangle}\ d) \\
&= \{\text{definition of } put_{\blacktriangle}\} \\
&get^{\circ}_{\vartriangle} \circ (get_{\vartriangle} \circ d \triangledown id) \circ put_{\vartriangle}\ d \circ get_{\vartriangle} \\
&= \{\textsc{PutGet}_{\vartriangle}; +\text{-}\textsc{Cancel}\}
\end{aligned}$$

$get^{\circ}_{\Delta} \circ get_{\Delta} \circ d$

$= \{ R \text{ entire and injective} \Leftrightarrow R^{\circ} \circ R = id \}$

d

$put_{\blacktriangle}\ id$

$= \{ \text{definition of } put_{\blacktriangle} \}$

$(get_{\Delta} \circ id \triangledown id) \circ put_{\Delta}\ id$

$= \{ id\text{-}\textsc{Nat}; \textsc{GetPut}_{\Delta} \}$

id

□

B.3.2 Composition

Proof ($f \circ g$ is a well-behaved horizontal delta lens).

$get_{f \circ g}\ (create_{f \circ g}\ v)$

$= \{ \text{definition of } get; \text{definition of } create \}$

$get_f\ (get_g\ (create_g\ (create_f\ v)))$

$= \{ \textsc{CreateGet} \}$

v

$create_{\Delta f \circ g} \circ get_{\Delta f \circ g}$

$= \{ \text{definition of } create_{\Delta}; \text{definition of } get_{\Delta} \}$

$create_{\Delta f} \circ create_{\Delta g} \circ get_{\Delta g} \circ get_{\Delta f}$

$= \{ \textsc{CreateGet}_{\Delta} \}$

id

$get_{f \circ g}\ (put_{f \circ g}\ (v, s)\ d)$

$= \{ \text{definition of } get; \text{definition of } put \}$

$get_f\ (get_g\ (put_g\ (put_f\ (v, get_g\ s)\ d, s)\ ((get_{\Delta f} \circ d \triangledown id) \circ put_{\Delta f}\ d)))$

$= \{ \textsc{PutGet} \}$

$get_f\ (put_f\ (v, get_g\ s)\ d)$

$= \{ \textsc{PutGet} \}$

v

$put_{\Delta_{f \circ g}}\ d \circ get_{\Delta_{f \circ g}}$

$= \{$definition of put_{Δ}; definition of $get_{\Delta}\}$

$((id + get_{\Delta_g}) \circ put_{\Delta_f}\ d \triangledown i_2) \circ put_{\Delta_g}\ ((get_{\Delta_f} \circ d \triangledown id) \circ put_{\Delta_f}\ d) \circ get_{\Delta_g} \circ get_{\Delta_f}$

$= \{\textsc{PutGet}_{\triangle}\}$

$((id + get_{\Delta_g}) \circ put_{\Delta_f}\ d \triangledown i_2) \circ i_1 \circ get_{\Delta_f}$

$= \{+\text{-}\textsc{Cancel}; \textsc{PutGet}_{\triangle}\}$

$(id + get_{\Delta_g}) \circ i_1$

$= \{+\text{-}\textsc{Def}; +\text{-}\textsc{Cancel}\}$

i_1

$put_{f \circ g}\ (get_{f \circ g}\ s, s)\ id$

$= \{$definition of put; definition of $get\}$

$put_g\ (put_f\ (get_f\ (get_g\ s), get_g\ s)\ id, s)\ ((get_{\Delta_f} \circ id \triangledown id) \circ put_{\Delta_f}\ id)$

$= \{id\text{-}\textsc{Nat}; \textsc{GetPut}_{\triangle}\}$

$put_g\ (put_f\ (get_f\ (get_g\ s), get_g\ s)\ id, s)\ id$

$= \{\textsc{GetPut}\}$

$put_g\ (get_g\ s, s)\ id$

$= \{\textsc{GetPut}\}$

s

$(get_{\Delta_{f \circ g}} \triangledown id) \circ put_{\Delta_{f \circ g}}\ id$

$= \{$definition of get_{Δ}; definition of $put_{\Delta}\}$

$(get_{\Delta_g} \circ get_{\Delta_f} \triangledown id) \circ ((id + get_{\Delta_g}) \circ put_{\Delta_f}\ id \triangledown i_2)$

$\circ\, put_{\Delta_g}\ ((get_{\Delta_f} \circ id \triangledown id) \circ put_{\Delta_f}\ id)$

$= \{id\text{-}\textsc{Nat}; \textsc{GetPut}_{\triangle}\}$

$(get_{\Delta_g} \circ get_{\Delta_f} \triangledown id) \circ ((id + get_{\Delta_g}) \circ put_{\Delta_f}\ id \triangledown i_2) \circ put_{\Delta_g}\ id$

$= \{+\text{-}\textsc{Fusion}; +\text{-}\textsc{Absor}; +\text{-}\textsc{Cancel}\}$

$((get_{\Delta_g} \circ get_{\Delta_f} \triangledown get_{\Delta_g}) \circ put_{\Delta_f}\ id \triangledown id) \circ put_{\Delta_g}\ id$

$= \{id\text{-}\textsc{Nat}; +\text{-}\textsc{Fusion}\}$

$(get_{\Delta_g} \circ (get_{\Delta_f} \triangledown id) \circ put_{\Delta_f}\ id \triangledown id) \circ put_{\Delta_g}\ id$

$= \{\textsc{GetPut}_{\triangle}; id\text{-}\textsc{Nat}\}$

$(get_{\Delta_g} \triangledown id) \circ put_{\Delta_g}\ id$

$= \{\textsc{GetPut}_{\triangle}\}$

id

□

B.3.3 Mapping

Proof (T f is a well-behaved horizontal delta lens).

$$\begin{aligned}
& get_{T\ f}\ (create_{T\ f}\ v) \\
& = \{\text{definition of } get; \text{definition of } create\} \\
& T\ get_f\ (T\ create_f\ v) \\
& = \{\textsc{Functor-Comp}; \textsc{CreateGet}; \textsc{Functor-Id}\} \\
& v
\end{aligned}$$

$$\begin{aligned}
& create_{\Delta\ T\ f} \circ get_{\Delta\ T\ f} \\
& = \{\text{definition of } create_\Delta; \text{definition of } get_\Delta\} \\
& id \circ id \\
& = \{id - \textsc{Nat}\} \\
& id
\end{aligned}$$

$$\begin{aligned}
& get_{T\ f}\ (put_{T\ f}\ (v, s)\ d) \\
& = \{\text{definition of } get; \text{definition of } put\} \\
& T\ get_f\ (recover\ (shape\ v, dput\ \cup\ dcreate)) \\
& = \{recover\text{-}\textsc{Data}\} \\
& recover\ (shape\ v, get_f \circ (dput\ \cup\ dcreate)) \\
& = \{\cup\text{-}\textsc{Fusion}\} \\
& \quad get_f \circ dput \\
& \quad = \{\text{definition of } dput\} \\
& \quad get_f \circ put_f \circ (data\ v \triangle data\ s \circ d) \\
& \quad = \{\textsc{PutGet}; \times\text{-}\textsc{Cancel}\} \\
& \quad data\ v \\
& \quad get_f \circ dcreate \\
& \quad = \{\text{definition of } dcreate\} \\
& \quad get_f \circ create_f \circ data\ v \circ (id - \delta dput) \\
& \quad = \{\textsc{CreateGet}\} \\
& \quad data\ v \circ (id - \delta dput) \\
& recover\ (shape\ v, data\ v\ \cup\ data\ v \circ (id - \delta dput)) \\
& = \{get_f \text{ total} \Rightarrow \delta dput = \delta(get_f \circ dput)\} \\
& recover\ (shape\ v, data\ v\ \cup\ data\ v \circ (id - \delta(data\ v)))
\end{aligned}$$

$= \{ R \circ (id - \delta R) = \bot ; \bot\text{-}\textsc{Neutral} \}$

$recover\ (shape\ v, data\ v)$

$= \{ recover\text{-}\textsc{Iso} \}$

v

$put_{\Delta\ T\ f}\ d \circ get_{\Delta\ T\ f}$

$= \{ \text{definition of } get_{\Delta}; \text{definition of } put_{\Delta} \}$

$i_1 \circ id$

$= \{ id - \textsc{Nat} \}$

i_1

$put_{T\ f}\ (get_{T\ f}\ s, s)\ id$

$= \{ \text{definition of } put; \text{definition of } get \}$

$\quad dput$

$\quad = \{ \text{definition of } dput; \text{definition of } get \}$

$\quad put_f \circ (data\ ((T\ get_f)\ s) \triangle data\ s \circ id)$

$\quad = \{ id - \textsc{Nat}; data\text{-}\textsc{Data} \}$

$\quad put_f \circ (get_f \circ data\ s \triangle data\ s)$

$\quad = \{ \times\text{-}\textsc{Fusion} \circ \textsc{GetPut} \}$

$\quad data\ s$

$\quad dcreate$

$\quad = \{ \text{definition of } dcreate \}$

$\quad create_f \circ data\ (get\ s) \circ (id - \delta dput)$

$\quad = \{ dput \text{ total} \Rightarrow id - \delta dput = \bot \}$

$\quad create_f \circ data\ (get\ s) \circ \bot$

$\quad = \{ \bot\text{-}\textsc{Fusion} \}$

$\quad \bot$

$recover\ (shape\ ((T\ get_f)\ s), data\ s\ \cup\ \bot)$

$= \{ \bot\text{-}\textsc{Neutral} \}$

$recover\ (shape\ ((T\ get_f)\ s), data\ s)$

$= \{ shape\text{-}\textsc{Data}; recover\text{-}\textsc{Iso} \}$

s

$(get_{\Delta\ T\ f} \nabla id) \circ put_{\Delta\ T\ f}\ id$

$= \{ \text{definition of } get_{\Delta}; \text{definition of } put_{\Delta} \}$

$(id \triangledown id) \circ i_1$

$= \{+ - \textsc{Cancel}\}$

id

□

B.3.4 Reshaping

Proof ($\overleftrightarrow{\eta}$ is a well-behaved horizontal delta lens).

$get_{\overleftrightarrow{\eta}}\ (create_{\overleftrightarrow{\eta}}\ v)$

$= \{\text{definition of } get; \text{definition of } create\}$

$get_\eta\ (create_\eta\ v)$

$= \{\textsc{CreateGet}\}$

v

$create_{\Delta \overleftrightarrow{\eta}}\ \{v\} \circ get_{\Delta \overleftrightarrow{\eta}}\ \{create_{\overleftrightarrow{\eta}}\ v\}$

$= \{\text{definition of } create_\Delta; \text{definition of } get_\Delta; \text{definition of } get\}$

$\overleftarrow{create_\eta}\ v \circ \overleftarrow{get_\eta}\ (create_{\overleftrightarrow{\eta}}\ v)$

$= \{\overleftarrow{\cdot}\ \textsc{-Comp}\}$

$\overleftarrow{get_\eta \circ create_\eta}\ v$

$= \{\textsc{CreateGet}; \overleftarrow{\cdot}\ \textsc{-Id}\}$

id

$get_{\overleftrightarrow{\eta}}\ (put_{\overleftrightarrow{\eta}}\ (v, s)\ d)$

$= \{\text{definition of } get; \text{definition of } put\}$

$get_\eta\ (put_\eta\ (v, s))$

$= \{\textsc{PutGet}\}$

v

$put_{\Delta \overleftrightarrow{\eta}}\ \{(v, s)\} \circ get_{\Delta \overleftrightarrow{\eta}}\ \{put_{\overleftrightarrow{\eta}}\ (v, s)\ d\}$

$= \{\text{definition of } put_\Delta; \text{definition of } get_\Delta; \text{definition of } put\}$

$\overleftarrow{put_\eta}\ (v, s) \circ \overleftarrow{get_\eta}\ (put_\eta\ (v, s))$

$= \{\overleftarrow{\cdot}\ \textsc{-Comp}\}$

$\overleftarrow{get_\eta \circ put_\eta}\ (v, s)$

$= \{\textsc{PutGet}\}$

$\overleftarrow{\pi_1}\ (v, s)$

$= \{\overleftarrow{\cdot}\text{-}\textsc{Fst}\}$

i_1

$put_{\overleftrightarrow{\eta}}\ (get_{\overleftrightarrow{\eta}}\ s, s)\ id$

$= \{\text{definition of } put; \text{definition of } get\}$

$put_{\eta}\ (get_{\eta}\ s, s)$

$= \{\textsc{GetPut}\}$

s

$(get_{\Delta\overleftrightarrow{\eta}}\ \{s\} \triangledown id) \circ put_{\Delta\overleftrightarrow{\eta}}\ \{(get_{\overleftrightarrow{\eta}}\ s, s)\}\ id = id$

$= \{\text{definition of } get_{\Delta}; \text{definition of } put_{\Delta}\}$

$(\overleftarrow{get_{\eta}}\ s \triangledown id) \circ \overleftarrow{put_{\eta}}\ (get_{\overleftrightarrow{\eta}}\ s, s)$

$= \{\overleftarrow{\cdot}\text{-}\textsc{Id}; \overleftarrow{\cdot}\text{-}\textsc{Split}\}$

$\overleftarrow{(get_{\eta} \vartriangle id)}\ s \circ \overleftarrow{put_{\eta}}\ (get_{\overleftrightarrow{\eta}}\ s, s)$

$= \{\overleftarrow{\cdot}\text{-}\textsc{Comp};\}$

$\overleftarrow{put_{\eta} \circ (get_{\eta} \vartriangle id)}\ s$

$= \{\textsc{GetPut}; \overleftarrow{\cdot}\text{-}\textsc{Id}\}$

id

□

B.3.5 Catamorphisms

Proof ($\llparenthesis f \rrparenthesis_{\mathcal{F}}$ *is a well-behaved horizontal delta lens* $\Leftarrow \begin{array}{l} \llparenthesis f \rrparenthesis_{\mathcal{F}}^{pos} \textit{ is well-behaved} \\ reduce_{\mathcal{F}} \circ out_{\mathcal{F}} \textit{ is well-founded} \end{array}$ *).*

To prove that the catamorphism horizontal delta lens is well-behaved, we must first prove that the respective backward transformations are terminating functions. For $create_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}$, this is a direct consequence of the positional $\llparenthesis f \rrparenthesis_{\mathcal{F}}^{pos}$ being well-behaved. Ignoring the deltas, we can redefine $put_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}$ as the following anamorphism (the $ifshrink$ and $ifgrow$ predicates represent our delta-based tests to identify view insertions and deletions):

$$\llbracket (grow_{coalg} \triangledown (shrink_{coalg} \triangledown pos_{coalg}) \circ ifshrink?) \circ ifgrow? \rrbracket_{\mathcal{I}d \boxplus \mathcal{F}}$$

$$grow_{coalg} \quad = i_2 \circ \sigma_{\mathcal{F}} \circ (create_f \times id)$$

$$\begin{aligned}
\mathit{shrink}_{\mathit{coalg}} &= i_1 \circ (\mathit{id} \times \mathit{reduce}_{\mathcal{F}} \circ \mathit{out}_{\mathcal{F}}) \\
\mathit{pos}_{\mathit{coalg}} &= i_2 \circ \mathit{fzip}_{\mathcal{F}}\ \mathit{create}_{\llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ (\mathit{put}_f \circ (\mathit{id} \times \mathcal{F}\ \mathit{get}_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}) \bigtriangleup \pi_2) \circ (\mathit{id} \times \mathit{out}_{\mathcal{F}})
\end{aligned}$$

Proving that this anamorphism is recursive amounts to showing that the accessibility relations for the three cases are well-founded. If each of the cases produces a strictly smaller view-source pair such that view or the source decreases, then if follows that the combined accessibility relation is also well-founded.

For insertions ($\mathit{grow}_{\mathit{coalg}}$), the corresponding coalgebra always consumes the view and the proof that its accessibility relation is well-founded can be done as follows:

$$\begin{aligned}
&\in_{\mathcal{I}d \boxplus \mathcal{F}} \circ i_2 \circ \sigma_{\mathcal{F}} \circ (\mathit{create}_f \times \mathit{id})\ \text{well} - \text{founded} \\
&\Leftrightarrow \{ \in - \textsc{Def}; +\text{-}\textsc{Cancel} \} \\
&\in_{\mathcal{F}} \circ \sigma_{\mathcal{F}} \circ (\mathit{create}_f \times \mathit{id})\ \text{well} - \text{founded} \\
&\Leftrightarrow \{ \in\text{-}\textsc{Strength}; \times\text{-}\textsc{Functor-Comp} \} \\
&(\in_{\mathcal{F}} \circ \mathit{create}_f \times \mathit{id})\ \text{well} - \text{founded} \\
&\Leftrightarrow \{ \mathit{create}_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ \text{terminates} \Leftrightarrow \in_{\mathcal{F}} \circ \mathit{create}_f\ \text{well} - \text{founded} \} \\
&\textsc{True}
\end{aligned}$$

For deletions ($\mathit{shrink}_{\mathit{coalg}}$), the intuition is that $\mathit{reduce}_{\mathcal{F}}$ always produces a smaller source. However, since this is not guaranteed by the monoid properties, we require that the corresponding accessibility relation $\mathit{reduce}_{\mathcal{F}} \circ \mathit{out}_{\mathcal{F}}$ is well-founded.

For the positional case ($\mathit{pos}_{\mathit{coalg}}$), the coalgebra may either consume the view or the source or both. The proof that its accessibility relation is well-founded comes directly from the fact that the positional catamorphism horizontal delta lens $\llparenthesis f \rrparenthesis_{\mathcal{F}}^{\mathit{pos}}$ is well-behaved.

After proving that $\mathit{put}_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}$ terminates, we can write the proofs of the horizontal delta lens laws using standard structural induction:

$$\begin{aligned}
&\mathit{get}_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (\mathit{create}_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ v) \\
&= \{ \text{definition of } \mathit{get}; \text{definition of } \mathit{create} \} \\
&\mathit{get}_{\llparenthesis f \rrparenthesis_{\mathcal{F}}^{\mathit{pos}}}\ (\mathit{create}_{\llparenthesis f \rrparenthesis_{\mathcal{F}}^{\mathit{pos}}}\ v) \\
&= \{ \llparenthesis f \rrparenthesis_{\mathcal{F}}^{\mathit{pos}}\ \text{well} - \text{behaved}; \textsc{CreateGet} \} \\
&v
\end{aligned}$$

$$\begin{aligned}
&\mathit{create}_{\bigtriangleup \llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ \mathit{get}_{\bigtriangleup \llparenthesis f \rrparenthesis_{\mathcal{F}}} \\
&= \{ \text{definition of } \mathit{create}_{\bigtriangleup}; \text{definition of } \mathit{get}_{\bigtriangleup} \} \\
&\mathit{create}_{\bigtriangleup \llparenthesis f \rrparenthesis_{\mathcal{F}}^{\mathit{pos}}} \circ \mathit{get}_{\bigtriangleup \llparenthesis f \rrparenthesis_{\mathcal{F}}^{\mathit{pos}}} \\
&= \{ \llparenthesis f \rrparenthesis_{\mathcal{F}}^{\mathit{pos}}\ \text{well} - \text{behaved}; \textsc{CreateGet}_{\bigtriangleup} \} \\
&\mathit{id}
\end{aligned}$$

$get_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (put_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (v,s)\ d) = v$

$\Leftrightarrow \{\text{definition of } put; \text{expand cases}\}$

$get_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (grow\ (v,s)\ d) = v$

$\Leftrightarrow \{\text{definition of } get; \text{definition of } grow\}$

$get_f\ (\mathcal{F}\ get_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (out_{\mathcal{F}}\ (in_{\mathcal{F}}\ (\sigma put_{\mathcal{F}}\ (create_f\ v,s)\ (d \circ create_{\Delta f}))))) = v$

$\Leftrightarrow \{in - out - \textsc{Iso}; \text{inductive step}; \mathcal{F}\ get_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (\sigma put_{\mathcal{F}}\ (v,s)\ d) = v\}$

$get_f\ (create_f\ v) = v$

$\Leftrightarrow \{\textsc{CreateGet}\}$

$\textsc{True}$

$get_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (shrink\ (v,s)\ d) = v$

$\Leftrightarrow \{\text{definition of } shrink; d_V = get^{\circ}_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ reduce_{\mathcal{F}}{}^{\circ}_{\Delta} \circ get_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ d\}$

$get_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (put_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (v, reduce_{\mathcal{F}}\ (out_{\mathcal{F}}\ s))\ d_V) = v$

$\Leftrightarrow \{\text{inductive step}\}$

$\textsc{True}$

$get_{\llparenthesis f \rrparenthesis_{\mathcal{F}}}\ (put_{f \circ \mathcal{F}\ \llparenthesis f \rrparenthesis_{\mathcal{F}} \circ out_{\mathcal{F}}}\ (v,s)\ d) = v$

$\Leftrightarrow \{\text{definition of } get\}$

$get_{f \circ \mathcal{F}\ \llparenthesis f \rrparenthesis_{\mathcal{F}} \circ out_{\mathcal{F}}}\ (put_{f \circ \mathcal{F}\ \llparenthesis f \rrparenthesis_{\mathcal{F}} \circ out_{\mathcal{F}}}\ (v,s)\ d) = v$

$\{\ldots; \text{inductive step}; \textsc{PutGet}\}$

$\textsc{True}$

$\textsc{True}$

$put_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}}\ d \circ get_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}} = i_1$

$\Leftrightarrow \{\text{definition of } put_{\Delta}; \text{expand cases}\}$

$grow_{\Delta}\ d \circ get_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}} = i_1$

$\Leftrightarrow \{\text{definition of } grow_{\Delta}; \text{definition of } get_{\Delta}\}$

$(create_{\Delta f} + id) \circ \sigma put_{\Delta \mathcal{F}}\ (d \circ create_{\Delta f}) \circ id \circ get_{\Delta \mathcal{F}\ \llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ get_{\Delta f} = i_1$

$\Leftrightarrow \{id\text{-}\textsc{Nat}; \text{inductive step}; \sigma put_{\Delta \mathcal{F}}\ d \circ get_{\Delta \mathcal{F}\ \llparenthesis f \rrparenthesis_{\mathcal{F}}} = i_1\}$

$(create_{\Delta f} + id) \circ i_1 \circ get_{\Delta f} = i_1$

$\Leftrightarrow \{+\text{-}\textsc{Def}; +\text{-}\textsc{Cancel}\}$

$create_{\Delta f} \circ get_{\Delta f} = i_1$

$\Leftrightarrow \{\textsc{CreateGet}_{\Delta}\}$

$\textsc{True}$

$shrink_{\Delta}\ d \circ get_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}} = i_1$

$\Leftrightarrow \{\text{definition of } shrink_{\Delta}\}$

$(id + id) \circ (id + reduce_{\Delta \mathcal{F}}) \circ put_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}} \circ get_{\Delta \llparenthesis f \rrparenthesis_{\mathcal{F}}} = i_1$

$\Leftrightarrow \{+ - \text{REFLEX}; \text{inductive step}\}$

$(id + reduce_{\Delta\mathcal{F}}) \circ i_1$

$\Leftrightarrow \{+\text{-CANCEL}\}$

i_1

$put_{\Delta f \circ \mathcal{F}\ (\!|f|\!)_{\mathcal{F}} \circ out_{\mathcal{F}}} \circ get_{\Delta (\!|f|\!)_{\mathcal{F}}} = i_1$

$\Leftrightarrow \{\text{definition of } get_{\Delta}\}$

$put_{\Delta f \circ \mathcal{F}\ (\!|f|\!)_{\mathcal{F}} \circ out_{\mathcal{F}}} \circ get_{\Delta f \circ \mathcal{F}\ (\!|f|\!)_{\mathcal{F}} \circ out_{\mathcal{F}}} = i_1$

$\Leftrightarrow \{\ldots; \text{inductive step}; \text{PUTGET}_{\Delta}\}$

i_1

TRUE

$put_{(\!|f|\!)_{\mathcal{F}}}\ (get_{(\!|f|\!)_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\text{definition of } put; \text{expand cases}\}$

$\rho V \neq \bot \wedge (\rho V \cap \rho id) = \bot \Rightarrow grow\ (get_{(\!|f|\!)_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\rho id = id; \Phi\text{-COMP}; id\text{-NAT}\}$

$\rho V \neq \bot \wedge \rho V = \bot \Rightarrow grow\ (get_{(\!|f|\!)_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\text{contradiction}\}$

$\text{FALSE} \Rightarrow grow\ (get_{(\!|f|\!)_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\text{logic implication}\}$

TRUE

$\rho S \neq \bot \wedge (\rho S \cap \delta id) = \bot \Rightarrow shrink\ (get_{(\!|f|\!)_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\delta id = id; \Phi\text{-COMP}; id\text{-NAT}\}$

$\rho S \neq \bot \wedge \rho S = \bot \Rightarrow shrink\ (get_{(\!|f|\!)_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\text{contradiction}\}$

$\text{FALSE} \Rightarrow shrink\ (get_{(\!|f|\!)_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\text{logic implication}\}$

TRUE

$\text{otherwise} \Rightarrow put_{f \circ \mathcal{F}\ (\!|f|\!)_{\mathcal{F}} \circ out_{\mathcal{F}}}\ (get_{(\!|f|\!)_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\text{definition of } get\}$

$\text{otherwise} \Rightarrow put_{f \circ \mathcal{F}\ (\!|f|\!)_{\mathcal{F}} \circ out_{\mathcal{F}}}\ (get_{f \circ \mathcal{F}\ (\!|f|\!)_{\mathcal{F}} \circ out_{\mathcal{F}}}\ s, s)\ id = s$

$\Leftrightarrow \{\ldots; \text{inductive step}; \text{GETPUT}\}$

$\text{otherwise} \Rightarrow \text{TRUE}$

$\Leftrightarrow \{\text{logic implication}\}$

TRUE

TRUE

$(get_{\Delta (\!|f|\!)_{\mathcal{F}}} \triangledown id) \circ put_{\Delta (\!|f|\!)_{\mathcal{F}}}\ id = id$

$\Leftrightarrow \{$definition of put_{Δ}; expand cases$\}$

$\rho V \neq \bot \wedge (\rho V \cap \rho id) = \bot \Rightarrow (get_{\Delta (\!|f|\!)_{\mathcal{F}}} \triangledown id) \circ grow_{\Delta}\ id = id$

$\Leftrightarrow \{\rho id = id;$ Φ-Comp; id-Nat; contradiction$\}$

$\textsc{False} \Rightarrow (get_{\Delta (\!|f|\!)_{\mathcal{F}}} \triangledown id) \circ grow_{\Delta}\ id = id$

$\Leftrightarrow \{$logic implication$\}$

$\textsc{True}$

$\rho S \neq \bot \wedge (\rho S \cap \delta id) = \bot \Rightarrow (get_{\Delta (\!|f|\!)_{\mathcal{F}}} \triangledown id) \circ shrink_{\Delta}\ id = id$

$\Leftrightarrow \{\delta id = id;$ Φ-Comp; id-Nat; contradiction$\}$

$\textsc{False} \Rightarrow (get_{\Delta (\!|f|\!)_{\mathcal{F}}} \triangledown id) \circ shrink_{\Delta}\ id = id$

$\Leftrightarrow \{$logic implication$\}$

$\textsc{True}$

otherwise $\Rightarrow (get_{\Delta (\!|f|\!)_{\mathcal{F}}} \triangledown id) \circ put_{\Delta f \circ \mathcal{F}\ (\!|f|\!)_{\mathcal{F}} \circ out_{\mathcal{F}}}\ id = id$

$\Leftrightarrow \{$definition of get_{Δ}; ...; inductive step; GetPut$_{\Delta}\}$

otherwise $\Rightarrow \textsc{True}$

$\Leftrightarrow \{$logic implication$\}$

$\textsc{True}$

$\textsc{True}$

□

Bibliography

M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

M. Antkiewicz and K. Czarnecki. Design space of heterogeneous synchronization. In *Proceedings of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *Lecture Notes in Computer Science*, pages 3–46. Springer-Verlag, 2008.

F. Atanassow and J. Jeuring. Customizing an XML-Haskell data binding with type isomorphism inference in Generic Haskell. *Science of Computer Programming*, 65 (2):72–107, 2007.

F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

R. Backhouse and H. Doornbos. Mathematics of recursive program construction. Manuscript available at `http://www.cs.nott.ac.uk/rcb/MPC/papers`, 2001.

J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

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

D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 193–204. ACM, 2010.

L. Barbosa. *Components as Coalgebras*. PhD thesis, University of Minho, 2001.

L. S. Barbosa and J. N. Oliveira. Transposing partial components: an exercise on coalgebraic refinement. *Theoretical Computer Science*, 365(1):2–22, 2006.

P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled schema transformation and data: Conversion for XML and SQL. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages (PADL 2007)*, volume 4354 of *Lecture Notes in Computer Science*, pages 290–304. Springer-Verlag, 2007.

R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the 25th ACM SIGMOD Symposium on Principles of Database Systems (PODS 2006)*, pages 338–347. ACM, 2006.

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2008)*, pages 407–419. ACM, 2008.

C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4-5):385–406, 2008.

P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2002)*, pages 150–158. ACM, 2002.

V. Capretta, T. Uustalu, and V. Vene. Recursive coalgebras from comonads. *Information and Computation*, 204(4):437–468, 2006.

M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 241–253. ACM, 2005.

A. Cicchetti, D. di Ruscio, R. Eramo, and A. Pierantonio. JTL: A bidirectional and change propagating transformation language. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *Lecture Notes in Computer Science*, pages 183–202. Springer-Verlag, 2011.

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.

J. Clark. XSL Transformations (XSLT) Version 1.0. World Wide Web Consortium (W3C) Recommendation. http://www.w3.org/TR/xslt, November 1999.

A. Cunha. *Point-free Program Calculation*. PhD thesis, University of Minho, 2005.

A. Cunha and H. Pacheco. Algebraic specialization of generic functions for recursive types. *Electronic Notes in Theoretical Computer Science*, 229(5):57–74, 2011.

A. Cunha and J. S. Pinto. Point-free program transformation. *Fundamenta Informaticae*, 66(4):315–352, 2005.

A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. *Electronic Notes in Theoretical Computer Science*, 174(1):17–34, 2007.

A. Cunha and J. Visser. Transformation of structure-shy programs with application to XPath queries and strategic functions. *Science of Computer Programming*, 76(6): 512–539, 2011.

A. Cunha, J. N. Oliveira, and J. Visser. Type-safe two-level data transformation. In *Proceedings of the 14th International Symposium on Formal Methods (FM 2006)*, volume 4085 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2006a.

A. Cunha, J. S. Pinto, and J. Proença. A framework for point-free program transformation. In *Selected Papers of the 17th International Workshop on Implementation and Application of Functional Languages (IFL 2005)*, volume 4015 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2006b.

J. Cunha, J. Saraiva, and J. Visser. From spreadsheets to relational databases and back. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2009)*, pages 179–188. ACM, 2009.

J. Cunha, J. P. Fernandes, J. Mendes., H. Pacheco, and J. Saraiva. Bidirectional transformation of model-driven spreadsheets. In *Proceedings of the 5th International Conference on Model Transformation (ICMT 2012)*, volume 7307 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, 2012.

K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the 2nd International Conference on Model Transformation (ICMT 2009)*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer-Verlag, 2009.

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

Z. Diskin. Algebraic models for bidirectional model synchronization. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, 2008a.

Z. Diskin. Algebra of bidirectional model synchronization. Technical Report CSRG-573, Department of Computing Science, University of Toronto, 2008b.

Z. Diskin. Model synchronization: Mappings, tiles, and categories. In *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009)*, volume 6491 of *Lecture Notes in Computer Science*, pages 92–165. Springer-Verlag, 2011.

Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *Journal of Object Technology*, 10:6:1–25, 2011a.

Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2011)*, volume 6981 of *Lecture Notes in Computer Science*, pages 304–318. Springer-Verlag, 2011b.

H. Doornbos and B. Von Karger. On the union of well-founded relations. *Logic Journal of the IGPL*, 6(2):195–201, 1998.

H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2006.

H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *Lecture Notes in Computer Science*, pages 72–86. Springer-Verlag, 2007.

R. Ennals and D. Gay. Multi-language synchronization. In *Proceedings of the 16th European Conference on Programming Languages and Systems (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 475–489. Springer-Verlag, 2007.

L. Fegaras. Propagating updates through XML views using lineage tracing. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE 2010)*, pages 309 –320. IEEE, 2010.

F. Ferreira and H. Pacheco. XPTO – an Xpath preprocessor with type-aware optimization. In *Proceedings of the Conference on Compilers, Related Technologies and Applications (CORTA 2007)*. University of Beira Interior, 2007.

M. M. Fokkinga. Tupling and Mutumorphisms. *Squiggolist*, 1(4), 1989.

J. N. Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, 2009.

J. N. Foster, M. B. Greenwald, C. Kirkegaard, B. C. Pierce, and A. Schmitt. Schema-directed data synchronization. Technical Report MS-CIS-05-02, University of Pennsylvania, 2005.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)*, pages 383–396. ACM, 2008.

J. N. Foster, B. C. Pierce, and S. Zdancewic. Updatable security views. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF-22)*, pages 60–74. IEEE, 2009.

A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4): 1–64, 2008.

J. Gibbons. Calculating functional programs. In *Revised Lectures from the International Summer School and Workshop on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 151–203. Springer-Verlag, 2002.

J. Gibbons. Datatype-generic programming. In *Proceedings of the 2006 International Spring School on Datatype-generic Programming (SSDGP 2006)*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer-Verlag, 2007.

H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *Lecture Notes in Computer Science*, pages 543–557. Springer-Verlag, 2006.

G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Transformation-based database reverse engineering. In *Proceedings of the 12th International Conference on the Entity-Relationship Approach (ER 1993)*, volume 823 of *Lecture Notes in Computer Science*, pages 364–375. Springer-Verlag, 1994.

F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, and Y. Xiong. Correctness of model synchronization based on triple graph grammars. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2011)*, volume 6981 of *Lecture Notes in Computer Science*, pages 668–682. Springer-Verlag, 2011.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 205–216. ACM, 2010.

R. Hinze. *Generic programs and proofs*. Habilitation thesis, Bonn University, 2000.

M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 371–384. ACM, 2011.

M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *Proceedings of the 39th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2012)*, pages 495–508. ACM, 2012.

P. Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Technische Universiteit Eindhoven, 1997.

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher Order and Symbolic Computation*, 21(1-2):89–118, 2008.

K. Inaba, S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Graph-transformation verification using monadic second-order logic. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP 2011)*, pages 17–28. ACM, 2011.

P. Jansson and J. Jeuring. PolyP – a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*, pages 470–482. ACM, 1997.

C. Jay. A semantics for shape. *Science of Computer Programming*, 25(2-3):251–283, 1995.

P. Johann and N. Ghani. Initial algebra semantics is enough! In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications (TLCA 2007)*, volume 4583 of *Lecture Notes in Computer Science*, pages 207–222. Springer-Verlag, 2007.

S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 203–233. ACM, 2001.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional programming (ICFP 2006)*, pages 50–61. ACM, 2006.

F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer-Verlag, 2006.

S. Kawanaka and H. Hosoya. biXid: a bidirectional transformation language for XML. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*, pages 201–214. ACM, 2006.

M. Kay. XSL Transformations (XSLT) Version 2.0. World Wide Web Consortium (W3C) Recommendation. http://www.w3.org/TR/xslt20, January 2007.

A. Keller. Choosing a view update translator by dialog at view definition time. In *Proceedings of the 12th International Conference on Very Large Databases (VLDB 86)*, pages 467–474. Morgan Kaufmann Publishers, 1986.

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

A. Königs and A. Schürr. Tool integration with triple graph grammars – a survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113–150, 2006.

R. Lämmel. Grammar adaptation. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME 2001)*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer-Verlag, 2001.

R. Lämmel. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 54(1-2):1–64, 2003.

R. Lämmel. Coupled software transformations (extended abstract). Proceedings of the 1st International Workshop on Software Evolution Transformations (SET 2004), 2004a.

R. Lämmel. Transformations everywhere. *Science of Computer Programming*, 52(1-3): 1–8, 2004b.

R. Lämmel and W. Lohmann. Format evolution. In *Proceedings of the 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

R. Lämmel and E. Meijer. Mappings make data processing go 'round. In *Proceedings of the 1st International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, volume 4143 of *Lecture Notes in Computer Science*, pages 169–218. Springer-Verlag, 2006.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI 2003*, pages 26–37. ACM, 2003.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM, 2005.

R. Lämmel and J. Visser. A Strafunski Application Letter. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, volume 2562 of *Lecture Notes in Computer Science*, pages 357–375. Springer-Verlag, 2003.

K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1995.

D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of XQuery. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM 2007)*, pages 21–30. ACM, 2007.

S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1998.

N. Macedo, H. Pacheco, and A. Cunha. Relations as executable specifications: taming partiality and non-determinism using invariants. In *Proceedings of the 13th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 13)*, Lecture Notes in Computer Science. Springer-Verlag, 2012. to appear.

D. MacQueen and M. Tofte. A semantics for higher-order functors. In Donald Sannella, editor, *Proceedings of the 5th European Symposium on Programming (ESOP 1994)*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 47–58. ACM, 2007.

L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

L. Meertens. Designing constraint maintainers for user interaction. Manuscript available at http://www.kestrel.edu/home/people/meertens, 1998.

E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA 1991)*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD 2007)*, pages 461–472. ACM, 2007.

N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Proceedings of the 2007 ACM SIGPLAN Haskell Workshop*, pages 49–60. ACM, 2007.

C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27 (6):481–503, 1990.

S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *Lecture Notes in Computer Science*, pages 2–20. Springer-Verlag, 2004.

K. Nakano, Z. Hu, and M. Takeichi. Consistent web site updating based on bidirectional transformation. *International Journal on Software Tools for Technology Transfer*, 11 (6):453–468, 2009.

U. Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP 2008)*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2009.

J. N. Oliveira. Calculate databases with ‘simplicity’. Presented at the IFIP WG 2.1 #59 Meeting, slides available at http://www3.di.uminho.pt/~jno/ps/ifip04sl.pdf, September 2004.

J. N. Oliveira. Transforming data by calculation. In *Proceedings of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *Lecture Notes in Computer Science*, pages 134–195. Springer-Verlag, 2008.

J. N. Oliveira. Extended static checking by calculation using the pointfree transform. In *Proceedings of the International Summer School on Language Engineering and Rigorous Software Development (LerNet 2008)*, volume 5520 of *Lecture Notes in Computer Science*, pages 195–251. Springer-Verlag, 2009.

OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Version 1.0. Associated OMG document ptc/07-07-08. http://www.omg.org/spec/QVT/1.0/, April 2008.

H. Pacheco and A. Cunha. Generic point-free lenses. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC 2010)*, volume 6120 of *Lecture Notes in Computer Science*, pages 331–352. Springer-Verlag, 2010.

H. Pacheco and A. Cunha. Calculating with lenses: optimising bidirectional transformations. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial evaluation and Program Manipulation (PEPM 2011)*, pages 91–100. ACM, 2011.

H. Pacheco and A. Cunha. Multifocal: A strategic bidirectional transformation language for XML schemas. In *Proceedings of the 5th International Conference on Model Transformation (ICMT 2012)*, volume 7307 of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, 2012.

H. Pacheco, A. Cunha, and Z. Hu. Delta lenses over inductive types. In *Proceeding of the 1st International Workshop on Bidirectional Transformations (BX 2012)*, Electronic Communications of the EASST, 2012a. to appear.

H. Pacheco, N. Macedo, A. Cunha, and J. Voigtländer. A taxonomy of bidirectional transformations. submitted, 2012b.

A. Pardo. Generic accumulations. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 49–78. Kluwer, B.V., 2003.

B. C. Pierce. *Basic Category Theory for Computer Scientists.* Foundations of Computing. MIT Press, 1991.

I. Sasano, Z. Hu, S. Hidaka, K. Inaba, H. Kato, and K. Nakano. Toward bidirectionalization of ATL with GRoundTram. In *Proceedings of the 4th International Conference on Model Transformation (ICMT 2011)*, volume 6707 of *Lecture Notes in Computer Science*, pages 138–151. Springer-Verlag, 2011.

T. Schrijvers, Sulzmann M., S. Peyton Jones, and M. T. Chakravarty. Towards open type functions for Haskell. In *Draft proceedings of the 19th International Symposium on Implementation and Application of Functional Languages (IFL 2007)*, pages 233–251. Computing Laboratory, University of Kent, 2007. Technical Report No. 12-07.

A. Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of the 21st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1995)*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer-Verlag, 1995.

A. Schürr and F. Klar. 15 years of triple graph grammars. In *Proceedings of the 4th International Conference on Graph Transformations (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer-Verlag, 2008.

D. Sereni. Termination analysis and call graph construction for higher-order functional programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 71–84. ACM, 2007.

T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture (FPCA 1993)*, pages 233–242. ACM, 1993.

G. Sittampalam and O. de Moor. Mechanising fusion. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, chapter 5, pages 79–104. Palgrave Macmillan, 2003.

P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2007.

P. Stevens. A landscape of bidirectional model transformations. In *Proceedings of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer-Verlag, 2008.

M. Takeichi. Configuring bidirectional programs with functions. In *Draft proceedings of the 21st Symposium on Implementation and Application of Functional Languages (IFL 2009)*. Seton Hall University, 2009. Technical Report SHU-TR-CS-2009-09-1.

J. F. Terwilliger, L. M. L. Delcambre, and J. Logan. Querying through a user interface. *Data & Knowledge Engineering*, 63(3):774–794, 2007.

W. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

D. Turner. Elementary strong functional programming. In *Proceedings of the 1st International Symposium on Functional Programming Languages in Education (FPLE 1995)*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1995.

A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

S. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *Lecture Notes in Computer Science*, pages 630–644. Springer-Verlag, 2008.

E. Visser. Stratego: A language for program transformation based on rewriting strategies system description of Stratego 0.5. In *Proceedings of the 12th International*

Conference on Rewriting Techniques and Applications (RTA 2001), volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.

E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

J. Visser. Coupled transformation of schemas, documents, queries, and constraints. *Electronic Notes in Theoretical Computer Science*, 200(3):3–23, 2008.

J. Voigtländer. Bidirectionalization for free! (Pearl). In *Proceedings of the 36th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2009)*, pages 165–176. ACM, 2009.

J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pages 181–192. ACM, 2010.

P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1987)*, pages 307–313. ACM, 1987.

M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Gradual refinement: blending pattern matching with data abstraction. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC 2010)*, volume 6120 of *Lecture Notes in Computer Science*, pages 397–425. Springer-Verlag, 2010.

M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, pages 392–403. ACM, 2011.

M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 133–142. Springer-Verlag, 2004.

Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 164–173. ACM, 2007.

Y. Xiong, H. Song, Z. Hu, and M. Takeichi. Supporting parallel updates with bidirectional model transformations. In *Proceedings of the 2nd International Conference on Model Transformations (ICMT 2009)*, volume 5563 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag, 2009.

T. Yokoyama, H. Bock Axelsen, and Glück R. Principles of a reversible programming language. In *Proceedings of the 5th International Conference on Computing Frontiers (CF 2008)*, pages 43–54. ACM, 2008.