Lenses and Bidirectional Programming in Curry

submitted by

Sandra Dylus

Master's Thesis

submitted in September 2014

Programming Languages and Compiler Construction
Department of Computer Science
Christian-Albrechts-Universität zu Kiel

Advised by Prof. Dr. Michael Hanus
M.Sc. Björn Peemöller

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift

Abstract

Programmers are used to define unidirectional functions. However, many applications are based on transformations between two data structures, e.g., synchronisation tasks or maintenance between several data sources. Such applications are common cases for bidirectional transformations, which are the core functions of bidirectional programming languages. Bidirectional programs can be applied both forwards and backwards and specify transformations and similar concepts in one program. In this thesis, we evaluate existing approaches to use bidirectional transformations in an unidirectional setting as well as approaches that define their own bidirectional languages to avoid upcoming problems in existing languages. We also implement two suitable libraries for bidirectional programming in the functional logic programming language Curry. Furthermore, we make two case studies to test our implementations and investigate further applications for bidirectional programming.

Contents

1 Introduction

In the last 10 years, the topic of bidirectional programming has become of interest in many areas of computer science. Broadly speaking, bidirectional transformations are programs that run forwards and backwards. Most common applications are synchronisation processes where two similar data structures maintain the same information and need to be kept in sync as well as serialisation processes where two structures are converted into each other.

A more general concept in the context of bidirectional transformations are lenses. Instead of an one-to-one mapping, lenses provide a component to project and update a given source. This concept arises from the area of databases, where we have a set of data and make queries on that data. The query yields a so-called view – or table, and we can make further modifications on that view. These modifications lead to an out-dated set of data, which we want to update with respect to the modifications of the view, again. The work of Foster et al. (2007) sets the trend for lenses in the field of programming languages.

In general, a lens is a pair of functions that operate on two structures S and V, which are defined as follows. The value of type S represents the source of the lens, and V is the type of the view.

$$\begin{aligned} get &: S \to V \\ put &: S \times V \to S \end{aligned}$$

In the context of lenses, we have a *get* function that describes the forward direction of a lens and yields the view for a given source. The *put* function, which represents the backward direction, describes the reversed situation, but needs an additional source as argument. That is, given an original source, the *put* function produces an updated source for a given view.

However, traditional unidirectional programming languages are not well-suited for bidirectional transformations. The programmer has to maintain both functions – get and put – individually. Recent work on bidirectional programming consists of new programming languages that fit the requirements described above. In these languages, the programmer does not define a transformation with two functions, but uses predefined combinators to build more complex transformations. Furthermore, many approaches are designed to let the user define only the get function, and then derive the corresponding put function from that definition.

1.1 Goals and Contributions

The goal of this thesis is to explore the usage of a functional logic programming language like Curry for bidirectional programming.

Recent approches make heavy usage of functional programming languages like Haskell or define their own functional language that fits a specific domain of bidirectional programming, e.g. string data, tree data, or relations. Curry offers functional features like higher-order functions, lazy evaluation, and algebraic data types, which enable us to reuse most of the existing ideas for functional languages in the context of bidirectional programming. Furthermore, Curry is a logic programming language that automatically provides us with the ability to read function definitions forwards and backwards. We can use logic features like free variables in combination with the built-in search capabilities to define bidirectional transformations directly in Curry. In particular, we pursue the following goals in this thesis.

- We want to evaluate the recent work in the area of bidirectional programming that gives enough leeway for follow-up work, especially with regard to their applicability to Curry. As the result of this evaluation, we intend to implement a library for lenses that exploits Curry's functional and logic features to gain new insights of bidirectional transformations.

- As a logic programming language, Curry supports nondeterministic function definitions. Thus, we would like to investigate nondeterminism in combination with lenses. Is a nondeterministic setting applicable in the context of bidirectional programming? Do we need to rethink certain properties of lenses when we allow nondeterministic get and put functions?
- On top of the implemented lens library, we want to examine useful applications for lenses in practical examples. Recent work shows that lenses are highly applicable to field accessors for algebraic data types. Thus, we want to investigate lenses for record type declarations in Curry.

1.2 Structure

The remainder of this thesis is structured as follows. Chapter 2 and Chapter 3 provide preliminaries of this thesis. We give a short introduction to Curry, which we use for the main implementation of the concepts presented in this thesis. Readers familiar with Curry can skip this introduction and go right to Chapter 3. In Section 3.1, we give a more detailed introduction of bidirectional programming and lenses in particular. We discuss fundamental properties of lenses and the different kinds of lenses that are defined in related work in Section 3.2.

In Chapter 4, we give a detailed insight into related work and different approaches for bidirectional programming. We divide the chapter into two sections: Section 4.1 discusses combinatorial approaches, and Section 4.2 deals with related approaches that use bidirectionalisation techniques.

Chapter 5 and Chapter 6 form the main part of this thesis. We begin with the presentation of two implementations for lenses in Curry: a put-based combinatorial approach in Section 5.1 and another simple put-based library that generates its corresponding get function in Section 5.2. On top of the second implementation, we study an application of lenses in Section 6.1, which unifies the specification of pretty-printers and parsers into one lens definition. In Section 6.2, we propose a series of transformations to generate lenses as field accessors when defining record type declarations in Curry.

Last but not least, we discuss emerging challenges with bidirectional programming in Curry and conclude in Chapter 7.

2 Introduction to Curry

"You want to seduce people into using your language. The more it looks and behaves like something they know, the more likely they are to use it."

Simon Peyton Jones

The main implementations we present in this thesis are programmed with Curry. All programs are compiled with KiCS2[1] – the most recent compiler for Curry, which compiles to Haskell, as presented by Braßel et al. (2011). We also use KiCS2's interactive environment to evaluate our examples.

Curry is a functional logic programming language similar to Haskell (Jones, 2002) and created by an international development team to provide an environment, mostly for research and teaching. In the following, we assume the reader is au fait with Haskell, especially its syntax and its type system as well as general functional concepts like algebraic data types, polymorphism, higher-order functions, and lazy evaluation. Hence, we focus on features that are specific to Curry. Besides the mentioned functional features, Curry also provides nondeterminism and free variables as typical characteristics of logic programming languages.

In the remainder of this chapter, we will introduce these two logic features with a series of examples. We add explanations to further features of Curry on demand as they appear in the remainder of this thesis.

[1]In particular, we use KiCS2 version 0.3.1.

2.1 Nondeterminism

In Curry, we define a function with a set of rules. As an example, we define a function to yield the first and the last element of a list, respectively, as follows.

```
head :: [a] → a
head (x: _) = x

last :: [a] → a
last [x] = x
last (_ : xs) = last xs
```

The definition of *head* works just fine in Curry and yields the first element of a given list. However, we have to be more careful with overlapping rules in function de-finitions. Instead of matching from top to bottom like in Haskell, Curry evaluates each matching rule. Thus, the definition of *last* is nondeterministic, because a list with one element matches the first and second rule. If we use *last* on an exemplary list, we get, however, the desired result.

```
> last [1 . . 10]
10
```

In the last step of evaluating *last*, the expression *last* [10] matches both given rules. In the case of the first rule, we can apply the right-hand side and yield 10 as result. For the second rule, we make an additional function call to the remaining list. The resulting expression *last* [ ] does, however, not match any rule and silently fails. Thus, the expression *last* [10] evaluates to 10, because a failure does not yield any result.

This notion of failure is slightly different to errors in Haskell. For example, the expression *head* [ ] raises an error in Haskell. like $***$*Exception*:*Prelude.head*:*empty list*, but in Curry the expression has no result, which is expressed with ! in the interactive environment of KiCS2.

```
> head [ ]
!
```

We can use this kind of failure in our program as well by using *failed* :: *a*. *failed* is a function defined Curry's Prelude and has a polymorphic type. Thus, in the case of *head*, we can make the following adjustments to fail for an empty list.

$head' :: [a] \rightarrow a$
$head'\ [\,] = failed$
$head'\ (x\!:\ \_) = x$

As an additional example for overlapping rules, we define a function *member* that nondeterministically yields an element of a given list. We can use an approach similar to the implementation of *last*.

$member :: [a] \rightarrow a$
$member\ (x\!:\ \_) = x$
$member\ (\_ : xs) = member\ xs$

Instead of matching for a singleton list in the first rule, we match for all lists with at least one element. Thus, we yield the head element of the list for each recursion step.

```
> member "Curry"
'C'
'u'
'r'
'r'
'y'
```

Furthermore, Curry provides a special operator ? to introduce nondeterminism; this choice operator yields one of its arguments nondeterministically.

$(?) :: a \rightarrow a \rightarrow a$
$x\ ?\ \_ = x$
$\_\ ?\ y = y$

With this operator at hand, we can rewrite our implementation of *member* without using overlapping rules.

$member' :: [a] \rightarrow a$
$member'\ (x : xs) = x\ ?\ member'\ xs$

$member'' :: [a] \rightarrow a$
$member'' = foldr1\ (?)$

The first example unifies the original rules into one rule by using the choice operator. Finally, we beautify this implementation and use *foldr1* instead of an explicit recursive definition in the second example.

2.2 Free Variables

The second logic feature of Curry that we want to discuss in greater detail is the usage of free variables. Free variables are unbound variables that can be used as data generators. For instance, assume that we have the first part of a list – $[(), ()]$, and want to generate the missing suffix to create the list $[(), (), ()]$.

$> [(), ()] +\!\!+\ xs \equiv [(), (), ()]$ **where** *xs free*
$\{xs = [\,]\}$ *False*
$\{xs = [()]\}$ *True*
$\{xs = (() : \_x3 : \_x4)\}$ *False*

The free variable *xs* is denoted as such with the keyword *free* and has the same scope as locally defined functions. In order to evaluate the given expression, Curry's built-in search system generates a series of lists for *xs*. Similar to the evaluation of a nondeterministic expression, we get a series of results. The first component of the result is the binding of the occurring free variables – surrounded by curly brackets. The evaluated expression is the second component. In our example, Curry generates a series of lists starting with the empty list and stopping for lists that have three or more elements. We do not go into further detail here, and postpone further explanations about the built-in search to Section 7.2.1.

The important message to get across here is that we can use Curry's built-in search capabilities in combination with free variables to use *generate-and-test* methods in function definitions. For example, we can give an additional implementation of *last* by using free variables.

$last' :: [a] \rightarrow a$
$last'\ xs \mid \_ +\!\!+ [y] \equiv xs = y$
  where *y free*

In this example, we use an anonymous free variable, declared with an underscore "_". Anonymous free variables are a syntactical abbreviation for **let** *x free* **in** *x*. If we do not use the binding of the free variable in the remainder of our expression, we can declare it anonymously. The idea of the implementation is to generate the given list in two steps: an anonymous list for the prefix and a single element to concatenate at the end of the list. Thus, we have the last element at our fingertips and can easily yield it as result if the condition holds.

```
> last' [1 .. 5]
5
```

3 About Lenses and Other Bidirectional Transformations

"He who moves not forward goes backward."

Johann Wolfgang von Goethe,
Herman and Dorothea

Many approaches in the context of data synchronisation and data transformation are error-prone and cumbersome to maintain. Typical examples for such approaches are widespread and can be found in several areas of computer science: printers and parsers that harmonise in a meaningful way (see Section 6.1); connection between user interfaces and the underlying data (Meertens, 1998); serialisation or synchronisation processes, e.g., transforming Safari's bookmarks to be suitable for Firefox (Foster et al., 2007). We believe that in many cases the application of bidirectional programming avoids the problems mentioned and is more suitable than unidirectional programming.

In this chapter, we introduce the notion of bidirectional transformations and a more general concept called *lenses*. The first section covers bidirectional programming and its origin from databases. Furthermore, we discuss the basic functionality of bidirectional transformations. In the subsequent section, we talk about a generalisation named *lenses*. The most important part of this section involves the underlying laws that apply to lenses as well as examples to get a better intuition of the usage of lenses.

3.1 Bidirectional Programming

Problems that are based on bidirectional transformations are typically handled with two separate functions. That is, one function maps the specific value to the abstract representation and one function serves as the reverse, from the abstract value to the concrete representation. This approach is rather error-prone and tedious to maintain for two reason. Firstly, we have to keep the two functions in sync by hand in order to guarantee correctness. Secondly, changes in one of the representations affect both functions due to certain round-tripping rules that we introduce later. This unidirectional programming mechanism is well-studied and many programmers are familiar with this paradigm. In contrast, bidirectional programming is a new approach to a specific domain of problems, which becomes more and more popular in different areas of computer science. Software engineering, programming languages, databases and graph transformations are some of the fields of computer science that currently participate in research activities concerning bidirectional transformations. For a more detailed introduction to the cross-discipline of bidirectional transformation, we recommend the work of Czarnecki et al. (2009). In the remainder of this thesis, we focus on bidirectional transformation from the perspective of the programming language community.

So, what is the new, challenging feature of bidirectional programming keeping researchers busy? A bidirectional transformation does not consist of two functions like in the unidirectional way, but of one function that can be read both forwards and backwards. We distinguish between a forward function $get :: A \to B$ and a backward function $put :: B \to A$; A is most commonly called the source and B is the view. The naming convention originates from applications in databases. These two functions form, in the easiest approach, a bijection between A to B and back. We visualise this idea of two types and their corresponding transformation functions in Figure 3.1.

In the next section, we discuss a more general approach of bidirectional transformation called lenses. Lenses are one of the most popular abstractions in bidirectional programming. A statement about the status quo of bidirectional programming is postponed to Chapter 4 and Section 5.2, where we discuss several implementation approaches. Another good source for further reading is the paper by Foster et al. (2010), which contrasts three different approaches of bidirectional programming.

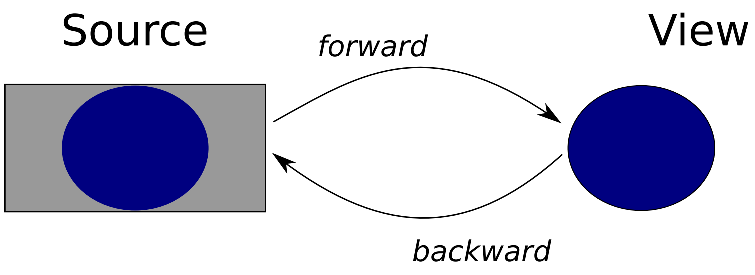

Figure 3.1.: Bijective relation between two types A and B

3.2 Lenses

Lenses describe bidirectional transformations, which originate from databases as introduced by Bancilhon and Spyratos (1981). In the setting of lenses, the *get* function describes a transformation from A to B. In most applications A is a product type, and B is one component of that product. Thus, the *get* direction of a lens discards information when projecting from A to B. The *put* function synchronises a given, potentially updated, view with respect to the original source.

As an example, let us take a look at a bidirectional transformation with a pair of *String* and *Integer* as the domain of the source, and *String* as the view's domain. In order to define an appropriate *get* function, we need a get function of type $(String, Integer) \rightarrow String$. In Haskell, or Curry, there already exists a function with such a type, namely *fst*.

$$
\begin{aligned}
&fst_{get} :: (String, Integer) \rightarrow String \\
&fst_{get}\ (str, \_) = str
\end{aligned}
$$

Our function fst_{get} yields the first component of a pair with no further changes or adjustments to the value; this definition is equivalent to *fst*. The put function has the type $fst_{put} :: (String, Integer) \rightarrow String \rightarrow (String, Integer)$; we define a function that sets the first component of a pair with a given string without further ado.

$$
\begin{aligned}
&fst_{put} :: (String, Integer) \rightarrow String \rightarrow (String, Integer) \\
&fst_{put}\ (\_, int)\ newStr = (newStr, int)
\end{aligned}
$$

In order to see the *get* and *put* function of such a lens in action, we give two exemplary expressions.

$> fst_{put}$ ("foo", 42) "bar"
("bar", 42)
$> fst_{get}$ ("bar", 42)
"bar"

Moreover, a popular example from databases shows the general idea quite well: we have a database with a data set S and a query that yields a table T matching the given criteria. The query is the forward transformation *get*. In a second step, we modify the resulting table, because we recognise a misspelled name field or suchlike, which yields an updated table T'. We definitely want to propagate the update back to our database; this is where the *put* function comes into play. The *put* function synchronises our changes of the view, T', with the original database set S.

Figure 3.2 illustrates the idea of an updated view – the red circle – that is synchronised with its original source – the grey rectangle with the blue circle.

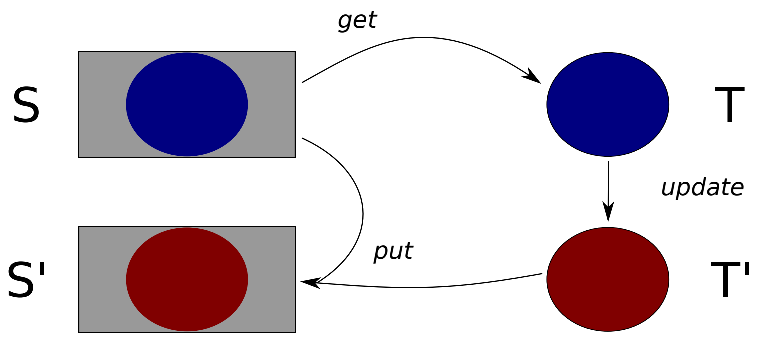

Figure 3.2.: Lenses in action: projecting a value with *get* and updating the original source with *put*

Furthermore, the literature distinguishes between two categories of lenses: *symmetric* and *asymmetric* lenses. The names *asymmetric* and *symmetric* describe the focus on the given pair of source and view.

In a symmetric setting, each structure A and B contains information that is not present in the other. We can update both structures, which leads to two put functions: $putl :: B \to A \to B$ to update B and a put function $putr :: A \to B \to A$ to update A.

In an asymmetric setting, we only consider changes of the view that will be propagated back to the source; this restricted view implies that the given source does not change in the meantime. In comparison to the bijective setting, the *put* function takes the initial source as argument to synchronise the updated view with that source. This additional argument leads to a slight change in the type of the *put* function: $put :: A \to B \to A$. Our code example from above is a representative for an asymmetric lens.

In the following, we will only examine asymmetric lenses. For a detailed introduction to symmetric lenses, consider reading the work of Hofmann, Pierce, and Wagner (2011) or the dissertation of Wagner (2014). A detailed listing of different properties that are applicable for lenses can be studied in the work of Pacheco et al. (2013b).

3.2.1 PutGet Law

So far, we characterised lenses as a bidirectional transformation with an adapted *put* function. It is important to note that lenses fulfil certain laws. The first two of three laws that we discuss are also called round-tripping rules, because they state how *get* and *put* interact when used consecutively. The first law states that if, using the same lens, we put something in and extract it again, we get the same thing back.

$$get\ (put\ s\ v) = v \tag{PutGet}$$

This law is called *PutGet*: we first *put* a new value in our source and then try to *get* it out again.

In order to give an example, we use the lens definition of fst_{get} and fst_{put} from above to check the *PutGet* law. We can show that the defined pair of fst_{get} and fst_{put} behaves according to the law for every initial pair and additional string.

Proof. For all w, v and v', where (v, w) is of type $(String, Integer)$ and v' is of type $String$, it holds that $fst_{get}\ (fst_{put}\ (v, w)\ v') = v'$.

$$\begin{aligned}
& fst_{get}\ (put\ (v, w)\ v') \\
\equiv\quad & \{ \text{ definition of } fst_{put} \ \} \\
& fst_{get}\ (v', w) \\
\equiv\quad & \{ \text{ definition of } fst_{get} \ \} \\
& v'
\end{aligned}$$

$\square$

3.2.2 GetPut Law

In addition to the *PutGet* law, lenses are also supposed to fulfil a second round-tripping criterion. The *GetPut* law states that if we get a view out of a source and put it back unmodified again, the source does not change at all, as if nothing happened. This law can be interpreted as a stability property. That is, a lens is stable if nothing *magical* happens during an update or a selection.

$$\mathit{put}\ s\ (\mathit{get}\ s) = s \tag{GetPut}$$

Proof. With our example above, we obtain the following equation, where for all w and v where (v, w) is of type $(\mathit{String}, \mathit{Integer})$, it holds that $\mathit{put}\ (v, w)\ (\mathit{get}\ (v, w)) = (v, w)$.

$$\begin{aligned}
& \mathit{put}\ (v, w)\ (\mathit{get}\ (v, w)) \\
\equiv\quad & \{ \text{ definition of } \mathit{get} \ \} \\
& \mathit{put}\ (v, w)\ v \\
\equiv\quad & \{ \text{ definition of } \mathit{put} \ \} \\
& (v, w)
\end{aligned}$$

$\square$

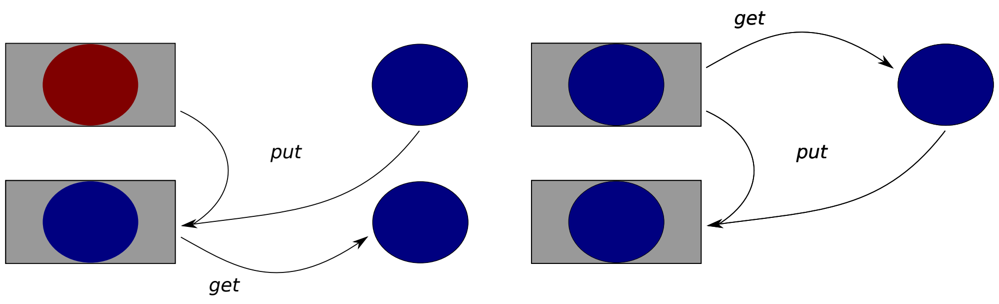

Figure 3.3.: PutGet Law (left) and GetPut Law (right)

In Figure 3.3, we illustrate both lens laws; the different colouring of the view distinguishes the original value of the view and the new updated value. In the pioneering work of Foster et al. (2007) in the topic of bidirectional programming and lenses, a lens is called *well-behaved* if both laws, the *GetPut* and the *PutGet* law, hold.

3.2.3 Partial Lenses

More and more frameworks for bidirectional transformations and bidirectional programming languages, respectively, endorse a weaker notion of the presented *PutGet* and *GetPut* laws. In our current notion of the laws, we only consider total *get* and total *put* functions. Most of the time we want, however, to be able to work with functions that are not total. For example, the classical function $head :: [a] \to a$ to select the first element of a list is only partial, because we cannot select an element for an empty list. We can define a lens that uses $head'$ as definition for its get direction. In order to form a lens, we need a put function as well: the put direction replaces the head element of the given list with a new element.

```
head' :: [a] -> a
head' []       = error "head': undefined for empty lists"
head' (x : xs) = x

replaceHead :: [a] -> a -> [a]
replaceHead []       y = [y]
replaceHead (_ : xs) y = y : xs
```

The given definition for the put direction is total, thus, we can observe the expected behaviour. In the following, we make some test function calls to check if the given lens definition is reasonable in regard to the lens laws.

```
> replaceHead [1, 2, 3, 4, 5] 42
[42, 2, 3, 4, 5]
> replaceHead [] 13
[13]
> head' (replaceHead [1, 2, 3, 4, 5] 10)
10
> replaceHead [1, 2, 3, 4] (head' [1, 2, 3, 4])
[1, 2, 3, 4]
```

The first two test expressions show the behaviour of *replaceHead*; it becomes apparent that *replaceHead* never yields an empty list as result. Thus, the PutGet law obviously holds for all possible values. The last expression is an example with a non-empty lists, where the GetPut holds as well. However, the get direction of the just defined lens, i.e. *head′*, is not defined for empty lists. Thus, the given lens does not fulfil the GetPut law for empty lists.

$> \mathit{replaceHead}\ [\,]\ (\mathit{head}'\ [\,])$
$\mathit{Main} : \mathit{UserException}$ `"head': undefined for empty lists"`

In order to use partial lenses like proposed by Pacheco et al. (2013b), we need to adjust the lens laws by means of partiality. In the following, the condition $(f\ x) \downarrow$ is satisfied if the function f yields a result for the argument x. We define the partial version of *PutGet* and *GetPut* in terms of inference rules. That is, if the above condition is not satisfied, the equation below does not need to be checked, and the rule trivially holds.

$$\frac{(\mathit{put}\ s\ v) \downarrow}{\mathit{get}\ (\mathit{put}\ s\ v) = v} \qquad \text{(Partial-PutGet)}$$

$$\frac{(\mathit{get}\ s) \downarrow}{\mathit{put}\ s\ (\mathit{get}\ s) = s} \qquad \text{(Partial-GetPut)}$$

In our example, we check if *head′* is defined for the given source first.

$> \mathit{head}'\ [\,]$
$\mathit{Main} : \mathit{UserException}$ `"head': undefined for empty lists"`

Since this is not the case, we do not apply the put direction; the condition only needs to be satisfied if the first application yields a valid result. Thus, the lens consisting of *head′* and *replaceHead* is a valid lens with respect to the PutGet law and the Partial-GetPut law.

As a second example, we define a lens with a put function that is similar to the well-known function $\mathit{take} :: \mathit{Int} \rightarrow [\,a\,] \rightarrow [\,a\,]$ and a corresponding get function, which behaves like the function $\mathit{length} :: [\,a\,] \rightarrow \mathit{Int}$ in Haskell and Curry, respectively.

```
take' []       _ = []
take' (x : xs) n
  | n ≡ 0       = []
  | n > 0       = x : take' xs (n − 1)
  | otherwise   = error "take': negative value"
length' []       = 0
length' (x : xs) = 1 + length' xs
```

As a minor adjustment, we define *take′* only on positive integer values to better harmonise with *length′*. Similar to before, we can observe that *length′* only yields positive integer values as result, thus, the GetPut law holds trivially for non-empty and empty lists.

```
> length' [1, 2, 3, 4]
4
> take' [1, 2, 3, 4, 5] 3
[1, 2, 3]
> take' [1, 2, 3, 4] (length' [1, 2, 3, 4])
[1, 2, 3, 4]
> take' [] (length' [])
[]
```

Due to the partial definition of *take′*, the defined lens does not fulfil the PutGet law as we can see from the following expressions.

```
> length' (take' [1, 2, 3] (−3))
"take': negative value"
> take' [] (−1)
"take': negative value"
```

Nevertheless, our second example is a valid lens with respect to GetPut and Partial-PutGet.

3.2.4 PutPut Law

There is also a third lens law, which is called *PutPut*. A lens satisfies the PutPut law if we run two consecutively *put* operations on a source with two different views, but only the second *put* matters. We can formulate this law with the following equation.

$$put\ (put\ s\ v)\ v' = put\ s\ v' \tag{PutPut}$$

Lenses that fulfil all three laws – GetPut, PutGet, and PutPut – are called *very well-behaved*. The PutPut law, however, does not play an important role in most applications, because the preconditions are too strong. That is, plenty of constructive well-behaved lenses are not very well-behaved. For example, the last lens we defined changes the source list dependent on the given view element. Thus, two consecutive calls to the put function with different view values can yield a different result than the second call alone.

```
> take′ (take′ [1, 2, 3, 4, 5] 1) 3
[1]
> take′ [1, 2, 3, 4, 5] 3
[1, 2, 3]
> take′ [1] 3
[1]
```

In the first expression, we start with the list $[1, 2, 3, 4, 5]$, and reduce it to just the first element, i.e., *take′* $[1, 2, 3, 4, 5]$ 1 yields [1]. The second application of put reduces the list to the first three elements; since the list only contains one element, we get $[1]$ as result again. The PutPut law states that two consecutive calls should have the same effect as just the latter one. In our case, the second put application to the source list $[1, 2, 3, 4, 5]$ yields the first three elements, i.e. the resulting list is [1,2,3], which differs from the result with consecutive put calls.

Nevertheless, the PutPut law can be applicable for a number of convenient lens definitions. For example, the lens consisting of *head′* and *replaceHead* obeys the PutPut law. In the put direction, we replace the head of a given list; thus, for two consecutive *replaceHead* actions, only the latter matters.

```
> replaceHead (replaceHead [1, 2, 3, 4] 13) 42
[42, 2, 3, 4]
> replaceHead [1, 2, 3, 4] 42
[42, 2, 3, 4]
```

4 Different Implementation Approaches

Bidirectional programming is a rising topic in the field of computer science, and many different approaches exist to tackle the problem. These approaches come from different disciplines of computer science such as databases, graph transformation, programming languages, and GUI design. This chapter summarises the two main approaches and highlights differences as well as some details.

The two main techniques to work with bidirectional transformations are combinatorial languages and bidirectionalisation. A combinatorial language is either defined as a DSL in a general purpose programming language or as a new programming language, and it provides a set of primitives, which the user combines to define more complex structures. The core primitive is a set of (get, put) function pairs for different lenses. The two aproaches differ in what must be defined by the programmer: either both functions or one unidirectional function that is synthesised to a bidirectional one.

The remainder of this chapter introduces combinatorial and bidirectionalisation approaches for lenses. The former approach has two subcategories, because the implementation can focus on defining either a get function or a put function; such a subdivision was not investigated for the bidirectionalisation of lenses yet. In this context, we discuss advantages and disadvantages of defining a get function and present a first proposal by Fischer et al. (2014) to set the focus on the put function.

4.1 Combinatorial Lenses

The first combinatorial technique is the pioneer work by Foster et al. (2007), who designed a domain-specific programming language to define bidirectional transformations on tree-structured data. Foster et al. formulate fundamental laws concerning lenses[1], combine these laws with the intuitive behaviour of lenses, and use fundamental tools from domain theory to also define lenses by recursion. They lay the focus of their language's design on robustness and ease of use. That is, their language guarantees well-behaved lens definitions and the totality of the primitive transformations with an integrated type system. The underlying type system, and with that the type safety, is the main contribution to the field of bidirectional programming. The authors state close connections with topics from the database community. Lenses are a well-known abstractions in databases concerning tables and queries: the work of Bancilhon and Spyratos (1981) tackles problems concerning precision and continuity, and the property of well-behaved lenses corresponds to ideas by Keller (1985).

4.1.1 Combinators for Tree-Structures

In their publication, Foster et al. define a handful of primitive lens combinators for trees, and the combination of these primitive lenses results in a powerful abstraction to describe transformations. The most important primitives are the composition, identity and constant lenses. These definitions work on arbitrary data structures, whereas all other combinators specialise on tree data structures only. The defined transformations are closely related to tables and views from databases: a transformation maps a concrete structure into an abstract view, and a possibly modified abstract view together with the original concrete structure maps to a modified concrete structure.

In the DSL, the user defines the forward transformation in a straightforward fashion, whereas the backward transformation is the result of reading the definition backwards.

The following expression shows a tree with two labels, *fst* and *snd*, representing a pair, which corresponds to a pair (42, `"Hello World"`) in Curry.

$$aPair = \left\{ \begin{array}{l} \quad \text{fst} \rightarrow 42 \\ , \ \text{snd} \rightarrow \text{"Hello World"} \end{array} \right\}$$

[1]We already presented these laws in Section 3.2.1, in particular, PutGet, GetPut and PutPut.

As an example, we define a lens that yields the first component of a pair – like the one we defined above. Foster et al. use $S \Leftrightarrow V$ as representation for a lens with a source of type S and a view of type V.[2] The corresponding get and put function of a lens is defined as follows.

$$\nearrow\colon (S \Leftrightarrow V) \times S \to V \qquad \text{(get function)}$$

$$\searrow\colon\ (S \Leftrightarrow V) \times (S, V) \to S \qquad \text{(put function)}$$

They define a tree combinator *filter p d* to keep particular children of the tree. The predicate p describes the set of names that we want to keep in the tree and d is a default value for missing information in the put direction. In order to distinguish between empty trees and empty sets, we represent the empty tree as $\{\,\}$. We define the following expression to extract the first component of a given pair.

$$\begin{aligned}
&(\mathit{filter}\ \{\mathit{fst}\}\ \{\,\}) \nearrow \mathit{aPair} \\
= &(\mathit{filter}\ \{\mathit{fst}\}\ \{\,\}) \nearrow \left\{ \begin{array}{l} \quad \text{fst} \to 42 \\ ,\ \text{second} \to \texttt{"Hello World"} \end{array} \right\} \\
= &\left\{\ \text{fst} \to 42\ \right\}
\end{aligned}$$

As a second example, we use the put function of the same lens to change the first component of our pair to 13.

$$\begin{aligned}
&(\mathit{filter}\ \{\mathit{fst}\}\ \{\,\}) \searrow (\mathit{aPair}, 13) \\
= &(\mathit{filter}\ \{\mathit{fst}\}\ \{\,\}) \searrow \left(\left\{ \begin{array}{l} \quad \text{fst} \to 42 \\ ,\ \text{snd} \to \texttt{"Hello World"} \end{array} \right\}, 13 \right) \\
= &\left\{ \begin{array}{l} \quad \text{fst} \to 13 \\ ,\ \text{snd} \to \texttt{"Hello World"} \end{array} \right\}
\end{aligned}$$

The work of Foster et al. originates in the Harmony project[3] (2006; 2006; 2003), a generic framework to synchronise tree-structured data. One example used repeatedly throughout their work is the synchronisation of different browser bookmarks, calendar, and address book formats. They continued their work on lenses with a project

[2] In their pair, Foster et al. use C and V as representative for the concrete and abstract value, respectively.

[3] `https://alliance.seas.upenn.edu/~harmony/old/`

called Boomerang[4] (2008) that focuses on string data instead of tree-structured data. Their language is also used beyond their own research projects: the developer team of *Augeas*[5] uses the Boomerang language as a framework for their configuration API. Both languages, Harmony and Boomerang, are based on a type system to guarantee well-behaved and total lenses as well as a rich set of lens combinators to define powerful transformations on tree structures and strings, respectively.

4.1.2 Preparation for Put-Based Combinators

Other combinatorial approaches for lenses exist, and they all focus on specifying a get function. The appropriate put function is then propagated through the definition of the used get-based combinators. For example, Pacheco and Cunha (2010) designed a point-free DSL in Haskell, in which the programmer defines only the get transformations. Fischer et al. (2012) are the first to push the usage of the put definition instead. It seems quite obvious that both – the forward and the backward – function of a bidirectional transformation can be used for bidirectionalisation. Nevertheless, so far, the current techniques mostly pursue the idea of Foster et al.

In the work of Fischer et al., it becomes apparent that typical problems of get definitions are the ambiguity of the derived put functions. That is, in several cases there exists more than one appropriate put function to correspond with the get definition. These problems concerning ambiguity arise when the defined get function is not injective; we will discuss this topic in more detail in Section 5.2. This ambiguity can be eliminated when we define the put direction instead. Fischer et al. show that the corresponding get function for a defined put function is unique if certain requirements apply to the put function. They prepare their theorem with some transformations on the PutGet and GetPut laws; instead of the classical representation, they express their requirements based on just the put definition. Fischer et al. (2012) state that the PutGet law can be reformulated as injectivity property of the put function. For that purpose, let us recapitulate the PutGet law.

$$get\ (put\ s\ v) = v$$

[4]`https://alliance.seas.upenn.edu/~harmony/`
[5]`http://augeas.net`

As a first step, we can express the equation in a more functional manner. The view v occurs on both sides of the equation, so we can use η-reduction to simplify the equation.

$$get \circ put\ s = id \tag{PutGet'}$$

The $(\circ)$ operator defines function composition, that is, the equation *PutGet'* says that get is a left inverse for *put s* for all sources s. A function $f :: A \to B$ is injective if and only if there exists a function $g :: B \to A$ such that $g \circ f = (id :: A \to A)$. In the *PutGet'* equation above, we have the function $put\ s :: V \to S$ that corresponds to f and the counterpart $get :: S \to V$ as the equivalent to g. The identity function in the equation above is obviously of type $id :: V \to V$, because we η-reduced the view argument $v :: V$, thus, V must be the resulting type as well.

In the end, we can express the identity property of the first round-tripping rule *PutGet* using just the put function. Thus, Fisher et al. demand *put s* to be injective for all sources s.

$$s' \in put\ s\ v \wedge s' \in put\ s\ v' \Rightarrow v = v' \tag{PutInj}$$

Similar to the η-reduction for the PutGet law, we can rewrite the GetPut law as well. It is a bit more complicated to rewrite the equation

$$put\ s\ (get\ s) = s$$

because of the two usages of the variable s. In order to simplify the equation, we need to use a pair as argument; then, we can apply *put* to this argument. The resulting argument is a function depending on s. The notion of using tuples instead of multiple arguments and vice versa is called *unccurrying* and *currying*, respectively. In this case, we need to apply the function $uncurry :: (a \to b \to c) \to (a, b) \to c$ to the $put :: S \to V \to S$ function in order to get a modified function $put' = uncurry\ put :: (S, V) \to S$ that takes a pair of S and V as its argument. With this auxiliary function at hand, we can express a point-free version of the GetPut law.

$$put' \circ (\lambda s \to (s, get\ s)) = id$$

For this equation, we can conclude that put', i.e., *uncurry put*, has a right inverse. That is, put' is surjective for all sources S, because a function $f :: A \to B$ is surjective if and only if there exists a function $g :: B \to A$ such that $f \circ g = id :: B \to B$ holds.

$$\forall s \in S\ \exists s' \in S : put'\ (s', get\ s') = s \tag{PutSurj}$$

Actually, this equation only holds for total put functions, because the equation requires to be fulfilled for all values s of the resulting type S. Fischer et al. formulate idempotence of $\lambda s \to put\ s\ v$ for all views v as an additional requirement for well-behaved lenses.

$$s' \in put\ s\ v \Rightarrow s' = put\ s'\ v \qquad \text{(PutTwice)}$$

In particular, *PutTwice* is a special case of the PutPut law.

4.1.3 Put-Based Combinators

Fischer et al. verified that there is only one get function for an arbitrary put function that obeys *PutInj* and *PutTwice*[6], and this get function can be determined with the following equation.

$$get\ s = v \Leftrightarrow s = put\ s\ v \qquad \text{(relation between } get \text{ and } put\text{)}$$

As a next step, Fischer et al. (2014) developed a put-based language in their subsequent work. They present a general design of put-based languages as well as an implementation of an embedded DSL for Haskell. The main idea of the put-based language is to provide a handful of combinators. The language allows to define the put function of a lens. The put function of a lens defines the synchronisation strategy between a modified view and a given source. In order to provide a wide scope of such strategies, the put-based language is based on functions with monadic effects. A lens is represented as

```
type LensPG' m s v = Maybe s → v → m s,
```

where m denotes a monadic context. Depending on the given instance of the monad, the programmer can influence the synchronisation behaviour. For example, we can program with traditional lenses without monadic effects by using the *Identity* monad.

```
data Identity a = Identity {runIdentity :: a}
instance Monad Identity where
  return valA          = Identity valA
  Identity valA >>= f  = f valA
type LensPG s v = LensPG' Identity s v
```

[6] In later work of Hu et al. (2014) these both properties are called *PutDetermination* and *PutStability* respectively

The put-based language is built upon a handful of combinators, which are inspired by the combinators of Foster et al., e.g., identity and constant lenses as well as lenses for filter, composition, products, sums and conditionals. The language assures well-behavedness by construction. All combinators, including composition, form well-behaved lenses. Thus, the composition of predefined combinators form well-behaved lenses as well. Additionally, the Haskell library provides functions to define custom lenses. Due to the lack of static checks concerning well-behavedness, the user can use the function *checkGetPut* and *checkPutGet*[7] to check for the corresponding lens laws at runtime. For simplicity reasons, we refrain from defining these check functions and focus on the lens combinators instead.

We can rebuild the example given above in terms of the put-based language with a predefined combinator called *addfst* that works on pairs.

```
addfst :: (Maybe (s₁, v) → v → m s₁) → LensPG′ m (s₁, v) v
addfst f = checkGetPut put′
  where
    put′ :: LensPG′ m (s₁, v) v
    put′ s v = f s v ≫= λs₁ → return (s₁, v)
```

The lens *addfst* adds a second component to the current source to create a pair. The first argument of *addfst* is a function to create the first component of the pair from the given source and view. We can use this combinator to define a lens fst_{PG}, which projects a pair to its first component.

```
fst_PG :: LensPG′ m (s₁, v) v
fst_PG =
  addfst (λs v → maybe (fail "Undefined") (return ∘ fst) s)
```

If there is no source available, we cannot do anything meaningful without losing generality, thus, we just throw an error.[8] Otherwise, we use *fst* to select the first component of the given pair.

[7] In the associated paper, Fischer et al. use the name *enforceGetPut* instead.

[8] The function *fail* is part of the Monad type class, thus, we can implement a mechanism to catch such errors.

$> get\ fst_{PG}$ (42, "Hello World")
42
$> put\ fst_{PG}$ (42, "Hello World") 13
(13, "Hello World")

We will discuss the actual implementation in Section 5.1 in more detail, because the Haskell library `putlenses`[9], which implements the ideas of the presented paper by Fischer et al. (2014), forms the basis of an implementation in Curry that we review later.

4.2 Bidirectionalisation

Bidirectionalisation is the process of automatically transforming unidirectional functions into bidirectional functions. In the following, we present two techniques to bidirectionalise a unidirectional get function: the first technique syntactically derives a put function for a given get function; the second technique takes a more semantic approach to generate an appropriate put function at runtime. Both techniques have their advantages and disadvantages. Hence, the authors also worked out a combined approach, which yields results at least as good as either one of the two techniques. We also give a short introduction for the combined approach.

4.2.1 Syntactic Bidirectionalisation

Matsuda et al. (2007) introduce a general first-order functional language called *VDL*. Their language has two syntactical restrictions, which we have to keep in mind when talking about derivable functions: the defined functions have to be *affine* and *treeless*. A function definition is affine if every variable on the left-hand side is used at most once on the right-hand side of the definition; treeless characterises function definitions without intermediate data structures. Nevertheless, VDL allows function definitions using arbitrary algebraic data structures, e.g., lists and trees. The user defines unidirectional get functions, and VDL automatically derives appropriate put functions. The underlying derivation algorithm is based on a similar idea in the field of databases, but follows a syntactical approach. As a first step, VDL derives a *view*

[9]`http://hackage.haskell.org/package/putlenses`

complement function $f^c : S \to (S \setminus V)$ for a get[10] function $f : S \to V$. The main idea of the view complement function f^c is to preserve any information disregarded by the original function f, such that $f\ x$ and $f^c\ x$ are sufficient to reconstruct the argument x. For instance, if we consider the function $f : (A, B) \to A, f = \mathit{fst}$, then $f^c : (A, B) \to B, f^c = \mathit{snd}$ serves as a valid view complement function.

Matsuda et al. require the view complement function f^c to be injective when paired with the view function f; a pair of X and Y is denoted with $X \triangle Y$. That is, $(f \triangle f^c)$ is injective for a get function f and its complement f^c. As the last step, an inverse transformation is performed on the pair to obtain the put function. The following equations illustrate the definitions given above.

$$\begin{aligned} f &: S \to V && \text{(get function)} \\ f^c &: S \to V' && \text{(view complement function)} \\ f \triangle f^c &: S \to (V \times V'), && \text{(tupled function)} \\ (f \triangle f^c)\ \mathit{valS} &= (f\ \mathit{valS}, f^c\ \mathit{valS}) \end{aligned}$$

All in all, the put function can be derived if the paired function and its inverse can be derived effectively.

$$\mathit{put}_{<f,f^c>}(s, v) = (f \triangle f^c)^{-1}(v, f^c\ s) \qquad \text{(derived put function)}$$

In their paper, the authors give an algorithm to automatically derive a view complement function for a given get function.

There are two details that we did not examine so far: a determinism property for the inverse transformation and further requirements for the complement function. The inverse transformation is not guaranteed to be deterministic; it is possible to generate equations with overlapping left-hand sides. In the case of nondeterministic programs, a backtracking search becomes necessary. However, Voigtländer et al. (2010) state that it would be preferable to obtain only deterministic programs. Furthermore, the complement, which we derive in the first step, must be injective and minimal with respect to a collapsing order, which needs to be defined. Fortunately, injectivity is decidable in VDL, and the proposed algorithm is sound and complete.

[10]Matsuda et al. call *get* functions view functions instead.

4.2.2 Semantic Bidirectionalisation

On the other hand, Voigtländer (2009) introduces an approach for semantic bidirectionalisation using free theorems to prove consistency conditions. Voigtländer defines a function $\mathit{bff} :: (\forall a.[a] \rightarrow [a]) \rightarrow (\forall a.[a] \rightarrow [a] \rightarrow [a])$ in Haskell, whose first argument is a polymorphic get function and which yields the appropriate put function. The acronym *bff* stands for *bidirectionalisation for free*, which is the title of the underlying paper.

In contrast to the syntactic approach we studied before, the resulting put is a functional value, which is semantically equivalent to a syntactically derived put function. The advantage is that we have fewer language restrictions, because we can use Haskell as our language of choice instead of a sublanguage. The *bff* function takes any Haskell function of appropriate type as its argument. The *bff* function is defined on lists, but the approach is also applicable for all data structures, which have shape and content, i.e., which apply to the category of containers as defined by Abott et al. (2003).

The approach exploits the fact that the get function is polymorphic over its first argument, i.e., the container's element. We can assume that it does not depend on any concrete element of its container, but only on positional information that are independent of the elements' value. The use of free theorems allows us to inspect the effect of the *get* transformation without knowing about the explicit implementation. That is, we can apply the get function to a specific list, e.g., integer values in ascending order, and observe the positional information. In the following, we call such a specific list *template*.

We show the internal behaviour of *bff* with an example. As underlying get function, we use our typical example *fst* again. In order to fit the restrictions of *bff*, we adjust *fst* to yield the head element as a singleton list.

$$\begin{aligned}&\mathit{get}_{\mathit{fst}} :: [a] \rightarrow [a]\\&\mathit{get}_{\mathit{fst}}\ (x{:}\ \_) = [x]\end{aligned}$$

In the following, we want to update a given source with a new view. We have a source $\mathit{eitherValues} = [\mathit{Left}\ 10, \mathit{Left}\ 12, \mathit{Right}\ \mathit{True}, \mathit{Left}\ 13]$, a list of ascending order as template, and an updated view $\mathit{upd} = [\mathit{Right}\ \mathit{False}]$.

The idea is to first construct a mapping between each element of the given list and the template: each element of *eitherValues* is mapped with the corresponding element of the template.

$$
\begin{array}{l}
\mathit{mapping} :: [\,a\,] \rightarrow [\,(\mathit{Int}, a)\,] \\
\mathit{mapping} = \mathit{zip}\ [\,0\,..\,] \\
>\mathit{mapping}\ \mathit{eitherValues} \\
[\,(0, \mathit{Left}\ 10), (1, \mathit{Left}\ 12), (2, \mathit{Right}\ \mathit{True}), (3, \mathit{Left}\ 13)\,]
\end{array}
$$

As a second step, we simulate the behaviour of the get function on the template list: we apply the get function to the template. Then, each element of the resulting list is mapped with the corresponding elements of the updated view *upd*.

$$
\begin{array}{l}
\mathit{mapping2} :: ([\,a\,] \rightarrow [\,a\,]) \rightarrow [\,\mathit{Int}\,] \rightarrow [\,a\,] \rightarrow [\,(\mathit{Int}, a)\,] \\
\mathit{mapping2}\ \mathit{getF}\ \mathit{is} = \mathit{zip}\ (\mathit{getF}\ \mathit{is}) \\
>\mathit{mapping2}\ (\mathit{get}_{\mathit{fst}}\ [0, 1, 2, 3])\ [\,\mathit{Right}\ \mathit{False}\,] \\
\quad \text{-- zip [0] [Right False]} \\
[\,(0, \mathit{Right}\ \mathit{False})\,]
\end{array}
$$

We combine both mappings with precedences to the second: if we find a value in both mappings, we choose the one from the updated view. That is, the auxiliary function *union′* combines the two lists accordingly.

$$
\begin{array}{l}
\mathit{mapping3} :: [\,(\mathit{Int}, a)\,] \rightarrow [\,(\mathit{Int}, a)\,] \rightarrow [\,(\mathit{Int}, a)\,] \\
\mathit{mapping3}\ m_1\ m_2 = \mathit{union}'\ m_1\ m_2 \\
>\mathit{mapping3}\ [\,(0, \mathit{Left}\ 10), (1, \mathit{Left}\ 12), (2, \mathit{Right}\ \mathit{True}), (3, \mathit{Left}\ 13)\,] \\
\qquad\qquad\quad [\,(0, \mathit{Right}\ \mathit{False})\,] \\
[\,(0, \mathit{Right}\ \mathit{False}), (1, \mathit{Left}\ 12), (2, \mathit{Right}\ \mathit{True}), (3, \mathit{Left}\ 13)\,]
\end{array}
$$

As last step, every element in the template is replaced by its associated value according to the combined mapping. With all the ground prepared, we can define the *bff* function given in the paper.

$$\begin{aligned}
&\mathit{bff} :: (\forall a.[a] \to [a]) \to (\forall a.[a] \to [a] \to [a])\\
&\mathit{bff}\ \mathit{get}_f\ s\ v = \mathit{map}\ (\mathit{fromJust} \circ \mathit{flip}\ \mathit{lookup}\ (\mathit{mapping3}\ m_1\ m_2))\ m_1\\
&\quad \textbf{where}\\
&\qquad m_1 = \mathit{mapping}\ s\\
&\qquad m_2 = \mathit{mapping}_2\ \mathit{get}_f\ (\mathit{map}\ \mathit{snd}\ m_1)\ v
\end{aligned}$$

$$\begin{aligned}
&> \mathit{bff}\ \mathit{get}_{\mathit{fst}}\ \mathit{eitherValues}\ \mathit{upd}\\
&[\mathit{Right}\ \mathit{False}, \mathit{Left}\ 12, \mathit{Right}\ \mathit{True}, \mathit{Left}\ 13]
\end{aligned}$$

Voigtländer also defines two additional functions, bff_{EQ} and bff_{ORD}, which use the functions of the type classes *Eq* and *Ord*, respectively. In order to apply his approach for a *get* function that duplicates elements, the defined mapping fails because of its simple definition. In a more practical mapping, equivalent elements in the original list need to map to the same element in the template. Thus, we need to compare the elements within the list: the *Eq* type class comes into play. For the function bff_{Ord}, the mapping needs a similar, but rather complicated and more technical adjustment to allow the use of free theorems again.

As the major disadvantage, any get function that changes the shape of its elements fails due to non-trackable updates. That is, the semantic approach is limited to get functions that preserve the shape of the given list.

As an enhancement for semantic bidirectionalisation, Wang and Najd (2014) presented a generalisation that extends the range of *get* function to higher-order functions that are not expressed by type classes or depend on different type classes than *Eq* and *Ord*. Instead of three single functions, like in Voigtländer's work, Wang and Najd define a function that takes an observer function, which gives rise to equivalence properties of the elements, and an observer function for the template. The approach uses these observer functions to build the mappings as in the original approach. These mappings are called observation tables and generalise the explicit usage of different functions for different type class dependencies.

As a second enhancement, Matsuda and Wang (2013a) introduce a type class to extend the range of *get* functions to monomorphic transformations. The main idea is to provide a type class $\mathit{PackM}\ \delta\ \alpha\ \mu$ to convert monomorphic functions into polymorphic ones.

$$\begin{array}{l}\textbf{class}\ (\mathit{Pack}\ \delta\ \alpha, \mathit{Monad}\ \mu) \Rightarrow \mathit{PackM}\ \delta\ \alpha\ \mu\ \textbf{where}\\ \quad \mathit{liftO} :: \mathit{Eq}\ \beta \Rightarrow ([\delta] \to \beta) \to [\alpha] \to \mu\ \beta\\ \textbf{class}\ \mathit{Pack}\ \delta\ \alpha \mid \alpha \to \delta\ \textbf{where}\\ \quad \mathit{new} :: \delta \to \alpha\end{array}$$

The type variable δ represents the type of the concrete data structure, whereas α is the type of the abstracted value. The last type variable μ is the used monad, which tracks the transformation on values of the concrete structure; these tracking data are called observation histories. The additional type class *Pack* δ α constructs labels to track information regarding the location of values within the concrete structure. In short, the approach replaces monomorphic values in the definition of the *get* function, which are, for example, used for comparisons with polymorphic values.

4.2.3 Combining Semantic and Syntactic Bidirectionalisation

It becomes apparent that both approaches have their pros and cons. Naturally, Voigtländer et al. (2010) propose a combination of the semantic and the syntactic bidirectionalisation. The combined approach uses each technique for its area of expertise: the semantic derivation for content updates and the syntactic derivation for shape-changing transformations. The authors categorise the two techniques; whereas the syntactic bidirectionalisation is used as black box, the semantic bidirectionalisation is similar to a glass box. That is, the semantic bidirectionalisation approach can be more powerful if we refactor the transformation in order to plug-in a syntactic technique; then, shape-changing transformations can be handled. The presented combination is general enough to allow any syntactical approach to be plugged-in. This generality is also the motivation to call it black-box: we do not know anything about the actual derivation, but that it yields the wanted results. The overall idea and implementation to plug-in a syntactical bidirectionalisation is discussed by Voigtländer et al. (2013) in more detail.

As a minor drawback, the range of *get* definitions covered by the combined approach is limited by two factors: only affine and treeless functions are allowed because of the usage of the syntactic bidirectionalisation, and we can only use polymorphic functions to use the semantic bidirectionalisation technique. Fortunately, the presented enhancements and extensions to semantic bidirectionalisation do consort well with the combined approach. That is, we can use the more general function *bffBy* in combination with specified observer functions for semantic bidirectionalisation or turn monomorphic functions into polymorphic ones with the monadic extension to gain a wider range of possible *get* functions. In the end, the combined approach performs never worse than one of the two approaches by themselves. The semantic bidirectionalisation on its own has difficulties in shape-changing update, but such updates are covered with the combined approach. The syntactic approach operates on specialised programs now, which can lead to better results.

In addition, the semantic bidirectionalisation uses free theorems also to prove consistency conditions. We discussed the syntactical bidirectionalisation, which formulates its derivation on the ground of the *GetPut* and *PutGet* law. In contrast, Voigtländer proves, with the help of free theorems, for each of his function definitions – *bff*, bff_{EQ} and bff_{ORD} – that they obey the lens laws. That is, instead of a correctness-by-construction approach, the laws are verified by hand.

5 Lens Implementations in Curry

In this chapter, we discuss two implementations of lenses in the functional logic programming language Curry. First, a combinatorial approach that focuses on the put function. Second, a put-based implementation using Curry's built-in search abilities. We also present *nondeterministic lenses*, a conservative extension that arises naturally in the setting of functional logic programming with Curry.

The first implementation is a combinatorial approach that is based on the Haskell library `putlenses` introduced in Section 4.1. The original library is built on monadic combinators that include a get and a put function, but the user only defines the put direction of a lens when using this library. For the Curry implementation, we adapt the underlying monadic approach by using Curry's built-in nondeterminism as the update strategy.

As the second implementation, we discuss a very simple library that does not use the combinatorial approach, but builds lenses only with the help of a put definition. In order to use lenses in the get direction, the library offers a general get function based on the given put definition and the lens laws. We discuss a potential get-based approach that follows the same idea as our implementation, argue about its disadvantages, and present related work.

In the context of nondeterministic lenses, we present adapted versions of the classical lens laws as well as a handful of lens definitions well-suited for a nondeterministic setting.

5.1 Combinatorial Lens Library

We discussed two different approaches on combinatorial frameworks for lenses in Section 4.1 to provide an insight of recent implementations. The combinatorial library for Curry is based on the approach of Fischer et al., who published the Haskell library `putlenses` as a result of their work on lenses. In the remaining section, we discuss the underlying implementation and give additional comments on its original counterpart. Additionally, we present some of the most important combinators as representatives. At last, we give some exemplary lens definitions to show the usage of the library and the definition of classical lens examples with the available combinators.

5.1.1 Motivation

The simplest representation of lenses is a pair of functions: one function for the get direction and one for the put direction. We can define such a data structure in Curry in three different ways: as a data type declaration with its own constructor, as a record type[1], or as a type synonym. For simplicity, we define lenses with a simple type synonym for a pair of get and put function.

$$\begin{aligned}
&\textbf{type}\ \mathit{Lens}_{Pair}\ s\ v = (s \to v, s \to v \to s)\\
&\mathit{put}_{Pair} :: \mathit{Lens}_{Pair}\ s\ v \to s \to v \to s\\
&\mathit{put}_{Pair} = \mathit{snd}\\
&\mathit{get}_{Pair} :: \mathit{Lens}_{Pair}\ s\ v \to s \to v\\
&\mathit{get}_{Pair} = \mathit{fst}
\end{aligned}$$

In addition to the definition of a lens type, we can access the get and put component of the pair with helper functions get_{Pair} and put_{Pair}, respectively. Next, we define a simple lens with a pair of arbitrary type as source that projects its first component in the get direction and updates the first component in the put direction. The update does not effect the second component.

[1]We will not pursue the usage of record types here, but seize the concept in Section 6.2.

$fst_{Pair} :: Lens_{Pair}\ (a, b)\ a$
$fst_{Pair} = (get', put')$
  where
    $get' :: (a, b) \to a$
    $get'\ (x, \_) = x$
    $put' :: (a, b) \to a \to (a, b)$
    $put'\ (\_, y)\ z = (z, y)$

The definition of this lens is straightforward, however, while we gain from simplicity, we lose in accuracy and maintainability. First of all, the given definition does not automatically form a well-behaved lens, because we did not consider the lens laws yet.[2] Second, if we modify one of the two functions, we have to make sure that the other one still harmonises with our modifications. That is, we have to check the lens definition manually with regard to consistency and validity. This circumstance is error-prone and requires a high maintenance effort, because we do not only have to check the lens property once after definition, but every time we modify one of the lens components. As an example, let us make a slight modification to the put function and define a lens *fstInc*, which, additionally, increments its first component in the put direction.

$fstInc :: Lens_{Pair}\ (Int, a)$
$fstInc = (get', put')$
  where
    $get'\ (x, \_) = x$
    $put'\ (\_, y)\ x = (x + 1, y)$

We have made slight changes to the put function, which only affect the first component. It would be interesting to check if the lens laws still hold after our modifications.

$get_{Pair}\ fstInc\ (put_{Pair}\ fstInc\ (1, 2)\ 13)$
> 14

$put_{Pair}\ fstInc\ (1, 2)\ (get_{Pair}\ fstInc\ (1, 2))$
$> (2, 2)$

[2]We know, however, from Section 3.2 that this exemplary lens definition is well-behaved.

The first expression checks the behaviour of the PutGet law, which does not behave as expected; the second expression behaves rather strangely with respect to the GetPut law. A very simple change breaks both laws, because we forgot to consider both functions. We can easily fix this bug by changing the right-hand side of the get direction to $x - 1$. However, this simple approach is not satisfactory at all. First, it does not *feel* bidirectional, because we still maintain two unidirectional functions in disguise. Second, we want to check the lens laws neither every time we define a new lens nor every time again we make changes to existing definitions.

5.1.2 Implementation

In order to provide a more user-friendly library with less maintenance effort, the library needs a rich set of combinators that are already well-behaved. Then, the user builds her own lenses with the help of these combinators without considering any laws, but only with her implementation in mind.

In the following, we present a reimplementation of the Haskell library `putlenses` in Curry. The original implementation provides a monadic interface to instantiate different update strategies. Due to the lack of type classes in Curry, we cannot use this approach, and we use the built-in nondeterminism instead. However, we can approximate the desired behaviour by instantiating the monad with a list when using the Haskell library.

```
data Lens s v = Lens (Maybe s → v → s) (s → Maybe v)

put :: Lens s v → Maybe s → v → s
put (Lens f _) = f

getM :: Lens s v → s → Maybe v
getM (Lens _ f) = f

get :: Lens s v → s → v
get (Lens _ f) s = case f s of
  Just v    → v
  Nothing → error "get': value is ‘Nothing‘"
```

In order to handle the problem of partial lens definitions, which we discussed in Section 3.2.3, the given representation of lenses wraps a *Maybe* data type around the view for the *get* function. That is, we can actually observe if the expression *get s* succeeds or fails. We could use *Set Functions*, introduced by Antoy and Hanus (2009), to identify defined and undefined values, but we adhere to the original implementation for simplicity reasons.[3]

Composition

Composition is the most valuable combinator, because it serves as a link between other primitive combinators to define more complex lenses. The composition function takes two lens functions that are well-suited and yields a specialised combination of these lenses. Two lenses are well-suited for composition if the view of the first lens and the source of the second lens have matching types.

```
(<.>) :: Lens s v → Lens v w → Lens s w
l1 <.> l2 = Lens putNew getNew
  where
    putNew ms@(Just s) w = put l1 ms (put l2 (getM l1 s) w)
    putNew Nothing     w = put l1 Nothing (put l2 Nothing w)
    getNew s             = getM l2 (get l1 s)
```

That is, in the get direction we can make two consecutive applications, i.e., the composition of two get functions is just function composition. Given two get functions – $get_{l1} :: s \to v$ and $get_{l2} :: v \to w$ – and a source of type s, we apply get_{l1} to yield a view of type v. Then, we apply get_{l2} to the result, which yields a value of type w. This application yields a new get function of type $get_{l1+l2} :: s \to w$. For the put direction, we have to play a bit more with the available functions and take a closer look at their type signatures. In addition to the get functions we discussed before, we have two put functions, $put_{l1} :: Maybe\ s \to v \to s$ and $put_{l2} :: Maybe\ v \to w \to v$, a source of type s, and an updated view of type w. The resulting put function is supposed to be of type $put_{l1+l2} :: Maybe\ s \to w \to s$. If there is no source available, i.e., the value of the source is *Nothing*, we can apply the two put functions consecutively. put_{l2} is applied to the source and the given view, and put_{l1} is applied to the resulting value as second argument. In case of an available source, we have to set the inner structure first with the given updated view. That is, we apply put_{l2} to the view of the given source and the updated view. Here, the usage of *Maybe* to wrap the result of a put function comes

[3]Set Functions might behave unexpectedly for partially applied functions.

in handy: we can easily make the distinction if a put function failed or not. As a last step, the source is updated with the resulting inner structure from the previous step using put_{l1}.

In the following, we present some primitive combinators more briefly than the previous description and emphasise examples of these combinators in action.

Basics: Identity and Filter

The identity combinator yields its source in the get direction and replaces its source with the given view for the put function. This lens is restricted to sources and views of the same type.

```
id :: Lens v v
id = Lens (\_v' → v') Just
```

A similar, but maybe more feasible, combinator filters its source and view, respectively, with regard to a specified predicate.

```
φ :: (v → Bool) → Lens v v
φ p = Lens get' put'
  where
    get' s    | p s       = Just s
              | otherwise = Nothing
    put' _ v  | p v       = v
              | otherwise = error "phi: predicate not fulfilled"
```

In particular, the put direction declines any updated view that does not fulfil the given predicate. That is, we demand the update on the view to be valid. The get function behaves as follows: if the given source fulfils the predicate, it yields that source; otherwise a valid view for the given source does not exist, and the function yields *Nothing*.

Products: Pairing and Unpairing

The second category of combinators covers products to build pairs and project components of pairs. The first lens builds a pair in the put direction by injecting a value to the left of the view, and it projects the second component of the source in the get direction.

```
addFst :: (Maybe (s1, v) → v → s1) → Lens (s1, v) v
addFst f = Lens put′ (Just ∘ snd)
  where
    put′ s v′ = (f s v′, v′)
```

The user constructs the injected value with a specified function, which takes a possible source and the updated view to yield a new first component. This specified function is, however, not restricted to a specific range in order to fulfil the lens laws. For instance, the following lens definition *fstAndInc* resets the first component of the source pair with an updated view and, simultaneous, increments the second component. We can define this lens by means of *addSnd*, the dual lens of *addFst* that behaves like *addFst*, but injects a second component and projects the first component, respectively.

```
fstAndInc :: Lens (Int, Int) Int
fstAndInc = addSnd inc
  where
    inc Nothing _ = error "fstAndInc: undefined source"
    inc (Just (_, v)) _ = v + 1
```

Unfortunately, this lens does not fulfil the GetPut law. We observe that the given implementation of *addFst* does not take any validation checks into account. In the original implementation, Fischer et al. ensure well-behavedness by using an auxiliary function *enforceGetPut* to resolve the irregularity. As a second option, they suggest to adjust the implementation of the get function to yield $\bot$ for every source that does not fulfil the GetPut law. For our implementation, we chose the latter solution, because the manual correction increases the range of valid lenses, whereas the elimination decreases the range and, thus, makes the lens less applicable.

The helper function *enforceGetPut* dynamically checks the behaviour of the given lens; the get function stays untouched, but the function applies the get function to the given source to check if the resulting value equals the current value.

```
enforceGetPut :: Lens a b → Lens a b
enforceGetPut (Lens putL getL) = Lens put′ getL
where
  put′ ms v = case ms of
    Just v′ | getL v′ ≡ Just v → v′
    _                          → putL
```

If the updated view is equal to the current view, we do not make any further changes and yield the source; otherwise we apply the put function as usual. That is, *enforeGetPut* yields the given source for an unchanged view according to the Get-Put law, hence, forces the lens to be well-behaved. We rewrite the definition of *addFst* as follows.

```
addFst :: (Maybe (s1, v) → v → s1) → Lens (s1, v) v
addFst f = enforceGetPut (Lens put′ (Just ∘ snd))
  where
    put′ s v′ = (f s v′, v′)
```

As counterpart to *addFst* and *addSnd*, we provide *remFst* and *remSnd* to destruct the view pair by discarding the first and second component, respectively.

```
remFst :: (v → v1) → Lens v (v1, v)
remFst f = Lens put′ (λs → Just (f s, s))
where
  put′ _ (v1, v)
    | f v ≡ v1 = v
    | otherwise = error "remFst: first and second value do not match"
```

For the definition of *remFst*, the given function creates the new first component in the get direction, and the first component is discarded in the definition of the put function. Additionally, we have to assure that the user-specified function applied to the second component of the source yields the same value as the first value of the source. Without this correction, the given lens definition would not fulfil the PutGet law.

Sums: Either Left or Right

In order to handle sum types like *Either*, we provide a lens that distinguishes between a *Left* and a *Right* value. The lens *injL* injects the given updated view as a left value and ignores the source; its counterpart *injR* injects a right value. In the get direction, the function ignores a given left and right value, respectively.

```
injL :: Lens (Either v1 v2) v1
injL = Lens (const ∘ Left) get′
  where
    get′ (Left v) = Just v
    get′ (Right _) = Nothing
injR :: Lens (Either v1 v2) v2
injR = Lens (\_v → Right v) get′
  where
    get′ (Left _) = Nothing
    get′ (Right v) = Just v
```

Unlike *addFst* and *remFst*, the given lens definition and its dual do not need any dynamic checks to ensure well-behavedness. These kind of lenses do not seem very feasible at first sight, however, we will see some practical lens definitions in the next section.

5.1.3 Usage and Examples

Although we gave the fundamental combinators of the library, we did not dive into programming our own lenses so far. When defining lenses, the user has to build his lens by composing the combinators of the library.

Example I: Starting with *fst*

As a first simple example, we define our running example, *fst*, by the means of *addSnd*.

fst_{comb} :: Lens (a, b) a
fst_{comb} = addSnd (λs _ → maybe failed snd s)

If there is no source available, we cannot do anything meaningful without losing generality – we could only yield the view instead. However, we do not want to restrict the types of source and view to match, thus, the lens fails if no source is available. Otherwise, we simply select the second component of the given pair and add it to the updated view to form a pair again. The usage of *addSnd* indicates that we inject the value as second component, whereas the second component is reserved for the updated view. Naturally, it follows that we can define *snd* as a lens as well: instead of *addSnd* and *snd*, we use their duals *addFst* and *fst*.

What about the get direction? We have only discussed the update strategy of the lens, i.e., the put direction. First of all, let us test the behaviour of fst_{comb}.

```
> put fst_comb (Just (1, "test")) 13
(13, "test")
> get fst_comb (13, "test")
13
```

The get as well as the put direction behaves as intended. We can observe that we do not need to take the get direction into account when we define a new lens. The library encourages the user to define his lenses by means of the put direction only. As we discussed in Section 4.1.2, it may be more conventional and intuitive. Nevertheless, the put functions that we defined for the library have a unique corresponding get function, because all put functions comply with the requirements stated by Fischer et al.

Example II: Lenses for Built-in Data Types

The library consists of several combinators that work on sums and products; but what about built-in data types or user-defined structures? We use the idea that every algebraic data type can be expressed by sums and products. For example, we can take a look at the *Maybe* data type in Curry, which is a classical representative of a sum type.

```
data Maybe a = Nothing | Just a
```

The *Maybe* data type has one constructor *Nothing* that represents a failure value, and the *Just* constructor for valid values. We can easily rewrite this data type and use sum types, i.e., *Either*, instead.

```
type Maybe a = Either () a
nothing :: Maybe a
nothing = Left ()

just :: a → Maybe a
just = Right
```

A failure value like *Nothing* can be represented with *Left* (), because () is the only value of the Unit type; any other value can be represented with *Right* instead of *Just*.

As a second example, we discuss how to use the available combinators to build lenses for lists. First of all, we need to think about a representation for lists by means of sum or product types; for this, we recall the definition of lists in Curry.

```
data [a] = [] | a : [a]
```

Similar to the *Maybe* data type, we have one value that stands for itself and does not hold any value: the empty list. In addition, the second constructor adds a new element to the head of a list, that is, the binary constructor can be represented as a product, i.e., with (,). With this general structure in mind, we can represent lists as combination of *Either* and (,) as follows.

```
type List a = Either () (a, [a])
empty :: List a
empty = Left ()
cons :: a → [a] → List a
cons x xs = Right (x, xs)
```

In this representation, the list $[1,2,3,4]$ is rewritten as *Right* $(1,[2,3,4])$, and the empty list, [ ], corresponds to *Left* ().

Every algebraic data type has a set of selectors to work with. In the following, we define lenses equivalent to *head* and *tail* in the get direction. Up to this point, we have withheld the information about another special combinator that builds an isomorphism between two data representations.

```
isoLens :: (a → b) → (b → a) → Lens b a
isoLens f g = Lens (\_ v → f v) (λs → Just (g s))
```

The *isoLens* forms an isomorphism between two types a and b, where b is the starting value, and a takes the role of the internal structure. The functions takes two functions for transformations: from a to b and vice versa. In the get direction, we transform the source into an internal structure, and we convert the updated structure back again in the put direction. In order to provide selectors for lists, we have to define such an isomorphism between the list data type and the rewritten structure based on sums and products.

```
inList :: Lens [a] (List a)
inList = isoLens inn out
where
  inn = either (λ() → [ ]) (λ(x, xs) → x : xs)
  out xs = case xs of
             [ ] → empty
             y : ys → cons y ys
```

The transformation functions follow naturally from the definition of *List a*, as above. We can eliminate the wrapping *Either* by using *injR* and *injL* that unwrap the *Left* and *Right* constructor, respectively, and yielding the containing value.

```
cons :: Lens [a] (a, [a])
cons = inList < . > injR
```

That is, for an exemplary list $[1, 2, 3, 4]$, we can apply our lens *cons* to transform the list into a pair of head element and remaining list or to replace the given list by a new one.

```
> get′ cons [1, 2, 3, 4]
(1, [2, 3, 4])
> put′ cons (Just [1, 2, 3, 4]) (13, [ ])
[13]
> get′ cons [ ]
"get’: value is ‘Nothing‘"
```

Unfortunately, we cannot transform the empty list into a representation that consists only of products for two reasons. First, the used combinator *injR* only selects *Right* values, and the empty list is represented as *Left* (). Second, product types are not suitable to model failure values like the empty list. The usage of *injL* instead of *injR* is not feasible either: *injL* can only select *Left* values and fails otherwise. However, this minor disadvantage does not affect the combinators that we want to define. The functions *head* and *tail* are partial functions that only operate on non-empty lists; we do not need to take the empty list into consideration to define our lenses.

The actual definition of *changeHead* and *changeTail*[4] is rather simple: the *cons* combinator splits the list into head and tail. Thus, we only need to choose between the first and second component as a last step.

```
changeHead :: Lens [a] a
changeHead = cons <.> keepSnd

keepSnd :: Lens (v, s1) v
keepSnd = addSnd (λs v′ → maybe (const failed) snd s)

changeTail :: Lens [a] [a]
changeTail = cons <.> keepFst

keepFst :: Lens (v, s1) v
keepFst = addFst (λs v′ → maybe (const failed) fst s)
```

Obviously, *changeHead* replaces the head of the list with a new element and leaves the tail untouched with *keepSnd* and vice versa for *changeTail*. In the corresponding get direction, we can access the head and tail of the list, respectively. The definition of the auxiliary functions *keepSnd* and *keepFst* is analogue to the definition of fst_{comb} given above.

```
> get′ changeHead [1, 2, 3, 4]
1
> put′ changeHead [1, 2, 3, 4] 13
[13, 2, 3, 4]
> get′ changeTail [1, 2, 3, 4]
[2, 3, 4]
> put′ changeTail [1, 2, 3, 4] [13, 14, 15]
[1, 13, 14, 15]
```

[4]The names conform to the functionality of their put function.

Example III: User-defined Data Types

Users can follow the same approach to define lenses for self-defined data types. As an example, we transform a data type with one constructor and several arguments into a sum. Consider the simple data type *Date* with one constructor and two arguments corresponding to a month and a day, respectively.

```
type Month = Int
type Day = Int
data Date = Date Month Day
```

We can easily transform this data type into a pair $(Month, Day)$ with the lens

```
date :: Lens Date (Month, Day)
date = isoLens inn out
  where
    in (m, d) = Date m d
    out (Date m d) = (m, d)
```

and provide selectors – *day* and *month* – to access the values in the get direction and replace them with new values in the put direction.

```
month :: Lens Date Month
month = dateLens < . > keepSnd
day :: Lens Date Day
day = dateLens < . > keepFst
> put′ month (Date 12 10) 10
Date 10 10
> get′ day (Date 11 18)
18
```

We can observe from this simple example that, in addition to simple accessors, we can add checks in order to restrict the range of valid values. In the case of a data structure for dates, valid values range between *Date* 1 1 and *Date* 12 31 with a lot of exceptions in between. We can modify the transformation functions of *date* easily to shrink the range of valid dates.

```
$date_{advance}$ :: $Lens\ Date\ (Month, Day)$
$date_{advance} = isoLens\ inn\ out$
  **where**
    **in** $(m, d) \mid check\ m\ d = Date\ m\ d$
    $out\ (Date\ m\ d) \mid check\ m\ d = (m, d)$
    $check\ m\ d = 0 < m \wedge m < 13 \wedge 0 < d \wedge d < 32$
```

Our modification still tolerates some invalide dates, e.g. *Date* 2 31, but for simplicity reasons we leave further adjustments to the reader.

This schema can be used for all kinds of algebraic data types; we provide some more examples in Appendix A.

5.2 Put-Lenses Library

In contradiction to most of the work in the area of bidirectional programming, Fischer et al. set their focus on defining the put direction instead of a get function. With this approach, they want to avoid a whole range of functions that are not suited for the get direction because of the ambiguity of their corresponding put function. The remainder of this section deals with a very simple implementation of a lens library in Curry that sets its focus on the put functions as well. We also give some examples for better comprehension. In order to motivate our approach, we start with a discussion about the current state of the art and why we decided to focus on the put function anyway.

5.2.1 Getting in the Way of Productive Results

As we have seen so far, most libraries and languages tackling the topic of bidirectional programming define their language by means of the get direction. This view on bidirectional programming arises from the way programmers are used to define functions. Programmers are more familiar with defining a get function – a function that selects information from a structure – rather than thinking about an appropriate update strategy. Therefore, let us take a look at a very simple implementation of a get-based approach.

$\textbf{type}\ \mathit{Lens}_{get}\ s\ v = s \rightarrow v$

$\mathit{get}_{get} :: \mathit{Lens}_{get}\ s\ v \rightarrow s \rightarrow v$
$\mathit{get}_{get} = \mathit{id}$

We define a type synonym for lenses that is equivalent to the signature of a get function, i.e., we represent lenses as their get function. This definition leads to a straightforward implementation of a get function for lenses: we merely apply the given lens to a given source. However, this implementation of lenses lacks information for the put direction. As a quick reminder: we want to define a function put_{get} that updates a given source with a modified view with respect to the given lens. The given lens is a function that projects some view from a given source. Thus, in contradiction to our previous approach of a combinatorial language, here, the given lens has no further information about the update strategy. All we know about the given lens is that it needs to obey certain round-tripping laws to be well-behaved: GetPut and PutGet. That is, we can define the put definition by means of the get definition with respect to the PutGet law.

$\mathit{put}_{get} :: \mathit{Lens}_{get}\ s\ v \rightarrow s \rightarrow v \rightarrow s$
$\mathit{put}_{get}\ \mathit{lens}\ s\ v \mid \mathit{get}_{get}\ s' \equiv v = s'$
$\quad \textbf{where}\ s'\ \mathit{free}$

In order to see this implementation in action, we take a look at our running example in the setting of get-based lenses: we define *fst* by means of get-based lenses.

$\mathit{fst}_{get} :: \mathit{Lens}_{get}\ (a, b)\ a$
$\mathit{fst}_{get}\ (x, \_) = x$

This lens definition is very simple and constitutes a very familiar setting for the programmer: we want to project the first component of a given pair. Thus, we ignore the second component and simply yield the first component. If we run exemplary function calls of get_{get} and put_{get}, we get more or less satisfactory results.

$> \mathit{get}_{get}\ \mathit{fst}_{get}\ (1, 42)$
1

$> \mathit{put}_{get}\ \mathit{fst}_{get}\ (1, 42)\ 3$
$(3, \_\mathit{x1})$

In the get direction, everything works as expected. Unfortunately, the put function yields a free variable as its second component. This result leads to the realisation that the get-based implementation is rather simple, not to say, too simple. If we use fst_{get} in the put direction, we loose the additional information of the source pair. The problem arises from the definition of put_{get}: we ignore the information about the original source, and examine the updated view instead. After all, we do not have any information about the second component of the given pair, because the definition of fst_{get} takes only the first component into consideration. In many cases, the discarded original source leads to an ambiguous put function. In particular, the above implementation is only applicable in case of injective get function: an injective function represents a one-to-one mapping, it preserves distinctness. That is, there are no two elements of the source domain that map to the same element in the codomain. In the case of our example lens fst_{get}, there are several source pairs that yield the same result; in fact, every pair with the same first component yields the same view.

$> \mathit{get}_{get}\ \mathit{fst}_{get}\ (1, 42)$
1
$> \mathit{get}_{get}\ \mathit{fst}_{get}\ (1,$ `"Hello World"`$)$
1
$> \mathit{get}_{get}\ \mathit{fst}_{get}\ (1, \mathit{True})$
1

The function fst_{get} is not injective, which leads to an ambiguity issue and, hence, to a non-constructive put function.

So, how do get-based lens definitions look like that are applicable to our implementation approach? As a main requirement, we need to define injective get functions. The first observation, which we made in the context of the previous two lens definitions, is that we cannot ignore any part of the given source without losing information of the corresponding put function. Let us take a look at an injective get function that is defined on pairs, with *Int*s as its first component, and an arbitrary type as its second component.

$\mathit{incFst}_{get} :: \mathit{LensGet}\ (\mathit{Int}, b)\ (\mathit{Int}, b)$
$\mathit{incFst}_{get}\ (x, y) = (x + 1, y)$

In the definition of the lens $incFst_{get}$, we increment the first component and do not touch the second one. In contrast to the previous lens definition, we do not ignore the second component; it is still a part of the resulting view. That is, we have a lens that maps pairs to pairs.

$> get\ incFst_{get}$ (1, `"Hello World"`)
(2,'' *Hello World''*)
$> put\ incFst_{get}$ (1, `"Hello World"`) (2, `"Haskell B. Curry"`)
(1, `"Haskell B. Curry"`)

The behaviour of the lens is rather simple, but in this case it is injective. The aim of this definition is to show that most get functions are not of real use for defining a lens library for two reasons. First, they may be non-injective and, thus, do not have a uniquely defined corresponding put function. Second, if the definition is injective, its behaviour is rather simple and not very useful. We presented typical lenses like $head_{get}$ and $head_{fst}$, which are still very simple, but already do not comply with the injectivity requirement.

In the end, the implementation of such a simple get-based is not promising; as a short excursion, we discuss related work on get-based lens implementation and how they approach the problem of ambiguity.

Excursion: Related Work on Get-Based Lenses

Most approaches try to build their bidirectional language with respect to a specific application area, e.g., XML data, strings, or databases. However, in the following, we discuss some approximations for a more general approach.

There are several existing ideas to overcome these limitations regarding ambiguity. One of the most popular idea is to choose the best put function based on similarities and differences between the original source and its potential update. The initial concept was proposed by Meertens (1998), whose framework of constraint maintainers for user interaction is sometimes called a pioneer work in the topic of bidirectional transformations and lenses. In his work, he states that UI-transformations are supposed to be as minimal as possible in respect to the given constraint. This approach aims to be user-friendly: the results of the transformations are more comprehensible the more they are related to the initial situation.

More recently, Diskin et al. (2010, 2011a,b) follow this approach in their work about *delta lenses*; they cover asymmetric as well as symmetric lenses. The general idea is to distinguish between the computed delta and the effectively update propagation; the get as well as the put function take the computed delta into consideration. The computed delta helps to choose the best update strategy. Therefore, delta lenses consist of a get and put function with a computed delta between original and updated source as an additional argument. Diskin et al. develop a framework on the grounds of algebraic theory. This idea of delta lenses is a conservative extension to the original lens framework; that is, the framework can reproduce the behaviour of ordinary lenses.

Additionally, Barbosa et al. (2010) put the theory into practice: their development of a new core language of matching lenses for strings can be seen as enhancement of their domain-specific language Boomerang (see Section 4.1). The framework parametrises lenses with respect to heuristics in order to calculate alignments. So-called *chunks* are used to label each element of the source and to recognise these elements, when they are modified with an updated view. As a drawback, the use of such one-to-one mappings as alignment strategy leads to a positional alignment only. That is, every element of the source needs to have a corresponding element in the view and vice versa; the focus lies on the data and the original shape is ignored during alignment.

At this point, the work of Hofmann et al. (2012) and Pacheco et al. (2012, 2013a) comes into play. The former approach develops a theory of *edit lenses*; the main difference to basic lenses is their focus on changes of structures similar to the idea behind delta lenses. Edit lenses establish the connection between original and updated source; an approach that does not allow any guessing, but has a strict rule to apply the resulting alignment. Hofmann et al. describe these lenses with a standard mathematical notion of monoids and monoid actions, where the former corresponds to the description of edits and the latter describes the actual application of such edits to the given structure. Whereas Diskin et al. merely propose a theoretical framework for descriptions of changes, Hofmann et al. introduce a more mature approach with additional combinators, e.g., composition, sums, products etc., that give rise to brighter area of application. Most recently, Wagner (2014) finished his dissertation about edit lenses in a symmetric setting that gives rise to the latest developments in that area.

Pacheco et al. identified that positional updates are only reasonable for data alignment, but shape alignment needs to be considered separately. Their approach tackles the problem of positional alignment and introduces an explicit separation of shape and data. In their paper, they describe a point-free delta lens language in a dependent type setting, which is based on their early work of point-free lenses (2010). They distinguish between horizontal and vertical deltas: the former describes an update, where source and view values are of different types; the latter is a special case that describes updates for values of the same type. Pacheco et al. criticise the lack of shape alignments in related work on lenses. Recent approaches focus on aligning the data of source and view, but fail to establish a convenient mapping of both shapes. This positional alignment leads to less predictable updates regarding insertion and deletion of elements, which either are not detected or effect only the end positions of the underlying structure, e.g. new elements are inserted at the end of a list. Thus, the main effort of Pacheco and colleagues' work are recursion patterns for horizontal delta lenses, which introduce shape alignments for combinators like fold and unfold.

In this thesis, we do not investigate additional measurement techniques or applicable restrictions to avoid ambiguous put functions. Instead, we focus on the put direction of lens definitions and search for an applicable get direction.

5.2.2 Putting it Straight

Fischer et al. (2012) are the first to push lens definitions by means of the put function instead in order to eschew the unavoidable ambiguity of the get function. Curry's built-in search capabilities form a fruitful ground for a bidirectional library that focuses on one direction and calculates the corresponding function for the other direction.

The pivot of the library is a very simple definition to represent lenses by means of the put direction as well as its selectors put_{put} and get_{put} to use a given lens in the put and get direction, respectively.

type $Lens_{put}\ s\ v = s \to v \to s$

$put_{put} :: Lens_{put}\ s\ v \to s \to v \to s$
$put_{put} = id$

$get_{put} :: Lens_{put}\ s\ v \to s \to v$
$get_{put}\ lens\ s \mid put_{put}\ lens\ s\ v \equiv s = v$
  where v *free*

The idea of the definition of get_{put} is symmetric to put_{get} from the get-based approach above. In this case, we use the PutGet law to define the requirements for an appropriate get function for a given lens definition. That is, Curry searches for an updated view that yields the given source, when we call the lens in the put direction with that source and the updated view.

As usual, we define a lens to project the first component of a pair. Actually, in the setting of a put-based lens library, we want to define a lens to set the first component of a pair.

$fst_{put} :: Lens_{put}\ (a, b)\ a$
$fst_{put}\ (\_, y)\ z = (z, y)$

However, this lens definition is supposed to be equivalent to other examples of *fst* in the setting of lenses. Since we have already defined the update strategy in the put direction, we have to check if the get direction actually yields the first component of the given pair.

$> put_{put}\ fst_{put}\ (1, 2)\ 13$
$(13, 2)$

$> get_{put}\ fst_{put}\ (13, 2)$
13

Clearly, the get direction works as expected – but how exactly does Curry evaluate this expression? In order to further examine the question, we take a closer look at the evaluation steps of the expression $get_{put}\ fst_{put}\ (13, 2)$.

$get_{put}\ fst_{put}\ (13, 2)$
$\equiv$ { (1) evaluate get_{put}; replace guard with if-then-else expression }
if $put_{put}\ fst_{put}\ (13, 2)\ v' \equiv (13, 2)$ **then** v' **else** *failed*
  where v' *free*
$\equiv$ { (2) evaluate (sub put put) to *id* and apply *id* it the given arguments }
if $fst_{put}\ (13, 2)\ v' \equiv (13, 2)$ **then** v' **else** *failed*
  where v' *free*
$\equiv$ { (3) evaluate $fst_{put}\ (13, 2)\ v'$ to $(v', 2)$; v' is still not bound yet }
if $(v', 2) \equiv (13, 2)$ **then** v' **else** *failed*
  where v' *free*
$\equiv$ { (4) in order to evaluate $(\equiv)$, the free variable is bound to 13 }

$$
\begin{array}{ll}
 & \{ \text{ replace each occurrence of } v' \text{ with } 13 \ \} \\
 & \mathbf{if}\ (13,2) \equiv (13,2)\ \mathbf{then}\ 13\ \mathbf{else}\ \mathit{failed} \\
\equiv & \{ \text{ (5) evaluate the if-condition to } \mathit{True} \ \} \\
 & \mathbf{if}\ \mathit{True}\ \mathbf{then}\ 13\ \mathbf{else}\ \mathit{failed} \\
\equiv & \{ \text{ (6) } \} \\
 & 13
\end{array}
$$

The definition of get_{put} introduces a free variable that is bound in the process and represents the value that the function yields as a result. The most important evaluation takes place at step 4, where the operator ($\equiv$) forces its left argument to be evaluated to reduce the whole conditional expression to a boolean value. The left argument consists of the free variable, which is then bound to the appropriate value that evaluates the condition to *True*. In this case, the expression $(v', 2) \equiv (13, 2)$ evaluates to *True* if both expressions can be reduced to the same value. The second components of the two pairs are already equivalent, thus, the free variable v' needs to be bound to the first component of the right pair, i.e., 13. More precisely, Curry generates a series of numbers that fail the condition before the variable is finally bound to 13 – we discuss Curry's built-in search capabilities in further detail in Section 7.2.1.

The most important function the library contains is the composition operator for lenses. As before, we define $(<\,.\,>)_{put}$ as composition of two lenses $l1 :: Lens_{put}\ a\ b$ and $l2 :: Lens_{put}\ b\ c$ to gain a resulting lens of type $Lens_{put}\ a\ c$.

$$
\begin{array}{l}
(<\,.\,>)_{put} :: Lens_{put}\ a\ b \to Lens_{put}\ b\ c \to Lens_{put}\ a\ c \\
(<\,.\,>)_{put}\ lAB\ lBC\ sA\ vC = put_{put}\ lAB\ sA\ sB \\
\quad \mathbf{where}\ sB = put_{put}\ lBC\ (get_{put}\ lAB\ sA)\ vC
\end{array}
$$

In order to see the composition operator in action, we need a second lens to connect two lenses in series. We first define an algebraic data type for a contact, like in an address book. The contact consists of an address and information about the contact's name, i.e., its first and last name.

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Name} \quad\ \ = (\mathit{String}, \mathit{String}) \\
\mathbf{type}\ \mathit{Address} = \mathit{String} \\
\mathbf{data}\ \mathit{Contact} = \mathit{Contact}\ \mathit{Name}\ \mathit{Address}
\end{array}
$$

$name :: Lens_{put}\ Contact\ Name$
$name\ (Contact\ \_\ address)\ newName = Contact\ newName\ address$
$address :: Lens_{put}\ Contact\ Address$
$address\ (Contact\ name\ \_)\ newAddress = Contact\ name\ newAddress$

Additionally, we have two lenses to operate on the algebraic type *Contact*: one selector to change and project the name, the other one for the address. The first lens, *name*, yields the name of a contact, which is represented as a pair. We define a lens that operates directly on the first name of a contact by composing *name* with fst_{put}.

$firstName :: Lens_{put}\ Contact\ String$
$firstName = nameLens <.> fst_{put}$

We define exemplary values of type *Contact* and use both selectors to project as well as to change the name and address.

```
contact1 = Contact ("Bob","Dylan") "Folkstreet 13"
contact2 = Contact name1 "Howard Lane 21"
name1 = ("Haskell","Curry")
```

```
> get_put name contact2
("Haskell","Curry")
> put_put name contact1 name1
Contact ("Haskell","Curry") "Folkstreet 13"
> get_put address contact1
"Folkstreet 13"
> put_put address contact1 "Folk Street 39"
Contact ("Bob","Dylan") "Folkstreet 39"
```

Furthermore, we apply the composed lens to *contact2* to change only the first name and set it to "Haskell B.". We can also project the first name of *contact1* with the composed lens.

```
> put_put firstName contact2 "Haskell B."
Contact ("Haskell B.","Curry") "Howard Lane 21"
> get_put firstName contact1
"Bob"
```

5.2.3 What About Well-Behavedness?

Up to now, we have not discussed any requirements for the definition of lenses in order to guarantee well-behavedness. The attentive reader may remember the two important laws: GetPut and PutGet. As we already use the underlying equation of the GetPut law, the definition of put-based lenses in our library guarantees to fulfil the GetPut law. Unfortunately, we cannot make any guarantees in case of the PutGet law. Instead, we provide an additional library to test properties like PutGet and GetPut. In the case of put-based lenses, we can express the requirements for well-behavedness with the put function only. We have already discussed this modification in Section 4.1.2 and introduced the laws *PutInj* and *PutTwice*.

The implementation of our testing library is built on an old version of EasyCheck[5]. EasyCheck is a lightweight library for specification-based testing in Curry implemented by Christiansen and Fischer (2008).[6] In a nutshell, the library provides functions to define specifications and tests theses specifications by enumerating possible values that obey the given type dependencies. In case of an error, the library provides the tested value that contravenes the given specification as well as the false result.

We had to make some adjustments to the implementation, because the latest version of the EasyCheck was written for KiCS – a predecessor of the currently used and maintained KiCS2 compiler. These adjustments cover mostly the renaming of used libraries and reimplementing modules that are not part of the KiCS2 contribution anymore.

We provide a library `LensCheck` with a set of testing functions to check several properties of user-defined lenses. First of all, the library consists of functions *checkGetPut* and *checkPutGet* to test the traditional two round-tripping rules and an additional testing function *checkPutPut* for the PutPut law. For the purpose of put-based properties – as proposed by Fischer et al. – the library provides testing functions *checkPutDet* and *checkPutStab*. In addition, all five properties can be tested in the context of lists with a specifically defined version, e.g., *checkListGetPut* and *checkListPutDet*.[7]

[5]`http://www-ps.informatik.uni-kiel.de/currywiki/tools/easycheck`

[6]The implementation of EasyCheck is highly motivated by the work of Hughes and Claessen (2000), who introduced QuickCheck. QuickCheck is a testing library for Haskell, which has achieved a very good reputation in the Haskell community and is still excessively used.

[7]We include a function for this special case, because it is explicitly recommended in the paper of Fischer et al. to use an additional function to exclude the empty list as value.

Unfortunately, our lens library does not prevent the user to define inaccurate lenses. The only guarantee the library gives is in the context of PutGet. Additional properties have to be checked manually by the user in order to prevent misbehaved lens definitions.

5.3 Nondeterministic Lenses

Due to our choice to use Curry as programming language, we want to investigate the applicability of lenses in a nondeterministic context. Let us use the setting of contacts from the previous section as example. Instead of changing the underlying data structure to a list of addresses, we model the possibility of several addresses nondeterministically.

```
contactSample :: Contact
contactSample = Contact ("John", "Sample") address1

address1 :: Address
address1 = "Any Street 213"
address1 = "Working Avenue 17"
```

That is, we have two rules for the definition of John's address: one rule for his home address and one to contact him at work. If we execute *contactSample* in the interactive environment of KiCS2, we get the following two results.

```
> contactSample
Contact ("John", "Sample") "Any Street 213"
Contact ("John", "Sample") "Working Avenue 17"
```

As a consequence, we also get two results if we use the lens *address* to project the address of *contactSample*. However, the put direction does not behave nondeterministically, because we ignore the current address. Note, we use *get* and *put* as selector and update function of lenses, respectively, instead of the subscripted variant of the previous section.

```
> get address person1
"Any Street 213"
"Working Avenue 17"

> put address person1 "Sesame Street 123"
Contact ("John", "Sample") "Sesame Street 123"
```

The order of the nondeterministic results depends on the order of the definition's rules. The the first rule of *address1* defines the private and the second rule the working address, thus, leading to the order when evaluating *contactSample*.

Note, *address* is not a lens definition with a nondeterministic get function, but the nondeterministic behaviour is introduced by the definition of *address1*. We use *address1* in our exemplary expressions to gain nondeterministic results in the get direction, but deterministic values for the put function.[8]

As an example for a nondeterministic put function, we modify our definition of $fstAndInc_{put}$ given in Section 5.1.2. The modified version increments the second component or yields the old value; the first component is updated by the given new value as before.

$$\mathit{fstAndInc}_{put} :: \mathit{Lens}\ (a, \mathit{Int})\ a$$
$$\mathit{fstAndInc}_{put}\ (\_, i)\ y = (y, i + 1\ ?\ i)$$

The following exemplary expressions show the behaviour of $fstAndInc_{put}$. Whereas the get direction of our lens definition is deterministic, the put function behaves nondeterministically.

```
> get fstAndInc_put ("Super Mario", 2)
"Super Mario"
> put fstAndInc_put ("Super Mario", 3) "Luigi"
("Luigi", 4)
("Luigi", 3)
```

5.3.1 Extended Laws

The attentive reader might wonder why we use a lens definition that we explicitly exposed as incorrect before. Consequently, the question arises of how this nondeterministic behaviour interacts with well-behavedness and the lens laws, respectively. Therefore, we discuss the definition of laws for nondeterministic lenses.

As a reminder, we recapitulate the lens laws.

[8]However, we give a lens with a nondeterministic get function in the next subsection.

$$get\ (put\ s\ v) = v \qquad \text{(PutGet)}$$
$$put\ s\ (get\ s) = s \qquad \text{(GetPut)}$$
$$put\ (put\ s\ v)\ v' = put\ s\ v' \qquad \text{(PutPut)}$$

The main idea of the adapted laws is to change the semantics of the equational operator instead of modifying the equation itself. In a nondeterministic setting, expressions evaluate to a set of results rather than one result as in a functional or imperative context. This notion of expressions gives rise to a modified notion of the equivalence of two expressions. In the context of nondeterministic lenses, we interpret the equational operator in terms of sets. In particular, we want the smaller set to be a subset of the greater one. We can specify the lens laws in a nondeterministic context with the following equations, which are also applicable for singleton sets, i.e., for deterministic values.

$$v \subseteq get\ (put\ s\ v) \qquad \text{(PutGet-Nondet)}$$
$$s \subseteq put\ s\ (get\ s) \qquad \text{(GetPut-Nondet)}$$

For the PutGet law, we demand the value that we put into the source to be one of the elements that we can get out of the modified source. We can state a similar requirement for GetPut: if we get a value out of a source and put it back again, the resulting set of sources should at least contain the initial source.

We can define a similar equation for the PutPut law, but do not check the law for our examples above. Instead, we postpone a detailed example for the PutPut law to the next subsection.

$$put\ s\ v' \subseteq put\ (put\ s\ v)\ v' \qquad \text{(PutPut-Nondet)}$$

With the first two equations in mind, we can check the lens definitions given above. We start with *address*, the lens definition with a nondeterministic get direction. In case of the GetPut law, the modified view of the put action is a nondeterministic value. Thus, the put action yields a nondeterministic result as well. Fortunately, the resulting set is a superset of the modified view.

put address person1 (*get address person1*)
$\equiv$
{*Contact* (`"John."`, `"Sample"`) `"Working Avenue 17"`
, *Contact* (`"John."`, `"Sample"`) `"Any Street 213"`}
$\supseteq$
{*Contact* (`"John."`, `"Sample"`) `"Working Avenue 17"`
, *Contact* (`"John."`, `"Sample"`) `"Any Street 213"`}
$\equiv$
person1

For a nondeterministic address entry, the PutGet law holds even with the old semantics, because the put direction yields a deterministic result regardless of the address's value. The new semantics holds trivially, because we work only on singletons.

get address (*put address* `"Sesame Street 123"`)
$\equiv$
{`"Sesame Street 123"`}
$\supseteq$
{`"Sesame Street 123"`}

Next, we take a look at our example with a nondeterministic put function. In case of the GetPut law, the inner function call of get is deterministic and yields the first component of the given pair. Then, the put function yields two results nondeterministically: one pair with the original second component and one pair with an incremented second component. Remember,we increment the second component on every update in the deterministic version of *fstInc*; thus, leading to a violation of te GetPut law. However, the nondeterministic version is well-behaved with respect to the modified lens laws.

put fstInc (`"Super Mario"`, 2) (*get fstInc* (`"Super Mario"`, 2))
$\equiv$
{(`"Super Mario"`, 3), (`"Super Mario"`, 2)}
$\supseteq$
(`"Super Mario"`, 2)

At last, we need to check the PutGet law. The get function is applied to the resulting set of the put action leading to a nondeterministic result. The put action yields two results with the same first component, thus, leading to the same two results in the get direction.[9] Obviously, the modified view is a subset of the resulting set, which contains the modified view twice.

```
get fstInc (put fstInc ("Super Mario", 2) "Luigi")
≡
{"Luigi", "Luigi"}
⊇
{"Luigi"}
```

5.3.2 Productive Nondeterministic Lenses

We motivate the usage of nondeterministic lenses with three additional examples. Furthermore, we want test the PutPut law for one of these examples.

In the previous subsection, we defined lenses with a nondeterministic put and get function, respectively. Our next example behaves nondeterministically in both directions.

```
replace :: Lens [a] a
replace (x : xs) y = y : xs ? x : replace xs y
```

We define *replace* to nondeterministically replace an element of a given list with a new value. This example is based on the implementation of *insert*, which is most commonly used to define a nondeterministic function to produce all permutations of a list. However, *insert* is not an applicable example, because the get function fails for every input value. We take a quick look at the implementation to figure out why *insert* is not suitable as a lens.

```
insert :: Lens [a] a
insert [] y       = [y]
insert (x : xs) y = y : x : xs ? x : putInsert x ys
```

We nondeterministically insert a value at every possible position of the list. However, the insertion of an element becomes a problem when we search for a corresponding get

[9]In the notion of sets, we can eliminate duplicates, but the elimination is not required to fulfil the equations of the current example.

function. For the get direction, we have to evaluate the following expression to find a corresponding result.

$insert\ s\ v \equiv s$ **where** v *free*

That is, we search for an element to insert to the list, such that we get the original list. In order to get the original list, we cannot change the given list at all. Thus, there is no element to insert and no suitable result for this expression to be true.

For our example, we use *replace* instead of *insert* to avoid this problem. As mentioned in the beginning, *replace* is nondeterministic in the get as well as in the put direction. We nondeterministically get an element of the list if we use *replace* in the get direction; the put direction replaces a given element nondeterministically. The nondeterministic behaviour leads to the following results for exemplary expressions.

```
> get replace [1 .. 3]
1
2
3
> put replace [1 .. 3] 5
[5, 2, 3]
[1, 5, 3]
[1, 2, 5]
```

If we check the PutPut law for *replace*, we get highly nondeterministic results. The first application of put yields nondeterministic values and a consecutive call to put is applied to all these results. That is, for a list with n elements, two consecutive put actions yield n^2 results. For our examples, we use rather small lists to reduce the number of results.

```
> put replace (put replace [1, 2] 4) 5
[5, 2]
[4, 5]
[5, 4]
[1, 5]
```

For the PutPut law, we stated that the set of running two consecutive put actions should at least contain the set of the second put action. In our example, the PutPut law is fulfilled, because $[5, 2]$ and $[1, 5]$ are part of the results. In particular, the PutPut law does not require any result of the first put action to be in the set as well. If we take a look at our example again, we can see that the intermediate results of the first put action – $[4, 2]$ and $[1, 4]$ – are not part of the result.

Moreover, the usage of nondeterministic lenses gives rise to functions that cannot be defined in a deterministic setting. For instance, we would have to change the definition of *replace* as follows.

$$\begin{aligned}
&\mathit{replace}' :: [\,a\,] \to a \to [\,[\,a\,]\,]\\
&\mathit{replace}'\ [\,]\qquad\ \_ = [\,]\\
&\mathit{replace}'\ (x : \mathit{xs})\ y = y : \mathit{xs} : \mathit{map}\ (x{:})\ (\mathit{replace}'\ \mathit{xs}\ y)
\end{aligned}$$

As a consequence, the type of the modified implementation is not applicable for lenses – the first argument and the result have to be of the same type.

Our second example of nondeterministic lenses is the definition of a pretty-printer with a corresponding parser in the get direction. We introduce this concept of so-called *printer-parsers* in detail in Section 6.1. A lens for pretty-printers consists of a get function for parsing, and put function for pretty-printing. The get function can be nondeterministic to model traditional parser structures with a list of results. That is, when we parse a string, we yield all corresponding results nondeterministically.

Later, we introduce printer-parsers with a handful of examples on arithmetic expressions with prefix notation. We decided to use prefix notation, because the corresponding definitions are simple and easy to follow. The string representation of these arithmetic expression are, however, not ambiguous and yield a deterministic result for the parsing direction. Therefore, we want to use an example for arithmetic expressions with infix notation in advance to provide another nondeterministic lens definition.

The following example uses arithmetic expressions that are defined as follows.

$$\begin{aligned}
&\mathbf{data}\ \mathit{Expr} = \mathit{BinOp}\ \mathit{Op}\ \mathit{Expr}\ \mathit{Expr}\\
&\qquad\qquad\ \ \mid \mathit{Num}\ \mathit{Int}\\
&\mathbf{data}\ \mathit{Op}\quad = \mathit{Plus} \mid \mathit{Mult} \mid \mathit{Div} \mid \mathit{Minus}
\end{aligned}$$

Let us assume that we have a lens *ppExpr′* to parse and pretty-print arithmetic expressions in infix notation. Then, we can use the nondeterministic behaviour of the parsing direction to yield all possible results. For example, a simple expression with a binary operator can be parsed in two ways: (1) we only parse the first argument as an identifier; (2) we parse the string as an expression with binary operator, thus, parsing both arguments as identifiers with the operator in the middle.

```
> get ppExpr' "1 + 2"
((BinOp Plus (Num 1) (Num 2)), "")
((Num 1), " + 2")
```

Obviously, the number of possible parsing results increases with the size of the arithmetic expression. For an expression with two binary operators, we have three possible parsing results.

```
> get ppExpr' "1 - 2 * 3"
((BinOp Minus (Num 1) (BinOp Mult (Num 2) (Num 3))), "")
((BinOp Minus (Num 1) (Num 2)), " * 3")
((Num 1), " - 2 * 3")
```

However, we can disambiguate the expression by adding parentheses resulting in a deterministic parsing result.

```
> get ppExpr' "(1 - 2) * 3"
((BinOp Mult (BinOp Minus (Num 1) (Num 2)) (Num 3)), "")
```

In Appendix B, we present the implementation of *ppExpr′* and make additional comments about the problems we ran into. All in all, our definition of a printer-parser and similar ideas given in Section 6.1 are prime examples for the usage of nondeterministic lenses.

6

Case Studies

In this chapter, we discuss two case studies for the application of lenses. First, an implementation of a library for pretty-printing and parsing. Second, a theoretical concept to generate lenses from record data types to manipulate a particular record field.

The first case study uses our put-based lens library to unify the definition of a pretty-printer and parser: a printer-parser. That is, we discuss three different implementations of data structures and lenses for a pretty-printer with a corresponding parser in the get direction. We start off with a simple lens definition without further data structures to model the printer-parser. Then, we give two additional approaches to solve emerging problems and disadvantages of the naive implementation. Moreover, we discuss related work and conclude the section with an evaluation of these approaches and the applicability of lenses for these approaches.

Secondly, we propose to generate lenses for record data types. A similar idea is already integrated in Haskell with the extension *OverloadedRecordFields*[1]. First, we take a look at the current usage of record syntax in Curry to motivate our proposal. Record syntax in Curry is similar to the concept in Haskell, but uses a special record selector in combination with labels instead of a function to project a particular field. Second, we give the definition of the underlying transformations to generate the appropriate lenses for a record data type.

[1]`https://ghc.haskell.org/trac/ghc/wiki/Records/OverloadedRecordFields`

6.1 Case Study I - Bidirectional Printer-Parser

Pretty-printers and parsers are well-studied fields in computer science and take an important role in the design of programming languages. In order to prove a programming language's expressiveness, the implementation of a printer or parser library is a very popular example. Printers and parsers are also represented in processes that include reading and updating data from files. For example, measurements of experiments in the field of biology are often tracked in CSV files. In order to analyse these results, we read the CSV data from the files, manipulate them and write the results back to another file. Whereas parsers specify the format of the CSV data at hand and yield an editable structure in the source language, printers write the results back in a valid format. We observe that printers and parsers should fulfil certain round-tripping rules: we want to print the data in a valid format, such that data can be reread again by the parser. This observation leads to the following equation for a data structure $\mathit{value} :: a$ and associated functions parse_a for parsing and print_a for printing, which are parametrised over the specific data structure.

$$\mathit{parse}_a\ (\mathit{print}_a\ \mathit{value}) = \mathit{value} \tag{Print-Parse}$$

We might also consider the inverse round-tripping rule and demand that if we parse a string and print it again, we get the original string as result. The following equation postulates this round-tripping rule.

$$\mathit{print}_a\ (\mathit{parse}_a\ \mathit{str}) = \mathit{str} \tag{Parse-Print}$$

In certain scenarios, the *Parse-Print* rule is a desirable requirement. For example, if we choose a format for the data that is unambiguous, both round-tripping rules together form the requirements for a correct implementation. *Show* and *Read* instances for data structures in Haskell often form these round-tripping rules as well. On the other hand, in many scenarios regarding data acquisition, users handle data manipulation, update, and sometimes even input manually. Especially in the field of programming languages, users write the code that the parser consumes. Therefore, most programming languages allow redundancies like spaces and parentheses, so that a parsed data structure corresponds to more than one string representation.

In the following, we present different approaches to combine the definition of pretty-printers and parsers into one function. We can achieve this combination in the setting of lenses: we define a lens for pretty-printing in the put and parsing in get direction. In addition, we provide a handful of combinators to simplify the definition of such *printer-parser* for arbitrary data structures.

All approaches have the same interface of combinators, and define a lens of type $\mathit{Lens}\ \mathit{String}\ (a, \mathit{String})$. The underlying lens implementation is the put-based lens library presented in Section 5.2. We provide the following set of combinators, where X stands for the used type of the printer-parser that varies with the different implementations:

- $(<>) :: X\ a \to X\ b \to X\ (a, b)$
- $(<\mid>) :: X\ a \to X\ a \to X\ a$
- $\mathit{digit} :: X\ \mathit{Int}$
- $\mathit{charP} :: (\mathit{Char} \to \mathit{Bool}) \to X\ \mathit{Char}$
- $\mathit{space} :: X\ ()$

As the running example for each implementation we use a standard example in the context of parsers as well as printers: arithmetic expressions. We use the following definition of an algebraic data type for arithmetic expressions.

$$\begin{aligned}
&\mathbf{data}\ \mathit{Expr} = \mathit{BinOp}\ \mathit{Op}\ \mathit{Expr}\ \mathit{Expr} \\
&\qquad\qquad\ \mid \mathit{Num}\ \mathit{Int} \\
&\mathbf{data}\ \mathit{Op} \quad = \mathit{Plus} \mid \mathit{Mult} \mid \mathit{Div} \mid \mathit{Minus}
\end{aligned}$$

6.1.1 Printer-Parser

The first implementation that we present is based on a lens definition that pretty-prints a data structure in the put direction, and parses a string into that data structure in the get direction.

The lens has the form **type** $\mathit{PPrinter}\ a = \mathit{Lens}\ \mathit{String}\ (a, \mathit{String})$. Remember, the extended type signature of $\mathit{PPrinter}\ a$ corresponds to $\mathit{String} \to (a, \mathit{String}) \to \mathit{String}$. The first argument is the input string, the second argument is a pair consisting of the actual data type that we want to print and parse, and a remaining string. This representation[2] is most commonly used for pretty-printers to achieve a linear run-time subject to the resulting pretty-printed string. With this lens definition at hand, we can define two functions *pParse* :: *PPrinter a* → *String* → *a* and *pPrint* :: *PPrinter a* → *a* → *String*.[3] Both functions are parametrised over a printer-parser lens; they run this lens with the given string in the get direction and with the given data structure in put direction, respectively.

```
pParse :: PPrinter a → String → a
pParse pp str = maybe err fst (find ((≡ "") ∘ snd) values)
where
  values = getND pp str
  err = error "no complete parse"
pPrint :: PPrinter a → a → String
pPrint pp val. = pp "" (val., "")
```

The definition of *pParse* first collects all results of the given printer-parser running in the get direction by using *getND*. The auxiliary *getND* uses *Set Functions* to yield a list of results instead of nondeterministic results. *pParse* chooses the first pair of the resulting list that has no remaining string, i.e., a result with a complete parse, and yields the parsed value. If there are no complete parses, we throw an error.

The function *pPrint* applies the given printer-parser to the empty string as first argument to trigger a pretty-print.

Primitives

As first primitive printer-parser, we define *digit* :: *PPrinter Int* that pretty-prints a digit in the *put*- and parses a digit in the *get*-direction.

```
digit :: PPrinter Int
digit _ (d, str′) | 0 ⩽ d ∧ d ⩽ 9 = show d ++ str′
```

We ignore the given string, replace it with the representation of the given digit, and add the remaining string at the end of the resulting string. Since the type of the data

[2]a slightly different one, but still equivalent

[3]Note: the additional *"p"* stands for *"pretty"*.

is not restricted to digits only, we include a check if the range of the given integer value is between 0 and 9.

The primitive *charP* prints and parses only characters that fulfil a given predicate. We can easily define a combinator that prints and parses a space character by means of *charP*.

$charP :: (Char \rightarrow Bool) \rightarrow PPrinter\ Char$
$charP\ p\ \_\ (c, str') \mid p\ c = c : str'$

$space' :: PPrinter\ Char$
$space' = charP\ (\equiv \texttt{' '})$

In order to define more meaningful printer-parsers, e.g., for the arithmetic expression we introduced earlier, we need combinators to compose primitives. The first composition combinator builds a pair from two given printer-parsers.

$(<>) :: PPrinter\ a \rightarrow PPrinter\ b \rightarrow PPrinter\ (a, b)$
$(pA <> pB)\ str\ ((expr1, expr2), str') = pA\ str\ (expr1, newString)$
where
  $newString = pB\ str\ (expr2, str')$

In order to get a better understanding of this composition, we expand the type of the result, i.e., $PPrinter\ (a, b)$ becomes $String \rightarrow ((a, b), String) \rightarrow String$. This observation leads to the four given arguments: the two printer-parsers, the given string and a pair of data structures paired with the remaining string. We can compose these two printer-parsers to achieve a meaningful consecutive execution. First, we apply the second printer-parser *pB* to produce a new string. Then, the resulting string is used as remaining string in the application of the other printer-parser *pA*. This construction works straightforward, because the given string is just replaced with the pretty-printed result in the definition of the primitives.

Furthermore, we provide a combinator to pretty-print one of two alternatives. That is, we have two printer-parsers at hand and run the second one only if the first one fails.

$(< \mid >) :: PPrinter\ a \rightarrow PPrinter\ a \rightarrow PPrinter\ a$
$(pA1 < \mid > pA2)\ str\ pair =$ **case** $isEmpty\ (set2\ pA1\ str\ pair)$ **of**
  $True \rightarrow pA2\ str\ pair$
  $False \rightarrow pA1\ str\ pair$

In this case, we use Set Functions to reason about a possibly failing replace-parser. That is, if the first replace-parser does not yield any result when applied to the appropriate arguments, i.e., the list of all results is empty, we run the second one instead.

Parsing and Printing Arithmetic Expressions in Prefix Notation

Next, we define the appropriate printer-parsers for the data structure of arithmetic expressions given above. In order to define a printer-parser for our data structure *Expr*, we need to cover both representations of the definition: an expression can be a number, or a composition of two expressions with a binary operator. For the purpose of simplicity, we implement the pretty-print of an arithmetic expression in prefix notation.

$$\begin{array}{l}
ppExpr :: PPrinter\ Expr \\
ppExpr\ str\ (Num\ v, str') = digit\ str\ (v, str') \\
ppExpr\ str\ (BinOp\ op\ e1\ e2, str') = \\
\quad ((ppOp <> ppExpr) <> ppExpr)\ str\ (((op, e1), e2), str')
\end{array}$$

In the case of a given *Num*-constructor, we use the primitive *digit* and, thus, limit the identifiers of the arithmetic expressions to integer values between 0 and 9.[4] The rule for a binary operator looks a bit more complicated. However, the rule follows naturally from the definition of the data type. At first, we print the given operator followed by both of its arguments. The arguments of a binary operator are arithmetic expressions, thus, we print the arguments in a recursive manner. Due the two consecutive uses of the composition combinator, (<>), the data have to be provided in form of a nested pair. The inner pair consists of the operator and the first arithmetic expression. Then, this pair is injected as the first component of the outer pair because of the second usage of (<>). The part of the second component of this outer pair is assigned to the second argument of the binary operator. In order to test this lens definition, we need to define the printer-parser for operators of type *Op* first.

[4]We make this limitation due to simplicity reasons, we could easily write a version that prints an *Int* value instead.

$$\begin{array}{l}
\mathit{ppOp} :: \mathit{PPrinter\ Op} \\
\mathit{ppOp\ str}\ (\mathit{op}, \mathit{str}') = \mathit{charP\ isOp\ str}\ (\mathit{fromJust\ opStr}, \mathit{str}') \\
\quad \textbf{where} \\
\qquad \mathit{opStr} = \mathit{lookup\ op}\ [(\mathit{Plus}, \texttt{'+'}), (\mathit{Minus}, \texttt{'-'}), (\mathit{Mult}, \texttt{'*'}), (\mathit{Div}, \texttt{'/'})] \\
\mathit{isOp} :: \mathit{Char} \to \mathit{Bool} \\
\mathit{isOp}\ c = c \in \texttt{"+*-/"}
\end{array}$$

We use the obvious symbols as string representatives for the arithmetic operators. Note, we use *fromJust*[5] from the `Maybe` library that unwraps the *Just* constructor from the given value.

With these lens definitions at hand, we can use the dedicated function *pPrint* to see the pretty-printed version of an exemplary arithmetic expression.

$$\begin{array}{l}
> \mathit{pPrint\ ppExpr}\ (\mathit{Num}\ 3) \\
\texttt{"3"} \\
> \mathit{pPrint\ ppExpr}\ (\mathit{BinOp\ Plus}\ (\mathit{Num}\ 2)\ (\mathit{Num}\ 3)) \\
\texttt{"+23"}
\end{array}$$

Unfortunately, we were a bit careless when we defined the string representation of an expression with a binary operator. The resulting string is not *pretty* at all. The upside is that the parser already works very well. In the following examples, we use *pParse* to reconstruct the expression from the string representation.

$$\begin{array}{l}
> \mathit{pParse\ ppExpr}\ \texttt{"+23"} \\
(\mathit{BinOp\ Plus}\ (\mathit{Num}\ 2)\ (\mathit{Num}\ 3)) \\
> \mathit{pParse\ ppExpr}\ \texttt{"42"} \\
\mathit{Main} : \mathit{UserException}\ \texttt{"no complete parse"} \\
\mathit{Evaluation\ terminated\ with\ non - zero\ status}\ 1
\end{array}$$

In the first expression, we gain the original expression when parsing `"+23"` as expected. Due to the restricted representation of numeric identifier as digits, each digit of the string `"23"` is parsed as an argument of the binary operator *Plus*. Because of this restriction, we cannot, however, parse a numeric value like `"42"`. Hence, the last

[5] $\mathit{fromJust} :: \mathit{Maybe}\ a \to a$
$\mathit{fromJust}\ (\mathit{Just}\ v) = v$
$\mathit{fromJust\ Nothing} = \mathit{error}$ `"Maybe.fromJust: Nothing"`

expression *pParse ppExpr* `"42"` fails with the remark that the given string cannot be parsed completely. Without investigating this failing expression in too much detail, we can take a quick look at the parsing result. The type of *ppExpr* is *PPrinter Expr*, that is, the function *ppExpr* is a lens function. For that reason, we can use *ppExpr* in get and put direction. As mentioned before, whereas the put direction corresponds to a pretty-print, we can parse a string into a value of type *Expr* when using the get direction of the lens definition.

$> get\ ppExpr$ `"42"`
$(Num\ 4,$ `"2"`$)$

In this case, the expression *get ppExpr* `"42"` is of type $(Expr, String)$ and yields the parsed value and the remaining string. Remember, the definition of *pParse* distinguishes if the resulting remaining string is the empty string or not. In case of a non-empty string, the parse could not be completed and the function throws an exception.

Pretty Arithmetic Expressions

In order to continue on *pretty*-printing arithmetic expression, we have to rewrite the second rule of *ppExpr*'s definition. Obviously, a pretty representation of arithmetic expression needs a space character between each token. For the purpose of adding space to the string representation, we integrate the function *space′* into our definition of *ppExpr* from above. In the end, we add a space after the binary operator and the first argument of that operator.

$ppExpr\ str\ (BinOp\ op\ e1\ e2, str') =$
$\quad ((ppOp <> space') <> (ppExpr <> space')$
$\qquad <> ppExpr)\ str\ ((((op, \_), (e1, \_)), e2), str')$

Unfortunately, due to the integration of space and the related increased usage of the composition combinator, this definition looks rather cumbersome and complicated. In particular, the spaces that we added are ignored in the data that are given in form of a deeply nested pair. The definition above uses an anonymous free variable, which is bound to a space character when actually calling this function. The variable is bound to a space, because a space is the only valid value, such that the *space′* function yields a result. Instead of using an anonymous free variable, we can explicitly assign space characters as arguments.

In the case of an injected prettiness factor like additional spaces, it would be convenient to provide a combinator to ignore a printer-parser's result. Those additional adjustments regarding the printed string occur mostly in the context of composition. Thus, we define two additional composition functions to ignore the printer-parser to the right and to the left, respectively. We can implement such a combinator by means of (<>).

$$(<<<) :: \mathit{PPrinter}\ a \to \mathit{PPrinter}\ () \to \mathit{PPrinter}\ a$$
$$(\mathit{pA} <<< \mathit{pB})\ \mathit{str}\ (\mathit{expr}, \mathit{str}') = (\mathit{pA} <> \mathit{pB})\ \mathit{str}\ ((\mathit{expr}, ()), \mathit{str}')$$
$$(>>>) :: \mathit{PPrinter}\ () \to \mathit{PPrinter}\ b \to \mathit{PPrinter}\ b$$
$$(\mathit{pA} >>> \mathit{pB})\ \mathit{str}\ (\mathit{expr}, \mathit{str}') = (\mathit{pA} <> \mathit{pB})\ \mathit{str}\ (((), \mathit{expr}), \mathit{str}')$$

Unfortunately, we need to restrict the type of the ignored result to the unit type. We have to make up a value to pass to (<>) as first and second argument, respectively. Since we cannot always devise a suitable value for any type, we restrict these combinators to use unit only. The unit type has only one valid value, thus, we can always pass along () as argument.

In order to use these combinators in our definition of *ppExpr*, we have to implement a space printer-parser of type *PPrinter* ().

$$\mathit{space} :: \mathit{PPrinter}\ ()$$
$$\mathit{space}\ \_\ ((), \mathit{str}') = \texttt{" "} + \mathit{str}'$$

Fortunately, the definition is straightforward: we ignore the given string and pattern match on the unit value in order to produce a space in front of the remaining string, which is given as part of the input pair.

Last but not least, we can integrate the newly defined functions in our definition of *ppExpr* to enhance the readability.

$$\begin{aligned}
&\mathit{ppExpr}\ \mathit{str}\ (\mathit{BinOp}\ \mathit{op}\ \mathit{e1}\ \mathit{e2}, \mathit{str}') = \\
&\quad ((\mathit{ppOp} <<< \mathit{space}) <> (\mathit{ppExpr} <<< \mathit{space}) \\
&\qquad <> \mathit{ppExpr})\ \mathit{str}\ (((\mathit{op}, \mathit{e1}), \mathit{e2}), \mathit{str}')
\end{aligned}$$

In order to assure that our reimplementation produces prettier string representations, we run our examples from above again. Furthermore, the string representations are supposed to be parsable as well. Thus, we also give some examples for parsing valid string representations of arithmetic expressions as well as some cases, where our parser does not yield a completely parsed string.

```
> pPrint ppExpr (BinOp Plus (Num 2) (Num 3))
"+ 2 3"
> pParse ppExpr "+ 2 3"
BinOp Plus (Num 2) (Num 3)
> pParse ppExpr "+ + 2 3 4"
BinOp Plus (BinOp Plus (Num 2) (Num 3)) (Num 4)
> get ppExpr "+ 2 34"
(BinOp Plus (Num 2) (Num 3), "4")
> get ppExpr "+ 23 4"
  -- no result
```

The second and third example show a successful parse for a simple and a nested arithmetic expression, respectively. In the forth expression, we reuse the example from above that has a number with two digits as second argument for *Plus*, thus, leading to a remaining string. Note, the parse does not fail completely: the prefix `"+ 2 3"` is parsed correctly and the parser yields *BinOp Plus* (*Num* 2) (*Num* 3) as resulting expression. In contrast, the last example fails completely and has no result. The given string `"+ 23 4"` does not consist of a valid prefix to yield a partial result with a remaining string.

Multiple Spaces

As a main disadvantage of this approach, we cannot ignore redundant parts in the parsing direction. If these redundancies do not appear in the pretty-printer's definition, we do have no option to parse them anyway. We have already mentioned an original precedent in the introduction of this section: optional spaces as delimiter between tokens. In order to make this case more clear, we define a printer-parser for one or more spaces.

```
spaces1' :: PPrinter [()]
spaces1' str (x : xs, str') =
  (space <> spaces') str ((x, xs), str')
```

```
spaces' str ([], str)        = str'
spaces' str (x : xs, str') =
  space str (x, "") ++ spaces' str (x, str')
```

The function *spaces1'* pretty-prints a series of spaces depending on the length of the given list, which cannot be empty. In contrast, the auxiliary function *spaces'* can pretty-print zero or many spaces. For the parsing direction, we want both functions to parse a series of spaces. A generalisation of these functions comes in handy to parse a series of characters or strings of the same category and to pretty-print a list of elements respectively. For that purpose, we define a generalised version *many1* and *many* that takes a printer-parser as argument and applies it to each element of a given list.

```
many :: PPrinter a → PPrinter [a]
many _ _ ([], str') = str'
many pp str (x : xs, str') = (pp <> many pp) str ((x, xs), str')


many1 :: PPrinter a → PPrinter [a]
many1 pp str (x : xs, str') = (pp <> many pp) str ((x, xs), str')
```

With this definition at hand, we can define a modified version of *spaces*.

```
spaces :: PPrinter [()]
spaces = many1 space
```

Next, we integrate the additional trailing spaces into our printer-parser for arithmetic expressions. This integration implicates to change the usage of (<<<) and (>>>) to the traditional composition operator again. We can only ignore data of type unit, but *spaces* expects a list of unit.

```
ppExprSpaces :: PPrinter Expr
ppExprSpaces str (BinOp op e1 e2, str') =
  ((ppOp <> spaces) <> (ppExprSpaces <> spaces)
    <> ppExprSpaces) str (((opSpaces, e1Spaces), e2), str')
  where
    opSpaces = (op, _)
    e1Spaces = (e1, _)
ppExprSpaces str (Num v, str')            = digit str (v, str')
```

Since we do not care about the number of trailing spaces after an operator and its expressions, respectively, we use an anonymous free variable as input. This free variable can be bound to any number of unit elements in order to parse all occurring spaces. As a first example, we can parse trailing spaces after the binary operator as well as a pretty-printed version.

```
> pParse ppExprSpaces "+    1 2"
(BinOp Plus (Num 1) (Num 2), "")
> pParse ppExprSpaces "+ 3 4"
(BinOp Plus (Num 1) (Num 2), "")
```

Fortunately, the integration of redundant spaces works like a charm.

The Downside

However, there is a downside to this solution. In the beginning we said that we cannot parse redundancies that are not included in the printer-parser's definition. In our case, we have added these redundancies and, thus, can parse them in a convenient way. This observation leads to the question: how does this integration effect the pretty-printing of the arithmetic expression? In the pretty-printed version of an arithmetic expression, we do not allow any leading and trailing spaces. Let us test the behaviour by pretty-printing the expression from above.

```
> pPrint ppExprSpaces (BinOp Plus (Num 1) (Num 2)
"+ 1 2"
"+ 1  2"
"+  1 2"
"+ 1   2"
    ⋮
```

Unfortunately, this expression does not terminate, but yields all possible versions of string representatives. Possible versions include a different number of trailing space after the operator and the first argument of that operator. This unsatisfactory result arises from the use of the free variables in the definition of *ppExprSpaces*. We cannot set the number of used spaces to one; the free variable is instantiated nondeterministically to a suitable value. In our case, $[()]$ is not the only suitable value, any list of unit values fits the specification of the pretty-printer.

The work of Foster et al. (2008) sounds promising as a solution to our problem. They introduce *quotient lenses*, well-behaved bidirectional transformations that allow the programmer to specify equivalence relations on the data he wants process. The authors integrated an implementation of quotient lenses to the Boomerang language. However, we did not have the time to implement a mature version of our own quotient lenses in the context of our study case.

As a second disadvantage, the definitions for the pretty-printer have to be more sophisticated than usual. The printer and parser definitions are connected, that is, typical restrictions known from parser constructions need to be considered. In our example, we defined a string representation for arithmetic expression in prefix notation, but with infix operators, we have to avoid left-recursion in the definitions. This modification leads to a more complex definition of the printer-parsers that bears a resemblance to a typical parser for arithmetic expression with infix operators. For the interested reader, we give the implementation for arithmetic expressions with infix operators as well as some examples for pretty-printing and parsing in Appendix B.

6.1.2 Replace-Parser

In a second approach, we want to tackle the disadvantages of printer-parsers concerning redundancies and optional parsing rules. These disadvantages arise from the operational, state-free approach of our first implementation. The design of printer-parsers provides the definition of pretty-printers and implicates a corresponding parser. Unfortunately, the corresponding parser is only suited for the pretty-printed string representation. In the previous subsection, we gave an unsatisfactory definition to parse a variable amount of spaces. The defined lens can be used in the parsing direction, but behaves heavily nondeterministic when pretty-printing a value. One cause of the problem is that we cannot reason about single steps of the underlying parser without changing the semantics of the pretty-printer. Therefore, we want to discuss another implementation that has more information about intermediate results. In order to achieve this additional information, we change the underlying data structure in contrast to the printer-parser. Because of the changed data structure, we can also adjust the semantics and add an inspection of the given input string. Instead of replacing the input string completely, we want to perform a layout preserving replacement. The new implementation provides a function $\mathit{replaceParse} :: \mathit{PReplace}\ a \to \mathit{RPLens}\ a$ that takes a specification of a so-called *replace-parser* and yields a lens function. The resulting lens has the same type as our printer-parser.

```
type RPLens a = Lens String (a, String)
```

We can use this lens function in the common way using *get* and *put*. The get function for a *RPLens* corresponds to a parsing action like before and in the put direction we replace a given string and try to preserve its layout. If the given string is empty or does not fulfil the replace-parser's specification, we pretty-print the data structure at hand. As an example: we have an arithmetic expression with more than one space as delimiter for its arguments and update only the second argument.

```
> put (replaceParse rpExpr) "+  3 2" (BinOp Plus (Num 1) (Num 2), "")
"+  1 2"
```

The result of our replace-parser assures that the layout of the given string is preserved. On the other hand, we can parse the resulting string with the replace-parser again and, hopefully, regain the original value of the arithmetic expression.

```
> get (replaceParse rpExpr) "+  1 2"
(BinOp Plus (Num 1) (Num 2), "")
```

Indeed, the parsing direction works like a charm and ignores the redundant spaces. As mentioned above, if the given string is empty, the replace-parser prints the given value in its prettiest version, i.e., as specified in the lens definition of *rpExpr*. The second argument of the pair represents a remaining string as in the previous version. We can use this component to add a generic string at the end.

```
> put (replaceParse rpExpr) "" (BinOp Mult (Num 3) (Num 7), "")
"* 7 2"
> put (replaceParse rpExpr) "" (BinOp Mult (Num 3) (Num 7), " test123")
"* 7 2 test123"
```

Implementation of Primitives

In the following, we take a closer look at the underlying implementation. At first, we want to discuss the used data structure that tracks the intermediate result of a replacement and pretty-print, respectively. We called this data structure the specification of a replace-parser above.

```
type PReplace a = String → (a, String) → Res String
data Res a = New a
           | Replaced a
```

We can observe that the used data structure is a variant of the *RPLens* that wraps the resulting string into a new data structure. This new data structure gives information about an action in put direction. If the resulting string is wrapped with the *New* constructor, the replace-parser performed a pretty-print on the given value. Otherwise, the value was pretty-printed with respect to the underlying layout of the given string. In order to give an example, we get the following results for a primitive combinator *charP* :: (*Char* → *Bool*) → *PReplace* () based on the given string.

```
> charP isDigit "1" ('5',"")
Replaced "5"
> charP isDigit "a" ('5',"")
New "5"
> charP isDigit "12" ('5',"")
New "5"
```

In these examples, we want to replace a digit character with `'5'`. The first example succeeds with a replacement, because the given string consists of a digit as well. In the second and third exemplary expression, we ignore the input, because it does not harmonise with the given specification and pretty-print the given value.

For the purpose of clarification, we start with the implementation of *charP* as first primitive combinator for replace-parsers. We replace the input and pretty-print the value for two cases: if the input string is empty, or if the input does not fulfil the given predicate.

```
charP :: (Char → Bool) → PReplace Char
charP p input (v, new) = case input of
                             "" → New (v : new)
                             _  → char′ input new
  where
    char′ (c′ : str′) rest
      | p c′ ∧ (null str′ ∨ rest ≡ str′) = Replaced rest
      | otherwise                        = New (v : rest)
```

In particular, we check the predicate on the first character of the input string, and restrict the remaining input to be empty or to be equal to the given remaining string *str′*. That is, if we want to replace a digit with another digit and the input string consists of two digits, the specification is not fulfilled and we pretty-print the value. However, for an identical remaining string, the replace-parser can perform a replacement, indicated by the usage of the *Replaced* constructor. We give two additional examples to clarify this circumstance.

```
> charP isDigit "41" ('5', "")
New "5"
> charP isDigit "41" ('5', "1")
Replaced "51"
```

With the help of *charP*, we can define a series of additional primitives. In the examples, we used *charP* to replace and pretty-print a digit; we can enhance this expression to be applicable for actual *Int* values. In addition, we can define a primitive to handle a space.

```
digit :: PReplace Int
digit str (d, str′)
  | 0 ⩽ d ∧ d ⩽ 9 = (charP isDigit) str (intToDigit d, str′)
space :: PReplace ()
space str ((), str′) = charP (≡ ' ') str (' ', str′)
```

We use the auxiliary function *intToDigit* :: *Int* → *Char* from the `Char` library that converts an integer value to a character. The condition $0 \leqslant d \wedge d \leqslant 9$ guarantees that no integer value greater than 9 is converted into a character.

Composition

Moreover, we want to compose several primitives to build new replace-parsers. For that purpose, we introduce the composition combinator (<>), and its descendants (<<<) and (>>>) in order to ignore the left and right result, respectively.

```
(<>) :: PReplace a → PReplace b → PReplace (a, b)
(pA <> pB) str ((expr1, expr2), str′)
   | null str = pA str (expr1, unwrap (pB str (expr2, str′)))
   | str ≡ str1 ++ str2 =
     (strict pA) str1 (expr1, unwrap ((strict pB) str2 (expr2, str′)))
  where str1, str2 free
```

When composing two replace-parsers, we split the input string into two parts. We take advantage of the logic features of Curry and split the input string nondeterministically. This idea is adopted from the general approach of the `Parser` library[6] in Curry that is based on functional-logic parsers as presented by Caballero and Lòpez-Fraguas (1999). In case of an empty input string, we run the first replace-parser on its corresponding value and its part of the input. As remaining string, we provide the result of the second replace-parser that works on its part of the input string and the given remaining string. That is, we concatenate the results of both parsers and add the remaining string at the end. Because of the constructor wrapped around the resulting string, we use the auxiliary function *unwrap* to access the containing string.

```
unwrap :: Res a → a
unwrap (New v)      = v
unwrap (Replaced v) = v
```

As an important part of the second case, we use the function *strict* to ensure that the given replace-parser performs a replacement.

```
strict :: PReplace a → PReplace a
strict pReplace str pair = case pReplace str pair of
                                New _ → failed
                                res → res
```

[6]http://www-ps.informatik.uni-kiel.de/kics2/lib/CDOC/Parser.html

We want to guarantee that both replace-parsers perform a replacement on their dedicated input string. Otherwise, the resulting string could be a combination of a layout-preserving and pretty-printed variant of the given value. In our implementation, the composition fails if one of the given replace-parsers fails to replace its value for the given input. In particular, the definition behaves nondeterministically and tries every combination of two substrings that can be combined to the input string. As a convenient side-effect, the usage of *strict* guarantees that an empty substring fails for both replace-parsers. The *strict* function fails for every value constructed with the *New* constructor and an empty string as input causes most primitives to perform a pretty-print. Thus, these primitives yield a string wrapped in a *New* constructor. In the case of an empty input string, we apply both replace-parsers in series without using the strict version.

Furthermore, the implementation of (<<<) and (>>>) is straightforward, and the same as for the printer-parser, but with an adapted type signature.

$$(<<<) :: \mathit{PReplace}\ a \rightarrow \mathit{PReplace}\ () \rightarrow \mathit{PReplace}\ a$$
$$(pA <<< pB)\ \mathit{str}\ (e, \mathit{str}') = (pA <> pB)\ \mathit{str}\ ((e, ()), \mathit{str}')$$
$$(>>>) :: \mathit{PReplace}\ () \rightarrow \mathit{PReplace}\ b \rightarrow \mathit{PReplace}\ b$$
$$(pA >>> pB)\ \mathit{str}\ (e, \mathit{str}') = (pA <> pB)\ \mathit{str}\ (((), e), \mathit{str}')$$

Arithmetic Expressions Revisited

In order to compare this approach with our previous implementation of printer-parsers, we define a replace-parser for arithmetic expressions.

$$\begin{aligned}
&\mathit{rpExpr} :: \mathit{PReplace}\ \mathit{Expr}\\
&\mathit{rpExpr}\ \mathit{str}\ (\mathit{BinOp}\ \mathit{op}\ \mathit{e1}\ \mathit{e2}, \mathit{str}') =\\
&\quad ((\mathit{rpOp} <<< \mathit{spaces}) <> (\mathit{rpExpr} <<< \mathit{spaces})\\
&\qquad\qquad\qquad\qquad <> \mathit{rpExpr})\ \mathit{str}\ (((\mathit{op}, \mathit{e1}), \mathit{e2}), \mathit{str}')\\
&\mathit{rpExpr}\ \mathit{str}\ (\mathit{Num}\ v, \mathit{str}') \qquad = \mathit{digit}\ \mathit{str}\ (v, \mathit{str}')
\end{aligned}$$

The definition of a replace-parser for arithmetic expressions has a high resemblance to our version for printer-parsers. This resemblance arises from the usage of the same set of primitives and combinators that can be used to define more complex function definitions. The only difference is that we use *spaces* here. Because of the underlying data structure of this approach, we can finally implement primitives to parse optional redundancies in the parsing direction without interfering with the pretty-printer.

The following code shows the implementation of *spaces* that pretty-prints exactly one space, but can parse and replace several spaces.

```
spaces :: PReplace ()
spaces input = case input of
  "" → space ""
  _  → (space >>> spaces') input
where
  spaces' input' ((), str') = case input' of
    "" → pure input' ((), str')
    _  → (space >>> spaces') input' ((), str')
```

In case of an empty input string, we pretty-print one space. Otherwise, we read or replace as much spaces as possible until the input string is empty. If the input string is finally empty, the first rule of the local function *spaces′* is used to add the remaining string to the end of the resulting string. Here, we use the auxiliary function *pure* that ignores its input string as well as its value, and yields the remaining string as a result of a replacement. That is, the resulting string is wrapped in the *Replaced* constructor and *pure* always succeeds to replace a given input string.

```
pure :: PReplace a
pure _ (_, str') = Replaced str'
```

This implementation differs from the one given for the printer-parser. Using a primitive like *many* and a free variable in the implementation of *rpExpr* does not lead to success either, because of the introduced nondeterminism. Nevertheless, the idea of this implementation is not applicable for the printer-parser, because we take advantage of the implementation of (>>>). In the replace-parser's version of (>>>), we actually consume the input string. Thus, we actually reach the point where we stop producing or reading spaces. In case of printer-parsers, the input string is ignored completely, hence, leading to a failing parsing action on every input string.

Last but not least, we use the same implementation to handle the operators of an arithmetic expression like for the printer-parser. In fact, if it were not for the delimiting spaces, we could have used the exact same implementation as before, but using a different type signature.

Poor Performance

The more interesting part is the behaviour of our given implementation for replace-parsers. We have already seen a handful of examples as a motivation for the idea of replace-parsers in the beginning of this subsection. Nonetheless, we want to give some more examples to highlight the improvement in contrast to the previous implementation.

```
> put (replaceParse rpExpr) "+  1  2" (BinOp Mult (Num 3) (Num 4), "")
"*  3  4"
> get (replaceParse rpExpr) "+  1     2"
(BinOp Plus (Num 1) (Num 2)), "")
```

Unfortunately, this approach comes with some disadvantages, too. The second example indicates a performance problem due to the string splitting on combinators like (<>),(<<<) and (>>>). Collectively, there are four redundant spaces to consume when replacing the given string with a new value. We introduce an additional combinator, (<<<), that splits the input string into two parts for each space. The longer the input string and the more composition combinators we use, the more splitting combinations arise and the number of splits increases, respectively. That is, the nondeterministic search for a suitable splitting increases fast and causes a bad performance. This performance issue affects the parsing direction as well as the get direction in case of a replacement.

In the following, we show a series of graphs to illustrate the performance issue.[7] Both graphs have the same labels: the x-axis represents the number of combinators used in an expression; the y-axis indicates the execution time of an expression. In Figure 6.1, we show the execution time of a replacement action for an increasing number of combinators. The behaviour of the graph indicates an exponential growth. In order to investigate this hypothesis, we used a logarithmic scale for the execution time in Figure 6.2. Indeed, the adjusted graph shows a linear growth, which indicates an overall exponential runtime with respect to the number of combinators.

We also measured the performance for a parsing action and, unfortunately, the results are even worse. The results are illustrated with a logarithmic scale in Figure 6.3;

[7] We tested the performance with KiCS2 and the *+time* flag; the testing device was a MacBook Air with Mac OS 10.9.4 as operation system, 4 GB memory, and a 1.8 GHz Intel Core i5 as processor.

the runtime of a parsing action behaves highly exponential as well. Parsing a string with our library performs bad, because the get function guesses the resulting value – the arithmetic expression. Thus, the performance depends on the size of the arithmetic expression and of the number of possible values of each component. For example, the operator data type has four possible constructors; each of this constructor is tested in order to find the corresponding arithmetic expression for a given string.

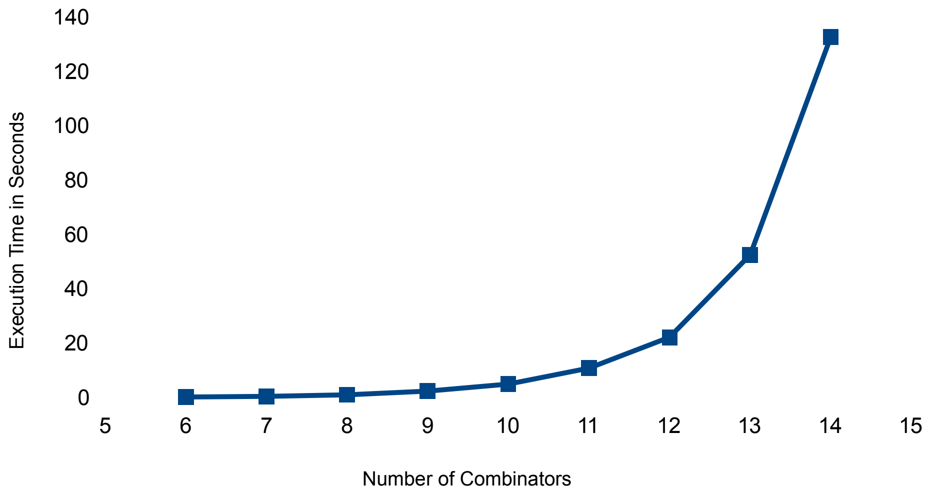

Figure 6.1.: Performance of replacing for increasing number of combinators

However, the pretty-printer still performs quite well for large expression terms.

```
> put (replaceParse rpExpr)
      ""
      (BinOp Mult (BinOp Plus
                      (BinOp Plus
                              (BinOp Plus (Num 3) (Num 1))
                              (BinOp Minus (Num 7) (Num 3)))
                      (BinOp Div (Num 8) (Num 2)))
              (BinOp Minus (Num 3) (Num 1)), "")
"* + + + 3 1 - 7 3 / 8 2 - 3 1"
```

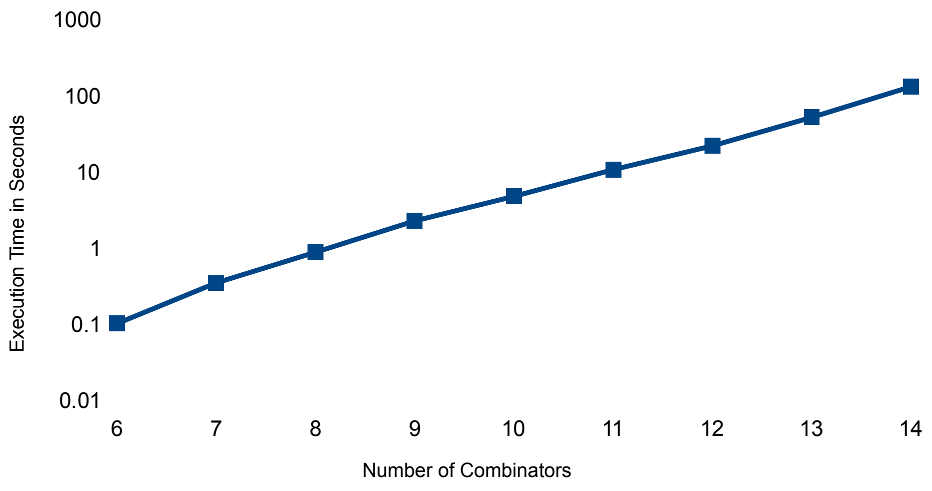

Figure 6.2.: Performance of replacing for increasing number of combinators with logarithmic scale

The expression consists of thirty combinators, has three levels of nesting for recursive calls, and executes in 2 milliseconds. In comparison to the get direction, we validate the execution time as a good result.

Ace In The Hole

Although we do not prioritise a good performance, we take a last try on implementing a replace-parser. Instead of a functional-logic approach that makes heavy use of nondeterminism, we change our underlying implementation to aim for a more functional approach. The weak point of our first implementation was the composition combinator. When composing two replace-parsers, we make a guess on how to split the input string, such that both replace-parsers yield appropriate results. In the functional approach, the first parser yields a remaining string that is used as input for the second parser. Our parsing result has the same representation: the result of the get direction is $(a, String)$. However, we cannot effectively use this result for the definition of a replace-parser, because of the underlying put-based lenses. Hence, we need

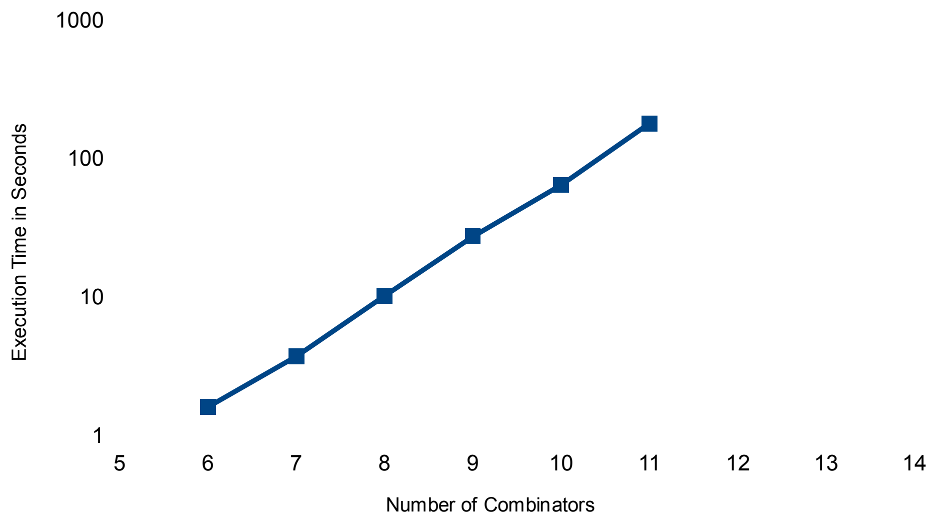

Figure 6.3.: Performance of parsing for increasing number of combinators with logarithmic scale

to find a convenient representation for the underlying data structure in order to use intermediate results for the parsing direction as well.

$$\textbf{type}\ \mathit{PReplace}\ a = \mathit{String} \to (a, \mathit{String}) \to (\mathit{String}, \mathit{String}, \mathit{String})$$

We choose a triple as result of our replace-parser. The first component represents the string of a successful replacement or pretty-print, whereas the other two components are remaining strings. We distinguish between the remaining input string that has not been consumed yet as the second component, and, lastly, the remaining string to concatenate at the end of the resulting string.

As a first representative example, we define a primitive to handle a char that fulfils a given predicate, again.

$$\begin{aligned}
&\mathit{charP} :: (\mathit{Char} \to \mathit{Bool}) \to \mathit{PReplace}\ \mathit{Char} \\
&\mathit{charP}\ \_\ \texttt{""} \qquad (e, \mathit{str}') = ([e], \texttt{""}, \mathit{str}') \\
&\mathit{charP}\ p\ (c : cs)\ (e, \mathit{str}') \\
&\quad \mid p\ c = ([e], cs, \mathit{str}')
\end{aligned}$$

For an empty input string, we pretty-print the given char value – the pretty-printed result is the first component of the triple. In case of a replacement, we consume the first character if the predicate holds, and remember the remaining input in the second component. The given remaining string is transfered as the third component of the triple.

With this technique at hand, we can define a more convenient composition combinator that does not rely on nondeterminism.

```
(<>) :: PReplace a → PReplace b → PReplace (a, b)
(pA <> pB) input ((e1, e2), str') = case input of
    "" → (res1, str2, str1' ++ str2')
    _  → if null str1 ∧ (res2, str', str2') ≢ pure str1 (e2, str1')
            then failed
            else (res1, str2, str1' ++ str2')
  where
    (res1, str1, str1') = pA input (e1, res2)
    (res2, str2, str2') = pB str1 (e2, str')
```

The general idea of the combinator is as before: we apply the first replace-parser on the input string, the corresponding value, and the result of the second parser. The important difference is that we can actually use the remaining string of the first replace-parser as argument for the second one. Unfortunately, we lose performance by concatenating both remaining strings for the resulting tuple. However, this concatenation is rather a technicality than a huge problem. The additional check on a non-empty string can be seen as an equivalent to the usage of *strict* in the previous implementation. When composing two replace-parsers, we want to make sure that both actually consume any input. However, there is one exception: if the given replace-parser does not care about its input, we do not want the composition to fail. Therefore, we use *pure* – a primitive replace-parser that succeeds on every input – to handle the special case of a non-consuming replace-parser.

In order to use the definition of *rpExpr* that we have given above, we still need to define *spaces*. The key for a modified definition of *spaces* is a combination of *pure* and the alternative combinator (< | >). In Curry, we can define the alternative combinator using the choice operator, and nondeterministically choose one of the given replace-parsers.

```
(<|>) :: PReplace a → PReplace a → PReplace a
(pA <|> _) "" = pA ""
(pA <|> pB) input@(_: _) = (pA ? pB) input
```

In order to restrict the nondeterministic behaviour to the parsing direction and a replacement action, we simply apply only the first replace-parser for an empty string. That is, pretty-printing is still deterministic, and does not introduce a choice between the two given replace-parsers. In the end, we can define *spaces* as follows.

```
spaces :: PReplace ()
spaces input = case input of
    "" → space ""
    _  → ((space >>> spaces')) input
  where
    spaces' :: PReplace ()
    spaces' input' ((), str') = case input' of
      "" → pure "" ((), str')
      _  → ((space >>> spaces') <|> pure) input' ((), str')
```

The definition of *spaces'* benefits from the combination of the alternative combinator and *pure*. This combination allows us to consume an arbitrary number of spaces, and stop whenever the *space* parser fails. We also stop if the input string becomes empty, and yield a result to express a successful consumption.

Due to the actual consumption of the input string, we achieve better results for the performance. In Figure 6.4, we show the measured execution time for the same expressions that we ran for the previous implementation. The results include two additional test expressions in order to show that the expected runtime with respect to the number of used combinators is nearly linear.

6.1.3 Conclusion and Similar Approaches

In this section we presented the usage of lenses for a new approach to specify printers and parsers in one definition. The resulting performance is still in need of improvement, but the implementation is sufficient for a first prototype. This approach is a big selling point for nondeterministic lenses. The nondeterministic parsing direction helps us to achieve a meaningful lens definition in both the get and the put direction.

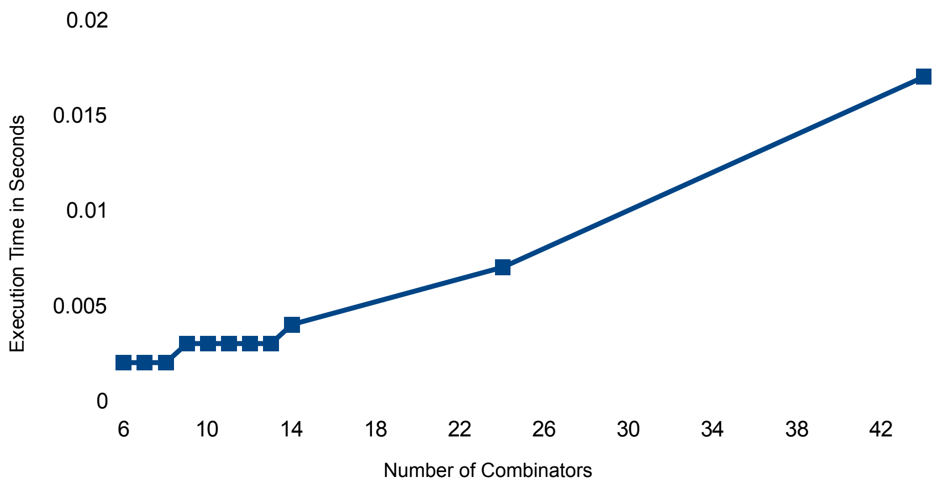

Figure 6.4.: Performance of replacing for an increasing number of combinators

Several papers dealing with bidirectional programming mention the usage of lenses to unify the definition of printers and parsers. However, there is only one actual implementation that realises this specification with the help of lenses. The approach is followed by Zaytsev (2014) and Zaytsev and Bagge (2014), respectively. The first publication is a conglomerate of case studies in the field of bidirectionalisation that also includes a section about printers and parsers. The second publication focuses on parsing and *unparsing*, in which the authors illustrate a so-called megamodel of parsing. This megamodel includes all different artefacts, and the corresponding mappings of one artefact to another. As a result of this investigation, the authors apply different bidirectionalisation techniques in order to implement parsing and unparsing in the meta-programming language Rascal (2011). The mappings include similar features that are available in our implementation: layout-preservation, parsing, unparsing, and redundancies in the parsing direction. In addition, the authors discuss a feature to automatically correct misspellings and likewise parse errors.

Other related work includes the idea of Rendel and Ostermann (2010), who propose a new interface to describe parsers and pretty-printers in a single program. They provide a type class *Syntax* δ that includes common parser functions like *empty*, *pure*, $(<\ \$\ >)$, $(< * >)$, and $(< | >)$. Due to the usage of pretty-printers, the composition combinator, $(< * >)$, forms a pair instead of using the traditional composition semantic of parsers – like in our implementation. In order to define parsers and pretty-printers, the programmer defines an instance for the *Syntax* typeclass. In the end, the implemented instance decides if the combinator behaves like a pretty-printer, or a parser. This approach has similarities to using a pair of functions to represent lenses: we still have to define and maintain both sides of the implementation. Thus, we see an advantage of our implementation in comparison to the idea of Rendel and Ostermann.

Last but not least, we want to mention the publication of Matsuda and Wang (2013b), who introduce a *FlipPr*, a program transformation system that can be used to define pretty-printers and gain a corresponding parser. FlipPr produces a parser for a context free grammar that is consistent with the given definition of the pretty-printer. Their advantage in comparison to bidirectional approaches is that they can outsource their parsing algorithms to parser generators and reuse efficient implementations of exisiting pretty-printer libraries. In particular, FlipPr is implemented in Haskell and reuses Wadler's existing implementation of pretty-printers, additionally, they make use of *happy*[8] as the parser generator.

6.2 Case Study II - Lenses for Records

This section gives an overview about record syntax in Curry and a draft proposal for a possible improvement concerning lenses. First, we discuss the current implementation of records in KiCS2 that covers the general idea, and some insights of the transformations that take place during compilation. As a second step, we take a closer look at these transformations, and the usage of records. The similarity of the challenges connected to records and the use-cases for lenses leads to the idea to translate record fields into lenses. In the last subsection, we sketch the transformations from record type declarations into corresponding lens definitions.

[8] `http://www.haskell.org/happy/`

6.2.1 Record Syntax in Curry

In the current KiCS2 implementation (2014), we can define types similar to data type declarations as records in Haskell. In the remainder of this section, we call these definitions record types. As an exemplary definition of a record types, we define a data structure with two fields named *Person*. A *Person* has a first and a last name, both are represented as *String*s. Fields of the same type can be grouped like in the following example.

```
type Person = {first, last :: String}
type Contact = {person :: Person, street :: String}
```

At first, the compiler desugars record declarations to data type definitions. The value constructor's name is identical to the name of the record type and this constructor takes as many arguments as fields exist in the record declarations.

```
data Person = Person String String
data Contact = Contact Person String
```

Furthermore, the field name accessors are labels that are only known in combination with special syntactical constructs, which are :> and :=. That is, we can define a value of type *Person* with record notation, where := is used to assign a field to a value within record construction and :> is an accessor that we can use to get a value out of a record.

```
aPerson :: Person
aPerson = {first := "Bob", last := "Dylan"}

> aPerson :> first
"Bob"

> aPerson :> last
"Dylan"
```

Record updates are also possible; in order to update a given record type *Person* with fields *first* and *last*, we use another special syntax operator, {_ := _ | _}, to annotate which record value, and which record fields of a record we want to change.

```
appendToFirst :: Person → Person
appendToFirst person = {first := value ++ "1" | person}
  where value = person :> first
```

The construction without the pipe operator looks like a normal record definition. In combination with the pipe, we can update a record value that is given to the right of the operator. In the example, we have a record value with two fields, but only one field is explicitly set to the left of the pipe operator. Record updates allow the programmer to only write down the fields to be updated for a given record; all other fields remain unchanged.

The usage of records can be very helpful and elegant, but has its downsides as well. In contrast to Haskell, the fields *first* and *last* of the *Person* type are not functions, but syntactical constructs called labels. The advantage of these labels is that the name spaces of function names and labels are disjoint. That is, we can define functions *first* and *person* without interfering with our existing record fields.

$$\begin{aligned}&\mathit{first} :: \mathit{Person} \to \mathit{String}\\&\mathit{first}\ p = p :> \mathit{first}\\&\mathit{person} :: \mathit{Contact} \to \mathit{Person}\\&\mathit{person}\ c = c :> \mathit{person}\end{aligned}$$

We postpone an example that highlights one of the disadvantages of labels in comparison with functions to the next subsection.

6.2.2 Step by Step: From Records to Lenses

In order to examine records a little bit further, let us define more complex record field accessors for nested record definitions. For example, the record type *Contact* contains a field *person* of type *Person*, which is also a record type itself with fields *first* and *last*, both of type *String*.

Getting There is Half the Fun

We can define a function *getFirstForContact* that takes a value of type *Contact* as argument and yields a *String* as result. The resulting string is the first name of the person of the given contact, i.e., we first access the field *person* and use the resulting record type *Person* to access the field *first*.

$$\begin{aligned}&\mathit{getFirstForContact} :: \mathit{Contact} \to \mathit{String}\\&\mathit{getFirstForContact}\ \mathit{contact} = \mathit{contact} :> \mathit{person} :> \mathit{first}\end{aligned}$$

The definition of this function looks straightforward and quite compositional, but since *person* and *first* are not real functions but labels, i.e., special syntactical constructs, we cannot compose these accessors like in Haskell. In order to make this point more clear, we define a second version *getFirstForContact′*, which is defined with the help of the functions *first* and *person* that we defined earlier.

$$\begin{aligned}&getFirstForContact' :: Contact \to String\\&getFirstForContact' = first \circ person\end{aligned}$$

In Curry, we cannot apply well-known simplification mechanisms, e.g., eta-reduction, or point-free style, for record accessors.

Set Your Records Straight

In a second step, we define a function to change the value of a field in a given record. Thus, we define a function *setFirstForContact*, which takes a *Contact* and a *String* as arguments in order to change the first name of the person within that contact.

$$\begin{aligned}&setFirstForContact :: Contact \to String \to Contact\\&setFirstForContact\ contact\ name =\\&\quad \{person := \{first := name \mid contact :> person\} \mid contact\}\end{aligned}$$

For this function definition, we need to update two record values: the person within the given contact, and the first name of that person. Therefore, we need to access the value of the field *person* of the given contact, i.e., *contact :> person*, in order to modify the *first* field. The resulting modified value of type *Person* is the new value of the *person* field for the given contact. As the record gets more and more nested, the more complex is the update mechanism. Similar as for the selection function, we take a try to simplify the update function as well. First, we define two auxiliary functions *first′* and *person′* that update the field corresponding to their names for a given value of type *Person* and *Contact*, respectively.

$$\begin{aligned}&first' :: Person \to String \to Person\\&first'\ p\ new = \{first := new \mid p\}\\&person' :: Contact \to Person \to Contact\\&person'\ c\ new = \{person := new \mid c\}\end{aligned}$$

With the help of the selection functions *person* and *first* to access the person within a contact and the first name of a person, respectively, we can redefine the setter function above.

```
setFirstForContact' :: Contact → String → Contact
setFirstForContact' c new = person' (person c) (first' (first (person c)) new)
```

The new version of the nested setter functions looks a little bit less complicated, but it seems time-consuming to define all these auxiliary functions for all record types that we define in a program. Thus, as a next step, we try to generalise the defined get and set function to work for all record types.

Make You a Lens for a Greater Good

We define a function *get* :: (*rec* → *recField*) → *rec* → *recField*, where *rec* is a record type and *recField* is the type of a field of that record.

```
type Get a b = a → b
get :: Get a b → a → b
get getF value = getF value
```

As we have seen above, it is easy to compose getters for nested record values; with the new defined *get* function, we can access a field of a record value as follows[9].

```
personGet :: Get Contact Person
personGet c = c :> person
firstGet :: Get Person String
firstGet p = p :> first
aContact :: Contact
aContact = {person := aPerson, street := "Folkstreet 1969"}
> get personGet aContact
Person "Bob" "Dylan"
> get (firstGet ∘ personGet) aContact
"Bob"
```

[9]As we stated before, KiCS2 translates a record type into a data type declaration, that is, the REPL uses this translated data type when printing a record value.

For a generalised setter function, we define $set :: (rec \rightarrow recField \rightarrow rec) \rightarrow rec \rightarrow recField \rightarrow rec$ and we use the type variables, again, as descriptive names.

```
**type** $Set\ a\ b = a \rightarrow b \rightarrow a$
$set :: Set\ a\ b \rightarrow a \rightarrow b \rightarrow a$
$set\ setF\ value\ new = setF\ value\ new$
```

When revising the setter function for the nested record value, we come to the conclusion that we cannot compose two setters in the same smooth way as the getters. Let us try to define a combinator $(<\,.\,>) :: Set\ a\ b \rightarrow Set\ b\ c \rightarrow Set\ a\ c$ anyway that takes two setter functions and yields a new, combined setter.

```
$(<\,.\,>) :: Set\ a\ b \rightarrow Set\ b\ c \rightarrow Set\ a\ c$
$(fAB <\,.\,> fBC)\ valA\ valC =$
  let $newB = fBC\ valB\ valC$
      $valB = \bot$
  in $fAB\ valA\ newB$
```

The second setter function $fBC :: b \rightarrow c \rightarrow b$ yields a value of type b, which is the same type the first setter function $fAB :: a \rightarrow b \rightarrow a$ takes as its second argument. That is, we can combine the given value $valA :: a$, and the result of the first setter to get a new value $newB :: b$. The setter function fBC takes a value of type b and one of type c as its arguments. We have $valC :: c$ as an argument, so the only missing piece is a value of type b. The two setter function alone cannot be combined in a meaningful way; we need the corresponding getter function $getAB :: a \rightarrow b$ to fill the missing piece. Thus, we add another argument to complete the definition.

```
$(<\,.\,>) :: (Get\ a\ b, Set\ a\ b) \rightarrow Set\ b\ c \rightarrow Set\ a\ c$
$((getAB, setAB) <\,.\,> setBC)\ valA\ valC =$
  let $newB = setBC\ valB\ valC$
      $valB = getAB\ valA$
  in $setAB\ valA\ newB$
```

With the new definition of the combinator, we change the first name of a contact as before, but, in addition, we gain a general mechanism to change any field of a record.

$personSet :: Set\ Contact\ Person$
$personSet\ c\ newP = \{person := newP \mid c\}$
$firstSet :: Set\ Person\ String$
$firstSet\ p\ newF = \{first := newF \mid p\}$
$> set\ ((personGet, personSet) < . > firstSet)\ aContact$ `"Bobby"`
$Contact\ (Person$ `"Bobby" "Dylan"`$)$ `"Folkstreet 1969"`

In the last step, we change the second argument to match the type of the first: we take two pairs, where the first component is a getter and the second component is a setter function. This change leads to a resulting type of pairs as well. That is, we can define both compositions of getter and setter functions in one combinator. In the end, we get the following definition of $(< . >)$.

$(< . >) :: (Get\ a\ b, Set\ a\ b) \to (Get\ b\ c, Set\ b\ c) \to (Get\ a\ c, Set\ a\ c)$
$((getAB, setAB) < . > (getBC, setBC)) = (getAC, setAC)$
  where
    $getAC = getBC \circ getAB$
    $setAC\ valA = setAB\ valA \circ setBC\ (getAB\ valA)$

The attentive reader may recognise the structure: it looks exactly like our primitive lens definitions from Section 5.1.1. This observation leads to the idea of a new transformation of record declarations in Curry, which we discuss in the next subsection.

6.2.3 Record Transformation

Instead of introducing special syntactical constructs like $rec :> recField$ to select, and $\{recField := newValue \mid rec\}$ to update a record field for a given record, we use lenses as a general mechanism. As a bonus, nested record updates gain a general combinator to change a deep nested record field more easily.

In order to give a better insight about this idea, we first give the Curry code we want to generate.

type $Contact = \{person :: Person, street :: String\}$
type $Person = \{first, last :: String\}$

```
  -- generated code
data Contact = Contact Person String
data Person = Person String String
person :: (Contact → Person, Contact → Person → Contact)
person = (personGet, personSet)
  where
    personGet (Contact p _) = p
    personSet (Contact _ s) newP = Contact newP s
first :: (Person → String, Person → String → Person)
first = (firstGet, firstSet)
  where
    firstGet (Person f _) = f
    firstSet (Person _ l) newF = Person newF l
```

As a first observation, we can rewrite the type signature of *person* to highlight the similarity to the definitions above; we generalise this type signature again and define a type synonym *Lens a b* for a less verbose type signature in future code examples.

```
  -- person :: (Contact -> Person, Contact -> Person -> Contact)
  -- person :: (Get Contact Person, Set Contact Person)
person :: Lens Contact Person
type Lens a b = (Get a b, Set a b)
```

Next, we examine the generated code a bit more. As in the current transformation of record types, we generate a data type declaration corresponding to the record type: one value constructor with the same name as the record type, and each field of the record type as an argument of the value constructor. The arrangement of arguments is adopted from the record declaration; in the process, we desugar grouped fields and write every pair of fields and type declaration consecutively. In the following, we call the desugared version of a record type flattened. For example, the record type

```
type Complicated = {first, second        :: Int
                   , onOff               :: Bool
                   , text1, text2, text3 :: String}
```

can be easily flattened to the following record type declaration.

$$
\begin{aligned}
\textbf{type}\ \mathit{Person} = \{ & \mathit{first} && :: \mathit{Int}\\
, & \mathit{second} && :: \mathit{Int}\\
, & \mathit{onOff} && :: \mathit{Bool}\\
, & \mathit{text1} && :: \mathit{String}\\
, & \mathit{text2} && :: \mathit{String}\\
, & \mathit{text3} && :: \mathit{String}\}
\end{aligned}
$$

We resign to give a general approach to flatten a record type, because it is rather technical to write down, and of little help. Hence, in the following we use flattened record types to simplify the transformation without losing expressiveness. For a given record type declaration **type** $\mathit{Rec}\ \alpha_1\ \ldots\ \alpha_n = \{f_1 :: \tau_1,\ \ldots\ , f_k :: \tau_k\}$, we get the following transformation rule.

Transformation 1.

$$
\frac{\tau_1\ \ldots\ \tau_k,\ \alpha_1\ \ldots\ \alpha_n \in \Theta \qquad \mathit{Rec} \notin \Theta \qquad \mathit{Rec} \notin \Psi}{(\Theta, \Psi) : \textbf{type}\ \mathit{Rec}\ \alpha_1\ \ldots\ \alpha_n = \left\{ \begin{array}{l} f_1 :: \tau_1 \\ ,\ \ldots \\ ,\ f_k :: \tau_m \end{array} \right\} \rightsquigarrow \begin{array}{l} \textbf{data}\ \mathit{Rec}\ \alpha_1\ \ldots\ \alpha_n = \\ \quad \mathit{Rec}\ \tau_1 \cdots \tau_k \end{array}}
$$

As a precondition for this transformation rule, we demand that the types, which we use in the record definition, are known types of the given environment. We denote the set of known types as Θ. Furthermore, since the name of the defined record type is used as the name for the generated data type and constructor, the name has to be unique in the given environment as well. That is, *Rec* is neither allowed to be an element of the set of types Θ nor an element of the set of constructors Ψ.

Transformation 2.

$$
\frac{f_1,\ \ldots\ , f_k \notin \Phi \qquad \textbf{type}\ \mathit{Lens}\ a\ b = (a \to b, a \to b \to a)}{\Phi : \textbf{type}\ \mathit{Rec}\ \alpha_1\ \ldots\ \alpha_n = \left\{ \begin{array}{l} f_1 :: \tau_1 \\ ,\ \ldots \\ ,\ f_k :: \tau_k \end{array} \right\} \rightsquigarrow \begin{array}{l} f_1 :: \mathit{Lens}\ (\mathit{Rec}\ \alpha_1\ \ldots\ \alpha_n)\ \tau_1 \\ f_1 = (f_{get_1}, f_{set_1}) \\ \quad \textbf{where}\ (*) \\ \qquad \vdots \\ f_k :: \mathit{Lens}\ (\mathit{Rec}\ \alpha_1\ \ldots\ \alpha_n)\ \tau_k \\ f_k = (f_{get_k}, f_{set_k}) \\ \quad \textbf{where}\ (*) \end{array}}
$$

$$\begin{aligned} &f_{get_i}\ (Rec\ \_\ \cdots\ val_i\ \cdots\ \_) && = val_i && (*)\\ &f_{set_i}\ (Rec\ val_1\ \cdots\ val_k)\ val_{new} && = Rec\ val_1\ \cdots\ val_{i-1}\ val_{new}\ val_{i+1}\ \cdots\ val_k && (*) \end{aligned}$$

The second transformation generates the corresponding lens function for every field of a given record. We demand all record field names to be unique in the given environment: functions with the same name are not allowed. Therefore, we introduce Φ as the set of all function names. If any record field is an element of Φ, the precondition is not fulfilled, thus, the transformation cannot be pursued and fails. In order to make the derivation rule more readable, we introduce a type synonym for lenses **type** $Lens\ a\ b = (a \to b, a \to b \to a)$ for further usage, but we do not generate the lens type synonym in our record transformation for simplicity reasons. For every record field f_i, we generate a lens f_i in two steps: first, we define two local functions f_{get_i} and f_{set_i}. Second, we combine these functions to a pair of getter and setter function and gain a lens definition.

7 Conclusion

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

ALAN TURING

We want to conclude this thesis with a summary of our work, the highlights of our accomplished results, and an outlook on future work. Upcoming challenges include further research on Curry's built-in search component, and further improvements of our lens implementation. Additionally, we propose the integration of lenses in the KiCS2 compiler in the context of record type declarations as well as the reimplementation of the `WUI`[1] library.

7.1 Summary and Results

This thesis addresses the topic of bidirectional programming and lenses in particular. Even though this topic has been investigated in great detail in the past, we gain a new view on lenses by using a functional logic programming language like Curry.

Related approaches concentrate on defining a new infrastructure that fits bidirectional programming perfectly – programming languages like Boomerang and VDL – or targets a specific domain – lenses for relations, strings, or trees. In this thesis, we do not create a new programming language, but use Curry and leverage its capabilities

[1] `http://www-ps.informatik.uni-kiel.de/kics2/lib/CDOC/WUI.html`

regarding nondeterminism, and the built-in search to gain a new bidirectionalisation approach for lenses. Due to its similarities to Haskell, our approach in Curry affords a familiar setting for programmers of both languages. Furthermore, we can have a wide range of lens definitions that are not limited to a specific context, but use all facets of the underlying language, e.g., algebraic data types, higher-order functions, recursion, and also existing libraries.

There also exists promising and well-studied work on bidirectionalisation techniques for get-based lenses that can be used in Haskell. In this thesis, we decided not to follow this traditional approach, but adopt the idea of put-based lenses. Whereas the get-based approach lacks an unique bidirectionalisation of a suitable put function, we have the possibility to formulate a sophisticated update strategy in the put-based approach. We have implemented two libraries for put-based lenses: one library pursues a combinatorial approach; the other library follows the idea of bidirectionalisation in the broadest sense and generates a corresponding get function on the fly. Though the combinatorial approach guarantees well-behavedness of the underlying lenses, we prefer the usage of the second library for two reasons. Firstly, we are not limited to use a predefined set of combinators. Secondly, we had a hard time to get our head around defining more complex lenses with the predefined combinators; the adopted interface was not very intuitive to use in practice.

Moreover, we developed a new notion of nondeterministic lenses and adopted the existing lens laws to be suitable for a nondeterministic setting. In our opinion, nondeterministic lenses enhance the application of bidirectional programming to new areas that were not applicable before. We implemented prototypical lenses to unify the specification of pretty-printers and parsers. On top of these printer-parsers, we developed lenses to facilitate a layout-preserving replacement for a given pretty-printer specification.

Another well-suited application for lenses are record type declarations. We proposed a concept for transforming record type declarations in Curry into a set of lenses. We define a lens for each field of the given record to provide a selector, and an update function by using get and put, respectively.

As a side-product of testing our implementation, we reactivated the testing framework `EasyTest`. We developed our own testing interface for lens laws to generate testing values based on `EasyTest` as well as an automated test generator.

7.2 Future Directions

We have several topics for future work in mind. Firstly, due to our heavy usage of Curry's built-in search, we ran into some complications for function definitions that are too strict. These complications are not lens-specific, but a general problem worth investigating. Secondly, we discuss future work in the context of lenses that include enhancements, and further applications.

7.2.1 Strictness Problems When Searching For Solutions

During the testing phase of our put-based lens libraries, we ran into complications due to Curry's internal structure of integer values in combination with lists. We can isolate these complications to a general problem with the built-in search: we cannot guess an argument of a function that evaluates in a strict manner, i.e., the argument needs to be evaluated completely and no information about the result can be propagated beforehand.

In this subsection, we give a detailed insight of Curry's built-in search capabilities with an example that needs to guess a list with a specific length. We also present two approaches to solve the upcoming problem. Both approaches try to harmonise the combination of inter values and lists. The first approach uses an alternative structure for lists; the second approach uses an alternative structure for integer values. However, we think it would be an interesting topic of its own to investigate function definitions that are too strict to be applicable for the built-in search component.

In the following, we work with the put-based lens library that we presented in Section 5.2. As a quick reminder: we have the following interface for put-based lenses that generate a corresponding *get*-function.

```
type Lens a b = a → b → a

put :: Lens a b → a → b → a
put lens s v = lens s v

get :: Lens a b → a → b
get lens s | put s v ≡ s = v
  where v free
```

In order to compare our lenses with related approaches, we want to define equivalent lenses to the ones presented by Voigtländer (2009). We pick *putHalve* as exemplary lens definition, which is defined as follows.

$$\begin{aligned}&\mathit{putHalve} :: [\,a\,] \to [\,a\,] \to [\,a\,]\\&\mathit{putHalve}\ \mathit{xs}\ \mathit{xs}' \mid \mathit{length}\ \mathit{xs}' \equiv n = \mathit{xs}' +\!\!+\ \mathit{drop}\ n\ \mathit{xs}\\&\quad \textbf{where}\ n = \mathit{length}\ \mathit{xs}\ \text{`}\mathit{div}\text{`}\ 2\end{aligned}$$

putHalve takes two lists – a source list and a view list – and concatenates the second list with the second half of the first list. Valid view lists have half the length of the source list, if otherwise, the function yields *failed*, i.e., no result is produced.

In the following, our goal is to generate a get function for *putHalve*. The desired *get* function is equivalent to the following definition of *halve*.

$$\begin{aligned}&\mathit{halve} :: [\,a\,] \to [\,a\,]\\&\mathit{halve}\ \mathit{xs} = \mathit{take}\ (\mathit{length}\ \mathit{xs}\ \text{`}\mathit{div}\text{`}\ 2)\ \mathit{xs}\end{aligned}$$

With the help of our interface, we can derive a corresponding get function by simply calling *get* with *putHalve* and our source value.

$$\mathit{getHalve} = \mathit{get}\ \mathit{putHalve}$$

Unfortunately, this easy task evolved into a long investigation of Curry's built-in search mechanism as well as the internal representation of integer values, and the interaction of integer values with lists.

The Problem

First of all, a function call like *getHalve* $[(),()]$ does not terminate. The problem is the combination of the internal representation of lists and integer values; they do not harmonise well. This effect is triggered by the usage of *length*. The free variable v in the definition of *get* corresponds to xs' in the definition of *putHalve*, i.e., the system guesses values for xs'. In order to be more precise, the system needs to guess lists of type () for xs' and checks if their length is the same as *length* $[(),()]$ `*div*` 2. We can evalute the expression *getHalve* $[(),()]$ as follows.

$$\begin{aligned}&\ \ \mathit{getHalve}\ [(),()]\\&\equiv \mathit{get}\ \mathit{putHalve}\ [(),()]\\&\equiv \mathit{putHalve}\ [(),()]\ v \equiv [(),()]\ \textbf{where}\ v\ \mathit{free}\end{aligned}$$

$\equiv \mathit{putHalve}\ [(),()]\ v \mid \mathit{length}\ v \equiv n = v +\!\!+\ \mathit{drop}\ n\ [(),()]$
$\quad\quad \textbf{where}\ n = \mathit{length}\ [(),()]\ `\mathit{div}`\ 2$
$\equiv \mathit{putHalve}\ [(),()]\ v \mid \mathit{length}\ v \equiv n = v +\!\!+\ \mathit{drop}\ n\ [(),()]$
$\quad\quad \textbf{where}\ n = 1$
$\equiv \mathit{putHalve}\ [(),()]\ v \mid \mathit{length}\ v \equiv 1 = v +\!\!+\ [(),()]$

In order to focus on the cause of the problem, we reduce our definition of *putHalve* to this explicte example of a list with two elements.

$\mathit{putHalveSimple} :: [a] \to [a]$
$\mathit{putHalveSimple}\ \mathit{xs'} \mid \mathit{length}\ \mathit{xs'} \equiv 1 = \mathit{xs'} +\!\!+\ \mathit{drop}\ 1\ [(),()]$

If we evaluate an expression with a free variable, e.g., *putHalveSimple v* **where** *v free*, Curry's built-in search component converts integer value to its internal representation. In particular, Curry uses binary numbers as representation for integer values, plus additional information about the algebraic sign to represent negative values, positive values, and 0. The data structure *Nat* defines constructors for binary numbers and *BinInt* is the overall representation, wrapping the binary numbers.

$\textbf{data}\ \mathit{BinInt} = \mathit{Neg}\ \mathit{Nat} \mid \mathit{Zero} \mid \mathit{Pos}\ \mathit{Nat}$
$\textbf{data}\ \mathit{Nat} = \mathit{IHi} \mid O\ \mathit{Nat} \mid I\ \mathit{Nat}$

If we replace the usage of integer values in our defintiion of *putHalveSimple* with the internal representation of *BinInt*, we end up with the following function.

$\mathit{putHalveSimple} :: [a] \to [a]$
$\mathit{putHalveSimple}\ \mathit{xs'} \mid \mathit{length}\ \mathit{xs'} \equiv \mathit{Pos}\ \mathit{IHi} = \mathit{xs'} +\!\!+\ \mathit{drop}\ 1\ [(),()]$

$\mathit{length'} :: [a] \to \mathit{BinInt}$
$\mathit{length'}\ [\,] \qquad\quad = \mathit{Zero}$
$\mathit{length'}\ (x : \mathit{xs}) = \mathit{inc}\ (\mathit{length'}\ \mathit{xs})$

Additionally, we use a function *length′* that computes the length of a given list using *BinInt* as well. The auxiliary function *inc* increments a *BinInt* value.

$\mathit{inc} :: \mathit{BinInt} \to \mathit{BinInt}$
$\mathit{inc}\ \mathit{Zero} \qquad\qquad = \mathit{Pos}\ \mathit{IHi}$
$\mathit{inc}\ (\mathit{Pos}\ n) \qquad\ = \mathit{Pos}\ (\mathit{succ}\ n)$
$\mathit{inc}\ (\mathit{Neg}\ \mathit{IHi}) \quad\ = \mathit{Zero}$
$\mathit{inc}\ (\mathit{Neg}\ (O\ n)) = \mathit{Neg}\ (\mathit{pred}\ (O\ n))$
$\mathit{inc}\ (\mathit{Neg}\ (I\ n))\ \ = \mathit{Neg}\ (O\ n)$

Guessing a List with a Specific Length

So, how do the evaluation steps look like when we want to compute the length of a list?

We can directly compute the result for an empty list, but for a non-empty list we build a sequence of *inc*-operations, e.g., *inc* (*inc* *Zero*) for a two-valued list, *inc* (*inc* (*inc* *Zero*)) for a three-valued list etc. In order to take this investigation one step ahead, we need to look at the definition of *inc*, where only lines 2 and 3 are of further interest. We give the evaluation of zero to two consecutive *inc* function calls in Figure 7.1.

$$(1) \qquad \begin{aligned} & \mathit{Zero} \equiv \mathit{Pos}\ \mathit{IHi} \\ \equiv\ & \mathit{False} \end{aligned}$$

$$(2) \qquad \begin{aligned} & \mathit{inc}\ (\mathit{Zero}) \equiv \mathit{Pos}\ \mathit{IHi} \\ \equiv\ & \mathit{Pos}\ \mathit{IHi} \equiv \mathit{Pos}\ \mathit{IHi} \\ \equiv\ & \mathit{True} \end{aligned}$$

$$(3) \qquad \begin{aligned} & \mathit{inc}\ (\mathit{inc}\ \mathit{Zero}) \equiv \mathit{Pos}\ \mathit{IHi} \\ \equiv\ & \mathit{inc}\ (\mathit{Pos}\ \mathit{IHi}) \equiv \mathit{Pos}\ \mathit{IHi} \\ \equiv\ & \mathit{Pos}\ (\mathit{succ}\ \mathit{IHi}) \equiv \mathit{Pos}\ \mathit{IHi} \\ \equiv\ & \mathit{Pos}\ (O\ \mathit{IHi}) \equiv \mathit{Pos}\ \mathit{IHi} \\ \equiv\ & \mathit{False} \end{aligned}$$

Figure 7.1.: Evaluation of *inc* for an increasing number of function calls

The attentive reader may have already noticed that the definition of *inc* is strict and does not propagate its constructor. The successor function on binary numbers, *succ*, is also strict. Hence, in the current implementation with integer values, the list that Curry guesses is evaluated completely due to the usage of *length*. The *length* function evaluates the whole list to determine its length; this leads to the construction of lists with increasing length when using a free variable. The built-in search for free variables in KiCS2 can be translated in nested injections of ?-operations, where every constructor of the given type is a possible guess and arguments of constructors are also free variables. For every free variable of type $[\,a\,]$ both constructors are possible values, therefore, both expressions are introduced with the ?-operator. We illustrate the built-in search of our example in Figure 7.2.

$$\begin{aligned}
&\mathit{length}'\ v \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ v\ \mathit{free}\\
\equiv\ &\mathit{length}'\ ([\,]\ ?\ \_x2 : xs) \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ \_x2, xs\ \mathit{free}\\
\equiv\ &(\mathit{length}'\ [\,]\ ?\ \mathit{length}\ \_x2 : xs) \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ \_x2, xs\ \mathit{free}\\
\equiv\ &\mathit{length}'\ [\,] \equiv \mathit{Pos}\ \mathit{IHi}\ ?\ \mathit{length}'\ \_x2 : xs \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ \_x2, xs\ \mathit{free}\\
\equiv\ &\mathit{Zero} \equiv \mathit{Pos}\ \mathit{IHi}\ ?\ \mathit{length}\ \_x2 : xs \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ \_x2, xs\ \mathit{free}\\
\equiv\ &\mathit{False}\ ?\ \mathit{length}'\ \_x2 : xs \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ \_x2, xs\ \mathit{free}\\
\equiv\ &\mathit{False}\ ?\ \mathit{inc}\ (\mathit{length}'\ xs) \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ xs\ \mathit{free}\\
\equiv\ &\mathit{False}\ ?\ \mathit{inc}\ (\mathit{length}'\ [\,]\ ?\ \mathit{length}'\ \_x4 : ys) \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ \_x4, ys\ \mathit{free}\\
\equiv\ &\mathit{False}\ ?\ \mathit{inc}\ \mathit{length}'\ [\,] \equiv \mathit{Pos}\ \mathit{IHi}\ ?\ \mathit{inc}\ (\mathit{length}'\ \_x4 : ys) \equiv \mathit{Pos}\ \mathit{IHi}\\
&\quad \textbf{where}\ \_x4, ys\ \mathit{free}\\
\equiv\ &\ \ldots
\end{aligned}$$

Figure 7.2.: Guessing a list with length one

The built-in search collects all possible values and works henceforth with a set of values, i.e., every list of the resulting set is used for further function calls. For our example, we get the following results in the interactive environment of KiCS2.

$$\begin{aligned}
&> \mathit{length}'\ v \equiv \mathit{Pos}\ \mathit{IHi}\ \textbf{where}\ v\ \mathit{free}\\
&\{v = [\,]\}\ \mathit{False}\\
&\{v = [\_x2]\}\ \mathit{True}\\
&\{v = [\_x2, \_x4]\}\ \mathit{False}\\
&\quad\vdots
\end{aligned}$$

In summary, the internal structure for lists and numbers do not harmonise well, we cannot guess the length of a list. Unfortunately, the *length* function cannot be implemented in a way that is sufficient to propagate a constructor, because it is problematic to map the empty list to a value of type *Nat*. How can we solve the problem that *putHalve* and *putHalveSimple*, respectively, does not terminate?

Approach One: Peano Numbers

As a first attempt, we use an alternative data structure with an unary representation for integer values: peano numbers.

$$\begin{aligned}\textbf{data}\ \mathit{Peano} &= \mathit{Zero}\\ &\mid S\ \mathit{Peano}\end{aligned}$$

Peano numbers are represented with a constructor for *Zero* and a successor constructor *S Peano*. The corresponding length function, *lengthPeano*, introduces an *S*-constructor for every element of the list and yields *Zero* for an empty list.

$$\begin{aligned}&\mathit{lengthPeano}\ [\,] = Z\\ &\mathit{lengthPeano}\ (x : \mathit{xs}) = S\ (\mathit{lengthPeano}\ \mathit{xs})\end{aligned}$$

Let us take a look at the simplified implementation of *putHalvePeano* that uses peano numbers instead of integer values.

$$\begin{aligned}&\mathit{putHalvePeano} :: [\,a\,] \to [\,a\,]\\ &\mathit{putHalvePeano}\ \mathit{xs}' \mid \mathit{lengthPeano}\ \mathit{xs}' \equiv S\ Z = \mathit{xs}' +\!\!+ [\,()\,]\end{aligned}$$

We evaluate the the expression *lengthPeano* $v \equiv S\ Z$ **where** v *free* in Figure 7.3. The actual evaluation of the given expression yields the following result in KiCS2's interactive environment.

$$\begin{aligned}&> \mathit{lengthPeano}\ v \equiv S\ Z\ \textbf{where}\ v\ \mathit{free}\\ &\{\,v = [\,]\,\}\ \mathit{False}\\ &\{\,v = [\,\_x3\,]\,\}\ \mathit{True}\\ &\{\,v = (\_x3 : \_\ x4 : \_\ x5)\,\}\ \mathit{False}\end{aligned}$$

For this implementation, the evaluation terminates after three steps. The get direction of *putHalve* also yields convenient results: the expanded version of the get direction yields the successful binding; the actual function *getHalve* yields only the resulting value as desired.

$$\begin{aligned}&> \mathit{putHalve}\ [\,(), ()\,]\ v \equiv [\,(), ()\,]\ \textbf{where}\ v\ \mathit{free}\\ &v = [\,()\,]\,\}\ \mathit{True}\end{aligned}$$

$$\begin{aligned}&> \mathit{getHalve}\ [\,(), ()\,]\\ &[\,()\,]\end{aligned}$$

$\mathit{lengthPeano}\ v \equiv S\ Z$ **where** v *free* 1

$\equiv \mathit{lengthPeano}\ ([\,]\ ?\ \_\mathit{x3} : \mathit{xs}) \equiv S\ Z$ **where** $\_\mathit{x3}, \mathit{xs}$ *free* 2

$\equiv (\mathit{lengthPeano}\ [\,]\ ?\ \mathit{lengthPeano}\ (\_\mathit{x3} : \mathit{xs})) \equiv S\ Z$ **where** $\_\mathit{x3}, \mathit{xs}$ *free* 3

$\equiv \mathit{lengthPeano}\ [\,] \equiv S\ Z\ ?\ \mathit{lengthPeano}\ (\_\mathit{x3} : \mathit{xs}) \equiv S\ Z$ **where** $\_\mathit{x3}, \mathit{xs}$ *free* 4

$\equiv Z \equiv S\ Z\ ?\ S\ (\mathit{lengthPeano}\ \mathit{xs}) \equiv S\ Z$ **where** xs *free* 5

$\equiv Z \equiv S\ Z\ ?\ \mathit{lengthPeano}\ \mathit{xs} \equiv Z$ **where** xs *free* 6

$\equiv \mathit{False}\ ?\ \mathit{lengthPeano}\ [\,]\ ?\ \mathit{lengthPeano}\ (\_\mathit{x4} : \_\ \mathit{x5}) \equiv Z$ 7

where $\_\mathit{x4}, \_\mathit{x5}$ *free* 8

$\equiv \mathit{False}\ ?\ \mathit{lengthPeano}\ [\,] \equiv Z\ ?\ \mathit{lengthPeano}\ (\_\mathit{x4} : \_\ \mathit{x5}) \equiv Z$ 9

where $\_\mathit{x4}, \_\mathit{x5}$ *free* 10

$\equiv \mathit{False}\ ?\ Z \equiv Z\ ?\ S\ (\mathit{lengthPeano}\ \_\mathit{x5}) \equiv Z$ **where** $\_\mathit{x5}$ *free* 11

$\equiv \mathit{False}\ ?\ \mathit{True}\ ?\ \mathit{False}$ 12

Figure 7.3.: Guessing a list with length one with peano numbers

The main difference to the first implementation is that *lengthPeano* can propagate the constructor of the underlying data structure – S or Z – to the front of the remaining evaluation. As the guessed list grows, the newly introduced ?-operators occur only as the argument of an S-constructor. This construction finally leads to a terminating search, because once we overreach the number of S-constructors, the remaining argument of that constructor is not evaluated anymore. Lines 12-13 of the example in Figure 7.3 show that no further guesses for free variables are necessary, because the partial evaluation of $S\ n$ can never be evaluated to Z. Hence, the expression $S\ n \equiv Z$ evaluates to *False* and the evaluation of the whole expression terminates.

Approach Two: Binary Lists

The second approach is to choose an alternative representation for lists. In particular, we are interested in a representation that behaves well with the internal *BinInt* data structure. Therefore, we use the following definition of binary lists.

data $L\ a = \mathit{LIHi}\ a \mid \mathit{LO}\ (L\ (a, a)) \mid \mathit{LI}\ (L\ (a, a))\ a$
data $\mathit{BinaryList}\ a = \mathit{Empty} \mid \mathit{NonEmpty}\ (L\ a)$

We define a data structure for non-empty lists that corresponds to binary numbers. *LIHi a* is a list with one element, *LO* (*L* (*a*, *a*)) represents a list with at least two elements, and *LI* (*L* (*a*, *a*)) *a* is the constructor for an at least three-valued list. Since this data structure has no representation for an empty list, we introduce an additional data type *BinaryList* that wraps a constructor *NonEmpty* around the list representation *L a*. The second constructor *Empty* represents empty lists.

```
lengthBList :: BinaryList a → BinInt
lengthBList Empty = Zero
lengthBList (NonEmpty list) = Pos (lengthL list)
  where
    lengthL :: L a → Nat
    lengthL (LIHi _) = IHi
    lengthL (LO l) = O (lengthL l)
    lengthL (LI l _) = I (lengthL l)
```

The definition of the length function *lengthBList* that computes the length of a *BinaryList* with the *BinInt* data structure benefits from underlying structure of the given list. The *BinaryList* has a special constructor for non-empty lists with an inner representation. Therefore, we can propagate *Pos* for a non-empty list without evaluating the actual list, i.e., the argument of the *NonEmpty*-constructor. The three different constructors of the binary list structure reflect which *Nat*-constructor to use, such that the constructor is propagated to the front of the expression, again.

We define a corresponding version *putHalveBinaryList* with our new representation of lists.

```
putHalveBinaryList :: BinaryList a → BinaryList a → BinaryList
putHalveBinaryList xs′
   | lengthBList xs′ ≡ Pos IHi = xs′ ++ listToBinaryList [()]
```

The auxiliary function *listToBinaryList* :: [*a*] → *BinaryList a* converts a given traditional list representation to the corresponding binary list.

Once again, we want to test our solution by running the get direction of the lens for a list of length one. We reduce the query to the expression *lengthBList v* ≡ *Pos IHi* **where** *v free*, which yields the following result.

```
> lengthBList v ≡ Pos IHi where v free
{v = Empty} False
{v = NonEmpty (LIHi _x2)} True
{v = NonEmpty (LO _x2)} False
{v = NonEmpty (LI _x2 _x3)} False
```

The evaluation terminates after computing four values to bind the free variables; a detailed evaluation of the expression is given in Figure 7.4. Lines 18-24 illustrate the propagation of the convenient binary number's constructor for the corresponding constructor of the binary list quite well. In the subsequent lines – 25-28 – the evaluation terminates due the propagated constructors *Pos* (*I* _) and *Pos* (*O* _), respectively, that can be finally compared with *Pos IHi*.

```
  lengthBList v ≡ Pos IHi where v free                                      1
≡ lengthBList (Empty ? NonEmpty xs) ≡ Pos IHi where xs free                 2
≡ (lengthBList Empty ? lengthBList (NonEmpty xs)) ≡ Pos IHi                 3
    where xs free                                                           4
≡ lengthBList Empty ≡ Pos IHi ? lengthBList (NonEmpty xs) ≡ Pos IHi         5
    where xs free                                                           6
≡ Zero ≡ Pos IHi ? Pos (lengthL xs) ≡ Pos IHi where xs free                 7
≡ False ? Pos (lengthL (LIHI _x2 ? LO _x2                                   8
                                 ? LI _x2 _x3)) ≡ Pos IHi                   9
    where _x2, _x3 free                                                     10
≡ False ? Pos (lengthL (LIHI _x2 ? LO _x2                                   11
                                 ? LI _x2 _x3) ≡ Pos IHi                    12
    where _x2, _x3 free                                                     13
≡ False ? Pos (lengthL (LIHi _x2) ? lengthL (LO _x2)                        14
                                 ? lengthL (LI _x2 y)) ≡ Pos IHi            15
    where _x2, _x3 free                                                     16
≡ False ? Pos (lengthL (LIHi _x2)) ≡ Pos IHi                                17
      ? Pos (lengthL (LO _x2) ? lengthL (LI _x2 _x3)) ≡ Pos IHi             18
    where _x2, _x3 free                                                     19
```

```
≡ False ? Pos IHi ≡ Pos IHi
        ? Pos (O (lengthL _x2) ? I (lengthL _x2 _x3)) ≡ Pos IHi
    where _x2, _x3 free
≡ False ? True
        ? Pos (O (lengthL _x2)) ≡ Pos IHi
        ? Pos (I (lengthL _x2 _x3)) ≡ Pos IHi
    where _x2, _x3 free
≡ False ? True ? False ? False
```

Figure 7.4.: Guessing a binary list with length one

In the end, the expression *get putHalveBinaryList* (*NonEmpty* (*LO* (*L* ((), ())))) yields *NonEmpty* (*LIHi* ()) as desired.

7.2.2 Further Directions

Unfortunately, our preferred put-based lens library does not guarantee well-behavedness by construction. This lack of well-behavedness has to be tackled in the future. We started by defining a test-suite that generates test cases for all lens definitions of a given module, but the tests still have to be run manually. It would be useful to have static analyses as a replacement for manual tests. Hu et al. (2014) propose two algorithms to check the two essential laws for put-based lenses – PutDet and PutStab – statically. However, they define these algorithms on top of a simple, self-defined language for lenses. This language allows only put definitions that are affine and in treeless form, thus, we cannot directly apply their results in Curry. Nevertheless, we think that their work is a good starting point to get ideas for statical analyses in the context of put-based lenses.

Due to the scope of this thesis, we had to lower our sights regarding record transformations. In this thesis, we proposed a series of transformations on record type declarations to generate lenses as convenient getter and setter functions instead of the current implementation with syntactical constructs. We have also implemented a prototype that works on the internal FlatCurry[2] representation of Curry programs as proof of concept. However, we would like to integrate these transformations into the KiCS2 compiler, and provide a simple lens library with a handful of primitives. For the purpose of these field accessors, it suffices to implement a simple representation of lenses as a pair of getter and setter functions.

Last but not least, we think that the work of Rajkumar et al. (2013) regarding lenses in the context of web development could be very applicable for Curry. In our opinion, lenses perfectly fit the setting of mapping database entities to user interfaces for two reasons. Firstly, these mappings usually project from database entities to user interfaces, which is a simple and common application for lenses. Secondly, possible performance issues of lenses have a smaller impact in the context of web development, where performance is usually affected by a communication overhead. Curry already provides a library called `WUI` to specify web user interfaces; it was implemented by Hanus (2006). On top of that, the *Spicey*[3] framework can generate an initial setup of a web-based system from an entity-relationship description of the underlying data. We have already reimplemented the `WUI` library to use lenses and changed the *Blog example* provided by Spicey to use our reimplementation. As a future work, we propose to integrate the lens component into the `WUI` specification and the Spicey framework in particular.

[2]`http://www.informatik.uni-kiel.de/~curry/flat/`
[3]`http://www.informatik.uni-kiel.de/~pakcs/spicey/`

Bibliography

Michael Abott, Thorsten Altenkirch, and Neil Ghani. Categories of Containers. In *Proceedings of Foundations of Software Science and Computation Structures*, 2003.

S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proc. of the 11th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.

F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Trans. Database Syst.*, 6(4):557–575, December 1981. ISSN 0362-5915. doi: 10.1145/319628.319634. URL `http://doi.acm.org/10.1145/319628.319634`.

Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching Lenses: Alignment and View Update. In *ICFP*, pages 193–204, 2010.

Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational Lenses: A Language for Updatable Views. In *PODS*, pages 338–347, 2006.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful Lenses for String Data. In *POPL*, pages 407–419, 2008.

Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. KiCS2: A New Compiler from Curry to Haskell. In Herbert Kuchen, editor, *WFLP*, volume 6816 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011. ISBN 978-3-642-22530-7. URL `http://dblp.uni-trier.de/db/conf/wflp/wflp2011.html#BrasselHPR11`.

Rafael Caballero and Francisco Javier Lòpez-Fraguas. A Functional-Logic Perspective on Parsing. In Aart Middeldorp and Taisuke Sato, editors, *Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 1999. ISBN 3-540-66677-X. URL `http://dblp.uni-trier.de/db/conf/flops/flops99.html#CaballeroL99`.

Jan Christiansen and Sebastian Fischer. EasyCheck - Test Data for Free. In *FLOPS*, pages 322–336, 2008.

Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *ICMT*, pages 260–283, 2009.

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From State- to Delta-Based Bidirectional Model Transformations. In *ICMT*, pages 61–76, 2010.

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology*, 10:6: 1–25, 2011a.

Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In *MoDELS*, pages 304–318, 2011b.

Sebastian Fischer, Hugo Pacheco, and Zhenjiang Hu. 'Putback' is the Essence of Bidirectional Programming. Technical report, National Institute of Informatics, 2012. available at: `http://grace-center.jp/wp-content/uploads/2013/01/GRACE-TR-2012-08.pdf`.

Sebastian Fischer, Hugo Pacheco, and Zhenjiang Hu and. Monadic Combinators for 'Putback' Style Bidirectional Programming. In *PEPM*, pages 39–50, 2014.

J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. *Harmony Programmer's Manual*, 2006.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient Lenses. In *ICFP*, pages 383–396, 2008.

Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three Complementary Approaches to Bidirectional Programming. In *SSGIP*, pages 1–46, 2010.

Michael Hanus. Type-Oriented Construction of Web User Interfaces. In *PPDP*, pages 27–38, 2006.

Michael Hanus, Bernd Braßel, Björn Peemöller, and Fabian Reck. KiCS2 - The Kiel Curry System User Manual Version 0.3.2 of 2014-07-30, 2014. URL `http://www-ps.informatik.uni-kiel.de/kics2/Manual.pdf`.

Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In *POPL*, pages 371–384, 2011.

Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Edit Lenses. In *POPL*, pages 495–508, 2012.

Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. Validity Checking of Putback Transformations in Bidirectional Programming. In *FM*, pages 1–15, 2014.

John Hughes and Koen Claessen. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP 2000, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351266. URL `http://doi.acm.org/10.1145/351240.351266`.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* http://haskell.org/, September 2002. URL `http://haskell.org/definition/haskell98-report.pdf`.

Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '85, pages 154–163, New York, NY, USA, 1985. ACM. ISBN 0-89791-153-9. doi: 10.1145/325405.325423. URL `http://doi.acm.org/10.1145/325405.325423`.

Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In *Generative and Transformational Techniques in Software Engineering III*, pages 222–289. Springer Berlin Heidelberg, 2011.

Kazutaka Matsuda and Meng Wang. Bidirectionalization for Free with Runtime Recording: Or, a Light-weight Approach to the View-update Problem. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 297–308, New York, NY, USA, 2013a. ACM. ISBN 978-1-4503-2154-9. doi: 10.1145/2505879.2505888. URL `http://doi.acm.org/10.1145/2505879.2505888`.

Kazutaka Matsuda and Meng Wang. FliPpr: A Prettier Invertible Printing System. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 101–120, Berlin, Heidelberg, 2013b. Springer-Verlag. ISBN 978-3-642-37035-9. doi: 10.1007/978-3-642-37036-6_6. URL `http://dx.doi.org/10.1007/978-3-642-37036-6_6`.

Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization Transformation Based on Automatic Derivation of View Complement Functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 47–58, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291162. URL `http://doi.acm.org/10.1145/1291151.1291162`.

Lambert Meertens. Designing Constraint Maintainers for User Interaction. Technical report, 1998.

Hugo Pacheco and Alcino Cunha. Generic Point-free Lenses. In *Proceedings of the 10th International Conference on Mathematics of Program Construction*, MPC'10, pages 331–352, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13320-7, 978-3-642-13320-6. URL `http://dl.acm.org/citation.cfm?id=1886619.1886640`.

Hugo Pacheco, Alcino Cunha, and Zhenjiang Hu. Delta Lenses over Inductive Types. *ECEASST*, 49, 2012.

Hugo Pacheco, Alcino Cunha, Nuno Macedo, and José Nuno Oliveira. Composing Least-Change Lenses. *ECEASST*, 57, 2013a.

Hugo Pacheco, Nuno Macedo, Alcino Cunha, and Janis Voigtländer. A Generic Scheme and Properties of Bidirectional Transformations. *CoRR*, abs/1306.4473, 2013b.

Benjamin C. Pierce, Alan Schmitt, and Michael B. Greenwald. Bringing Harmony to Optimism: A Synchronization Framework for Heterogeneous Tree-Structured Data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003. Superseded by MS-CIS-05-02.

Raghu Rajkumar, Nate Foster, Sam Lindley, and James Cheney. Lenses for Web Data. *ECEASST*, 57, 2013.

Tillmann Rendel and Klaus Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863525. URL `http://doi.acm.org/10.1145/1863523.1863525`.

Janis Voigtländer. Bidirectionalization for Free! (Pearl). In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 165–176, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480904. URL `http://doi.acm.org/10.1145/1480881.1480904`.

Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Combining Syntactic and Semantic Bidirectionalization. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 181–192, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863571. URL `http://doi.acm.org/10.1145/1863543.1863571`.

Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. *Journal of Functional Programming*, 23:515–551, 9 2013. ISSN 1469-7653. doi: 10.1017/S0956796813000130. URL `http://journals.cambridge.org/article_S0956796813000130`.

Philip Wadler. A Prettier Printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.

D. Wagner. *Symmetric Edit Lenses: A New Foundation for Bidirectional Languages*. PhD thesis, University of Pennsylvania, June 2014.

Meng Wang and Shayan Najd. Semantic Bidirectionalization Revisited. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 51–61, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. doi: 10.1145/2543728.2543729. URL `http://doi.acm.org/10.1145/2543728.2543729`.

Vadim Zaytsev. Case Studies in Bidirectionalisation. In *Pre-proceedings of the 15th International Symposium on Trends in Functional Programming (TFP 2014)*, pages 51–58, May 2014. Extended Abstract.

Vadim Zaytsev and Anya Helene Bagge. Parsing in a Broad Sense. Submitted to the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014). Accepted, March 2014.

A

Combinatorial Library: Examples for Lenses

We give some exemplary lens definitions on user-defined data types using the combinatorial library we implemented in Curry. The corresponding implementation can be found in Section 5.1. Most of the following lens definitions are straightforward. In the first example, we use the combinator *isoLens* to define a mapping from our data type to a tuple representation. This tuple representation can be further processed with the combinators of the library. On the hand, we have two examples that work directly on primitive data types. In addition, these two examples include constraints for constructing a value of the data structure. The lens definitions can be seen as smart constructors for these algebraic data types.

Person examples

```
data Person = Person Name City
type First = String
person :: Lens Person (Name, City)
person = isoLens inn out
  where
    inn (n, c) = Person n c
    out (Person n c) = (n, c)
nameOrAddress :: Lens Person String
nameOrAddress = nameLens ? addressLens
nameLens :: Lens Person First
nameLens = person <.> keepSnd
addressLens :: Lens Person Address
addressLens = person <.> keepFst
```

Temperature examples

```
data Temp = Temp Float
centigrade :: Lens Temp Float
centigrade = isoLens inn out
  where
    inn celsius = Temp (cToF celsius)
    out (Temp temp) = fToC temp

cToF :: Float → Float
cToF c = c *. 1.8 +. 32
fToC :: Float → Float
fToC f = (f -. 32) *. (5 /. 9)
```

Time examples

```
data Time = Time Int Int

time :: Lens Time Int
time = isoLens innT (λ(Time hour min) → hour * 60 + min)

mins :: Lens Time Int
mins = isoLens innT (λ(Time _ min) → min)

innT :: Int → Time
innT m = Time (m `quot` 60) (m `mod` 60)
```

B

Printer-Parser for Arithmetic Expressions with Infix Notation

In addition to the presented printer-parser for arithmetic expression in prefix notation in Section 6.1.1, we define a more complex and convenient printer-parser. The following code shows a printer-parser implementation for arithmetic expression in infix notation. Because of the complication with redundant whitespaces in the parsing direction, the implementation eschews this improvement. As a side-effect of integrating parsing techniques into the printer-parser, the implementation needs to avoid left-recursions. Thus, the printer-parser follows the traditional technique for implementing a parser for arithmetic expressions.

```
ppExpr' :: PPrinter Expr
ppExpr' str t@(BinOp op e1 e2, str')
  | op ≡ Plus ∨ op ≡ Minus =
    ((ppTerm <<< whitespace)
       <> ppPlusMinus
       <> (whitespace >>> ppExpr')) str (((e1, op), e2), str')
  | otherwise              = ppTerm str t
ppExpr' str t@(Num _, _) = ppTerm str t
```

```
ppTerm :: PPrinter Expr
ppTerm str f@(BinOp op e1 e2, str')
  | op ≡ Mult ∨ op ≡ Div =
    ((ppFactor <<< whitespace)
      <> ppMultDiv
      <> (whitespace >>> ppTerm)) str (((e1, op), e2), str')
  | otherwise              = ppFactor str f
ppTerm str f@(Num _, _)    = ppFactor str f

ppFactor :: PPrinter Expr
ppFactor str f@(e, str') = case e of
  Num v → digit str (v, str')
  _     → "(" ++ ppExpr' str (e, ")" ++ str')

ppMultDiv :: PPrinter Op
ppMultDiv _ (Mult, str') = "*" ++ str'
ppMultdiv _ (Div, str')  = "/" ++ str'

ppPlusMinus :: PPrinter Op
ppPlusMinus _ (Plus, str')  = "+" ++ str'
ppPlusMinus _ (Minus, str') = "-" ++ str'
```

In order to see the printer-parser in action, we give some exemplary expressions for printing and parsing.

```
> pPrint ppExpr' (BinOp Mult (Num 1) (BinOp Mult (Num 2) (Num 3)))
1 * 2 * 3

> pPrint ppExpr' (BinOp Mult (Num 7) (BinOp Plus (Num 5) (Num 3)))
"7 * (5 + 3)"

> pPrint ppExpr' (BinOp Mult (Num 1) (BinOp Mult (BinOp Plus (Num 2)
(Num 3))(BinOp Minus (Num 5) (Num 1))))
"1 * (2 + 3) * (5 - 1)"

> pParse ppExpr' "1 + 3 * 5"
(BinOp Plus (Num 1) (BinOp Mult (Num 3) (Num 5)))

> pParse ppExpr' "1 + 3 * (4 - 9)"
(BinOp Plus (Num 1) (BinOp Mult (Num 3) (BinOp Minus (Num 4) (Num 9))))
```