Applying Inductive Functional Programming on Bidirectional Transformations

A Case Study With Igor2

Bachelor Thesis

Advisor: Prof. Dr. Ute Schmid
Prof. Dr. Janis Voigtländer

28.09.2018

Sebastian Seufert
1482673
## Contents

1 Introduction

2 Bidirectional Transformations
   2.1 BX And Functional Programming
   2.1.1 A Concrete BX Task
   2.2 Lenses

3 Inductive Programming
   3.1 Inductive Functional Programming
   3.1.1 Generate-and-Test vs. Analytical Approaches
   3.2 IGOR2
   3.2.1 Basic Algorithm
   3.2.2 Demonstration of Interpreter

4 Synthesizing BX with IP
   4.1 Conceptual Framework
   4.2 Combining IGOR2 and BX
   4.2.1 Rewriting Strategy
   4.2.2 Indexing Strategy
   4.2.3 Illustrating the Preprocess
   4.3 Exploring Function Classes
   4.3.1 Non-indexed gets
   4.3.2 Indexed gets
   4.4 Practical Problems
   4.4.1 Issues with Software Decay
   4.4.2 Homoiconicity vs. Typed Languages

5 Conclusion
   5.1 Summary
   5.2 Future Work

A Specification
   A.1 Views with lost Information
A.1.1 init ................................................. 24
A.1.2 halve ............................................. 24
A.2 Views as Permutation .................................. 26
  A.2.1 swapCons ...................................... 26
  A.2.2 swap_halves ................................... 26
A.3 Indexed Views ........................................ 27

B Contents of Medium ...................................... 29

C Formalization ............................................ 30
  C.1 Definition of Lens in Agda .......................... 30
  C.2 sieve and putsieve ................................ 31
  C.3 Instantiation of Lens-record ....................... 32
LIST OF FIGURES

2.1 Reduced schematic of the View-Update-Problem . . . . . . . . . . . . 3
2.2 Example of View from a Source . . . . . . . . . . . . . . . . . . . . . . 4

3.1 Example of interpreted session with IGOR2 . . . . . . . . . . . . . 13
List of Tables

2.1 Results of sieve on strings ........................................... 5
2.2 Results of put_sieve on strings ........................................ 5
3.1 Partitioning example ..................................................... 11
4.1 Overview on tested gets ................................................ 19
## List of Listings

1. `put_sieve` on lists .................................................. 6
2. Ambiguous call of `put_sieve` ........................................ 6
3. Specification for `reverse` on lists .................................. 10
4. Initial hypothesis ....................................................... 10
5. Successor after refinement ............................................ 12
6. Successor after 2nd refinement ....................................... 12
7. Specification for `sieve` ................................................ 14
8. Conflicting definitions for `put_tail` ............................... 16
9. Unused variables for `put_tail` ....................................... 17
10. Generated get for `init` ............................................... 18
11. Generated put for `init` ............................................... 18
12. `put_sieve_ind` on lists ............................................. 19
13. Applying `put_sieve_ind` ............................................. 20
Regardless of perspective, computer scientists and programmers would agree that if one takes an abstract view on programs, most of them can be subsumed as systems that operate on data to transform some given input to a desired output. Although this unidirectional flow of transformation appears inherent to any form of computation, there exist many situations in several domains where it is desirable to have a canonical method for both forward and backward transformations, for either reversing a generated output back to its original input, or – depending on the context – transforming it back to some input(s) that may be deemed sensible with a notion of ‘best choice here’.

**Bidirectional Transformations (BX)** Systems or frameworks that intend to support aforementioned functionality are in general required to design and implement both forward and backward transformations, leading to a higher effort in cost and maintainability. The research field of *bidirectional transformations (BX)* is concerned with techniques and methods to derive from a given transformation its appropriate backward transformation in a canonical way. A bidirectionalization can be described as program, component or function that operates forwards as well as backwards – without the necessity to explicitly implement both aspects of transformation.
**Inductive Programming (IP)**  
*Inductive programming*, or programming by examples, is a subdiscipline of program synthesis and machine learning, with research dating back to the 1960s. A system that implements program synthesis aims to derive or induce a generalization for a given specification, for instance, input-output examples of a functional program and a general recursive function.

Janis Voigtländer has proposed to connect both these topics by raising the question of how inductive programming can alleviate some of the problems arising in bidirectional transformation tasks, and further, where eventual boundaries of this support may lie [1].

To this end, Chapter 2 gives a terse introduction to the field of *BX*, focusing on the field of lenses and functional programming. This is followed by a short introduction to the theory and applications of inductive programming, presenting the Haskell-port of the analytical IP-system IGOR2 in Chapter 3.

After these necessary preliminaries, Chapter 4 lays out a conceptual framework for examining and testing the combination of BX with inductive programming. Its aim is to map out classification of contextual settings and functions as well as possible methods to specify bidirectionalization tasks for IGOR2.

The programming languages Haskell and Agda are employed throughout this thesis, the latter for formalizing a minor statement where the former lacks expressiveness. Basic knowledge regarding the concepts of functional programming is assumed; however, advanced techniques or less obvious function names are explained.
A substantial amount of tasks in software engineering revolve around keeping a
notion of consistency between (at least) two separate representations of the same
underlying data [2],[1]. Throughout existing literature, an often cited instance of
such task is the View-Update-Problem (figure 1), arising in the field of relational
databases [3] where the two representations consist of a source database and a pro-
jected view on it. Changing the view gives rise to the question of how one can
propagate this updated view back to the source database in a meaningful way, keep-
ing certain consistency constraints.

\[
\begin{array}{c}
S \xrightarrow{\sigma} V \\
S' \xleftarrow{\text{?}} V'
\end{array}
\]

**Figure 2.1:** Reduced schematic of the View-Update-Problem

Take as source *staff* and *projects*, (figure 2.2) combined to a conditional view
(compare [2, p. 4]): What would be an appropriate translation for the datum M.S.
being deleted from this view with respect to the original source – should it be deleted
from all projects, or should it be moved to project B? Albeit this basic examples is
Bidirectional Transformations

<table>
<thead>
<tr>
<th>Name</th>
<th>Room 05.043</th>
<th>Phone 863-1234</th>
<th>Project</th>
<th>Person U.S</th>
<th>Role Lead</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.S.</td>
<td>863-1234</td>
<td></td>
<td>A</td>
<td>U.S</td>
<td>Lead</td>
</tr>
<tr>
<td>J.R.</td>
<td>863-1235</td>
<td></td>
<td>A</td>
<td>M.S</td>
<td>Test</td>
</tr>
<tr>
<td>M.S.</td>
<td>863-1234</td>
<td></td>
<td>B</td>
<td>M.S</td>
<td>Lead</td>
</tr>
</tbody>
</table>

Staff Projects View

**Figure 2.2:** Example of View from a Source

admittedly underspecified, it serves to show that there exist several sensible solutions that may be acceptable as behavior.

Bidirectional transformations are concerned with deriving such a suitable backward transformation when the only information given consists of the forward transformation. As mentioned, from a higher perspective one can see many instances of this problem arising across several disciplines of computer science, thus making it a topic of increased research interest [4]. Examples include, for instance, the development of graphical interfaces, serialization issues or model driven development, all related regarding the necessity to keep consistency between different representations of the same data. A comprehensive overview can be found in [4].

It is a given consequence that terminology and approaches to describing and solving BX differ due to their respective fields of application. Since this thesis aims to investigate BX in the context of functional programming, definitions and terminology provided by [1] and [5] are adhered to. An exception are the terms source and view that can be found ubiquitous throughout the literature, even if they are primarily associated with the field of relational databases.

## 2.1 BX And Functional Programming

Due to the ubiquity of its applications and existing theoretical approaches, the remainder of this thesis is constrained to the perspective of BX in the context of functional programming languages. To that accord, a motivating example\(^1\) of bidirectional transformation in this setting may give an informal intuition about the task before introducing the more formally concept of lenses.

\(^1\)Based on a talk given by J. Voigtlander at the Dagstuhl seminary on AAIP, [https://www.dagstuhl.de/15442](https://www.dagstuhl.de/15442)
2.1.1 A Concrete BX Task

Consider a polymorphic function

\[ \text{sieve} :: [\alpha] \rightarrow [\alpha] \]

on homogeneous lists of arbitrary type \( \alpha \) that returns a \textit{view} on its input source by keeping every second element (see Table 2.1).

| source | "" | "a" | "ab" | "abc" | "abcd" | "abcde" | ...
|--------|-----|-----|------|-------|--------|--------|...
| let view = sieve s | "" | "" | "b" | "b" | "bd" | "bd" | ...

Table 2.1: Results of \textit{sieve} on strings

The task at hand here consists of finding a suitable function \textit{put_sieve}, such that if the view is changed or updated, \textit{put_sieve} reflects the change back to the original source. Regarding the example, let the update consist of upper casing the given elements, then the desired results of \textit{put_sieve} are as follows:

| source | "" | "a" | "ab" | "abc" | "abcd" | "abcde" | ...
|--------|-----|-----|------|-------|--------|--------|...
| let view = sieve s | "" | "" | "b" | "b" | "bd" | "bd" | ...
| let view' = to Upper v | "" | "" | "B" | "B" | "BD" | "BD" | ...
| putSieve s v' | "" | "A" | "aB" | "aBc" | "aBcD" | "aBcDe" | ...

Table 2.2: Results of \textit{put_sieve} on strings

Since \textit{sieve} works in a destructive way, that is, loses information compared to the view, and there is no further machinery in the background to keep track of changes, a function performing this reflection needs to have the original list for input as well, thus accepting two parameters:

\[ \text{put_sieve} :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \]

Listing 1 shows one possible function to that effort:

**Intricacies and Ambiguities**

While \textit{put_sieve} works for the examples from above, it is admittedly contrived. Thinking about further possible inputs, one is bound to encounter very soon be-
2.1 Bidirectional Transformations

<table>
<thead>
<tr>
<th>put_sieve :: [a] -&gt; [a] -&gt; [a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>put_sieve x [] = x</td>
</tr>
<tr>
<td>put_sieve [x,y] (v:<em>:</em>) = [x,v]</td>
</tr>
<tr>
<td>put_sieve (s:t:ts) (v:vs) = s : v : put_sieve ts vs</td>
</tr>
</tbody>
</table>

Listing 1: put_sieve on lists

Behavior that may not be intended with ‘inverting’ the effects of this view-producing function, especially when aiming for a generalized approach that entails more complex operations on data. Although the semantic of sieve seems clear, the behavior of its counterpart may very quickly turn out to be less clear: barring non-totality here for a moment⁴, the call to put_sieve as shown in listing 2 is not clearly defined concerning its result. Is it acceptable to omit the letter E to produce "aBcD", or should it rather take semantic aspects of how the view was produced into stride and thus append the update to produce something of "aBcDE"?

A more precise definition of the notions presented in this chapter is needed to capture the essential idea of BX and to state desired and undesired effects.

Listing 2: Ambiguous call of put_sieve

put_sieve "abcd" "BDE"

²Not to be confused with the notion of a program inversion.
³put_sieve as shown would fail on these arguments since it is not totally defined.
2.2 Lenses

_Lenses_ are an abstraction that describe bidirectional transformations as well as their (intended) behavior and properties. In general, lenses are pairs of functions that operate on two structures $S, V$, denoting the _source_ and _view_:

**Definition** Basic Lens  Let $S, V$ be sets. A tuple of functions _get_ and _put_

\[
\begin{align*}
\text{get} &: S \rightarrow V \\
\text{put} &: S \times V \rightarrow S
\end{align*}
\]

form a lens if the following two properties hold:

\[
\begin{align*}
\text{PutGet}: & \quad \text{get}(\text{put}(s,v)) \equiv v \\
\text{GetPut}: & \quad \text{put}(\text{get}(s)) \equiv s
\end{align*}
\]

_PUTGET_ and _GETPUT_ are called _round-tripping_ rules since they state the intended behavior of consecutive application. Even more strict conditions can be imposed by the following requirements: [1]

\[
\begin{align*}
\text{PutPut}: & \quad \text{put}(\text{put}(s,v')) v'' \equiv \text{put}(s,v'') \\
\text{Undoability}: & \quad \text{put}(\text{put}(s,v')) (\text{get}(s)) \equiv s
\end{align*}
\]

With this definition at hand the task of bidirectionalization can now be rephrased to: Given some function _get_ resulting in $v \in V$, how can one derive a meaningful _put_ that, when applied to the source and an updated $v'$ then produces an accordingly updated source $S'$ which behaves to the laws given above. [1]

With regard to the initial example in 2.1, _get = sieve_ while _put = put_sieve_. Including only PutGet and GetPut, a small formalization within Agda in Appendix C shows that a relaxed formulation of this tuple is a lens.
Inductive Programming

The field of inductive programming lies on the boundaries of machine learning, software engineering and declarative programming. It is concerned with the task of deriving algorithms from a set of incomplete specifications (cf. [6], [7]). The following introduces basic concepts of inductive programming and provides some prerequisite foundations on the methods that are used. Special focus is placed on inductive functional programming and one of its representatives, the system IGOR2.

3.1 Inductive Functional Programming

At large, induction describes the process of inferring a general law or rule from a set of particular instances (cf. [8]); analogous, inductive programming labels systems that are capable of inferring programs or functions from a set of inputs.

Substantiating the description above for inductive programming, a system incorporates the paradigm of inductive functional programming as follows; the desired output is a (general) recursive function called the target function, induced from its specification, consisting here of explicit examples for the input and output values, or I/O examples. Listing 3 shows a specification with reverse as target function and each line denoting one I/O example, illustrating the aspect of being specified incompletely. A successful synthesis would return a function definition that contains a recursive call, that is, a generalized function total with respect to its input domain.
3.1.1 Generate-and-Test vs. Analytical Approaches

One can differ between two approaches used for synthesis which are diametrical in concept. Systems using a generate-and-test approach generate repeatedly several hypothetical functions that are then tested, taking the specification as a fitness criterion [7, pp. 64]. The capabilities of generate-and-test systems are eventually only restricted to their respective language bias, that is, the set of all possible programs that are syntactically correct with respect to the underlying language that is used for construction. Consequently, this may cause a lack in performance due to the necessity of searching through the generated functions.

On the other hand, the analytical approach induces its hypotheses depending on the given specification. By analyzing the I/O examples on reoccurring regularities, a general recursive function can directly be constructed. Under certain circumstances this outperforms generate-and-test approaches, but also reduces a certain amount of flexibility compared to the former.

However, it should be noted that this distinction is rather to be understood as a spectrum when it comes to realized prototypes of inductive programming systems, since implementations may share methods or features from both approaches.

3.2 IGOR2

IGOR2 is a prototype system of inductive functional programming that uses the analytical approach described above. It has been developed at the university of Bamberg, the latest iteration in the course of two doctoral theses which explored further methods and concepts to enhance the systems capabilities (see [7], [9]). Harnessing the advantages of homoiconicity, it was first implemented in the term-rewriting language Maude and later ported to Haskell. In its latest version, IGOR2 is augmented by extended features, for instance, methods to include enumerative methods [10] or, taking advantage of Haskell’s type system, to guide synthesis [9].

\footnote{That is, languages that support \textit{data as code, code as data}.}
\footnote{http://maude.cs.illinois.edu/w/index.php/Maude_download_and_installation}
more, IGOR2 is capable of using functions within a given specification as background knowledge [7] and inventing auxiliary functions.

3.2.1 Basic Algorithm

IGOR2 features several advantageous properties for inductive programming, foremost allowing a specification to include user-defined algebraic data types, and guaranteed termination amongst others [7]. The general framework describes the application of term rewriting for functional programs[11], extended by rewrite rules allowing for recursion on inductively defined algebraic datatypes.

The following gives a shortened sample synthesis up to the first iteration for the target function \texttt{reverse} as defined in listing 3 (cf. [7, pp. 84]).


definitions for reverse:
1. reverse [] = []
2. reverse [a] = [a]
3. reverse [a,b] = [b,a]
4. reverse [a,b,c] = [c,b,a]

Listing 3: Specification for \texttt{reverse} on lists

Find initial hypothesis The first step consists of pattern matching on the given specification and searching for a term that represents the minimal generalization covering all examples (1 - 4), leading to the initial hypothesis (listing 4), combined with the unbound variables \textit{a : as} on the right hand side.

\begin{verbatim}
reverse (x:xs) = (a:as)
\end{verbatim}

Listing 4: Initial hypothesis

Partitioning of examples With respect to the initial hypothesis, the original specification can be partitioned into sets by means of case distinction on the constructor(s) used. In the case of lists this partitions the given examples into two sets,
depending on the constructor \textit{Nil} ([]) or \textit{Cons} (:)\textsuperscript{3}:

\[
\begin{array}{l}
\{1\} \quad \{2,3,4\} \\
\hline \\
\text{reverse } [] = [] & \text{reverse } [a] = [a] \\
& \text{reverse } [a,b] = [b,a] \\
& \text{reverse } [a,b,c] = [c,b,a]
\end{array}
\]

\textbf{Table 3.1:} Partitioning example

\textsuperscript{3}Idiosyncrasy of Haskell: Note that \([a]\) and \((a : [])\) are equivalent in a functions body, while \([a]\) in a signature refers to the type of lists of \(a\).
The successive candidate for hypothesis is now further refined to cover both partitions, resulting in the equations of 5. Only the second line allows further refinement,

<table>
<thead>
<tr>
<th></th>
<th>reverse [] = []</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>reverse (x:xs) = (a:as)</td>
</tr>
</tbody>
</table>

**Listing 5:** Successor after refinement

here by again partitioning according to as before. This divides the set {2,3,4} further to {{2},{3,4}} and leads to equation 2 and 3 of listing 6, subsuming further the original I/O examples. Further operators to close the unbound variables consist of abducting new auxiliary functions, calling provided background knowledge or making a recursive call to the target function.

<table>
<thead>
<tr>
<th></th>
<th>reverse [] = []</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>reverse (x : []) = (x : [])</td>
</tr>
<tr>
<td>3</td>
<td>reverse (x:y:z) = (a:b:c)</td>
</tr>
</tbody>
</table>

**Listing 6:** Successor after 2nd refinement
3.2.2 Demonstration of Interpreter

Revisiting `sieve` again with the set of equations to form I/O examples depicted in listing 7, running a synthesis in an interpreted session of IGOR2 is shown in figure 3.1.

```
Welcome to IgorII.
Type :h to get help.
IgorII > :l examples/Pre/Generated/Sieve.hs
File loaded in 0.9160s
IgorII > :generalise sieve
- - - - START SYNTHESIS WITH - - - -
Targets 'sieve'
Background <none>
Simplified False
Greedy rule-splitting False
Accumulators False
Enhanced False
Use paramorphisms False
Compare rec args A Wise
DumpLog False
Debug False
Maximal tiers 0
Maximal loops -1
- - - - - - FINISHED - - - - - - - -
sieve in 67 loops
CPU: 0.4600s

HYPOTHESIS 1 of 1
sieve _[] = []
sieve (a0 : (a1 : a2)) = a1 : fun1 (a0 : (a1 : a2))
fun1 (a0 : (a1 : a2)) = sieve (fun23 (a0 : (a1 : a2)))
fun23 (_ , _ ) = [error "?a"]
fun23 (_ : (_ : (_ : a3)))) = error "?a" : a3

IgorII > :test sieve on "sieve_{[1,2,3,4,5,6,7,8,9,10]}"
Testing 1. hypothesis of: 'sieve'
sieve [1,2,3,4,5,6,7,8,9,10] == [2,4,6,8,10]
```

Figure 3.1: Example of interpreted session with IGOR2
The term **error** "?a" does *not* indicate an error concerning synthesis, but is the result of extending IGOR2 with wildcards within the target function\(^4\); the function could be rewritten as

\begin{verbatim}
sieve [_] = []
sieve (a0 : (a1 : a2)) = a1 : sub1_sieve (a0 : (a1 : a2))
sub1_sieve (a0 : (a1 : a2)) = sieve (sub2_sieve (a0 : (a1 : a2)))
sub2_sieve [_, _] = [_, _]
sub2_sieve (_ : (_ : (_ : a3))) = _ : a3
\end{verbatim}

\textbf{Listing 7:} Specification for \texttt{sieve}

\footnote{From correspondence with J. Voigtländer.}
Synthesizing BX with Inductive Programming

With an overview over IGOR2’s capabilities and an idea of the difficulties arising within bidirectional transformations, the following presents a conceptual framework on how both topics may be evaluated regarding their ‘interplay’. This framework is then applied to selected instances, inspecting some preliminary results and discriminating problems. An outline of automating the framework is sketched afterwards.

4.1 Conceptual Framework

The general idea is to let IGOR2 synthesize put functions in a black box approach and test afterwards if the returned hypotheses behave well according to PUTGET and GETPUT, as defined in chapter 2.2. To facilitate a systematic evaluation of this question, it is necessary to determine a concept that describes what kind of functions are tested and how the flow of data should be arranged. The following breaks the intended process down to each single step.

Select function Firstly, a function is selected as designated get such that a lens as described in 2.2 can be defined. A first approximation of selection criteria is based on how the function is typed, meaning its signature denoting domain and codomain.
Concerning functions that work on structures, a further distinction can be made if the number of elements remains the same or changes, that is, if the produced view has lost information compared to its source or not.

**Translation to I/O examples** The function to be tested is executed on a set of inputs and rewritten to I/O examples for both `get` and `put` such that their representation is acceptable for IGOR2. This is done semi-automatically with a preprocessor realized in Haskell (see preprocessing toolchain in Appendix B).

**Synthesize put and compare** Using a black box approach, that is, neglecting its inner workings for the task at hand, IGOR2 is then used to synthesize an according `put` function on the given specification. The function resulting from this synthesis can then be checked with respect to the intended behavior with a wider range of values than the original specification entailed.

### 4.2 Combining IGOR2 and BX

Although the team around Voigtländer extended IGOR2 to support synthesis including wildcards within a target function, IGOR2 does neither accept wildcards nor the same identifier on the left hand side of the examples. This makes rewriting apt `put` functions more difficult if the desired structure should be typed as denoted in section 2.2.

#### 4.2.1 Rewriting Strategy

Considering a lens with `tail` as `get`, a conceivable specification of `put` is shown in listing 8, causing conflicting definitions on the left hand side of the equations.

```haskell
1  put_tail :: [a] -> a -> [a]
2  put_tail [a,b] a = [a,b]
3  (....)
```

**Listing 8**: Conflicting definitions for `put_tail`
Renaming alone as depicted in listing 9 to circumvent this issue would not suffice, since there is no deeper inspection regarding equality on the elements $x$ and $a$.

\begin{verbatim}
put_tail :: [a] -> a -> [a]
put_tail [a,b] x = [a,b]
(....)
\end{verbatim}

Listing 9: Unused variables for $\text{put\_tail}$

IGOR2 would discard $x$ and synthesize $\text{put\_sieve} (x:y) z = (x:y)$.

In the following this is circumvented by modifying the structure of $\text{put}$ to form a tuple, that is,

$$\text{put} : S \times V \rightarrow S$$

gets translated into specifications of the form

$$\text{put}_{Ig} : S \rightarrow (V \times S)$$

Besides avoiding trivial hypothesis or rejected specifications, this rewrite hopefully enforces synthesis over the resulting tuple, with the consequence of entailing the actual $\text{get}$ function in a reusable manner. However, this also requires the laws to be modified to

$$\text{put} s \equiv (\text{get}(s), s) \quad (\text{PutGet}_{Ig})$$

$$\text{get} (\text{put}_v s) \equiv v \quad (\text{GetPut}_{Ig})$$

with $\text{put}_v$ denoting the projection on $(V \times S)$.

### 4.2.2 Indexing Strategy

If, however, some $\text{get}$ can be combined with a notion of inspecting equality on its type, the machinery described before turns out to be unnecessary. Forfeiting polymorphism as a first experiment and using constructors instead of free variables as above, IGOR2 might differentiate the elements of the specification.
4.2.3 Illustrating the Preprocess

The following gives a complete example of the preprocessing for \texttt{init} as \texttt{get}. \texttt{init} is defined within Haskell’s prelude, so it is applied to a set of lists in ascending length. Maintaining consistency with earlier descriptions, lists of characters are used here. The domain of \texttt{init} contains one input that has no defined output, namely the empty list; this case is exempt from the generated specification(s). The

\begin{verbatim}
init :: [a] -> [a]
init [a] = []
init [a,b] = [a]
init [a,b,c] = [a,b]
init [a,b,c,d] = [a,b,c]
\end{verbatim}

\textbf{Listing 10: Generated get for init}

The final specification results then from rewriting the examples of the respective \texttt{get} to the designated pattern for \texttt{put} as shown in listing 11.

\begin{verbatim}
put_init :: [a] -> ([a],[a])
put_init [a] = ([],[a])
put_init [a,b] = ([a],[a,b])
put_init [a,b,c] = ([a,b],[a,b,c])
put_init [a,b,c,d] = ([a,b,c],[a,b,c,d])
\end{verbatim}

\textbf{Listing 11: Generated put for init}

4.3 Exploring Function Classes

The following functions have been processed according to the scheme described above, after discerning them on the factor if they lose information or change information. The detailed specifications as well as the target function synthesized by IGOR2 are listed in their entirety under appendix A.
4.3 Synthesizing BX with IP

4.3.1 Non-indexed *gets*

A *get* function loses information if it discards parts of the source it is applied to, while it performs some kind of permutation if it retains its overall information. Exemplary candidates are shown in table 4.1.

<table>
<thead>
<tr>
<th>Class</th>
<th>get</th>
<th>Type</th>
<th>put</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Losing</td>
<td>halve</td>
<td>[a] -&gt; [a]</td>
<td>put_halve</td>
<td>[a] -&gt; ([a],[a])</td>
</tr>
<tr>
<td></td>
<td>init</td>
<td>[a] -&gt; [a]</td>
<td>put_init</td>
<td>[a] -&gt; ([a],[a])</td>
</tr>
<tr>
<td>Permuting</td>
<td>swap_cons</td>
<td>[a] -&gt; [a]</td>
<td>put_swap_c</td>
<td>[a] -&gt; ([a],[a])</td>
</tr>
<tr>
<td></td>
<td>swap_-halves</td>
<td>[a] -&gt; [a]</td>
<td>put_swap_h</td>
<td>[a] -&gt; ([a],[a])</td>
</tr>
</tbody>
</table>

Table 4.1: Overview on tested *gets*

Unfortunately, all synthesized *puts* from the functions listed above are ultimately just rewrites of their original *get*. With the exception of *swap_halves*, they are a successful generalization of their specification, but no new perspective or change presented itself as connecting factor to develop further approaches.

4.3.2 Indexed *gets*

Since the approach of rewriting *puts* has proved unsuccessful, a further examination attempts indexing positions to enable IGOR2 to discriminate between elements. The lens to be tested is derived similar to the steps before; however, in this instance polymorphism is omitted and replaced by a unary representation of the natural numbers \( \mathbb{N} \) as denoted in the Peano axioms. The selected *get* is *sieve_ind*, reduced to contain only positional information.

Listing 12 shows the synthesized *put* function, with \( S \ n \) denoting the successor-constructor and \( Z \) zero.

1

\[
\text{put_sieve_ind} \ (S \ _ : \ _) \ (_ : \ a2) = S \ Z : a2
\]

Listing 12: *put_sieve_ind* on lists
Checking the lens laws is omitted since a simple call with arguments \([1, 2] [3]\) makes it obvious that this function does not adhere to them (see listing 13).

```
Prelude> put_sieve_ind [S Z, S (S Z)] [S (S (S Z))] => [S Z]
```

**Listing 13:** Applying `put_sieve_ind`

### 4.4 Practical Problems

#### 4.4.1 Issues with Software Decay

The version used was conveyed via e-mail from Janis Voigtländer and is at the time of writing not publicly available for download; its source code can be found on the accompanying medium (see appendix B). Trying to build this version failed due to some minor syntactical errors that were easily mended; this is presumably due to changes in Haskell’s build environment and not a mistake in the original implementation.

More grave proved to be dependency-issues of libraries that are either no longer supported or, when substituted with newer alternatives, resulted in non-resolvable interdependencies\(^1\). An attempt of restoring a build-environment to accommodate this problem was made, unfortunately to no avail and was finally given up. In the end, barring many warnings and fixing the syntax errors, a build was possible under Linux Debian 8 in combination with the Stack Toolchain\(^2\), using the standard compiler `ghc-8.0.2`\(^3\).

#### 4.4.2 Homoiconicity vs. Typed Languages

The preprocessing toolchain described in sections 4.2.1 and especially 4.2.3 can also be found in appendix C. The original intention was to conduct all testing completely

\(^1\)The Haskell community coined the name *cabal hell* due to Haskell’s package management tool *cabal*.


\(^3\)[https://www.haskell.org/ghc/download_ghc_8_0_2.html](https://www.haskell.org/ghc/download_ghc_8_0_2.html)
automatically, including a call to IGOR2 and checking the result of synthesis on properties formulated within the framework QuickCheck\(^4\).

Using the interpreter library hint\(^5\) as a light-weight substitution for emulating an interpreted session with Haskell’s own official interpreter ghci resulted in obstacles which were not anticipated; while probably not unsolvable, the main hindrance was imposed by the type system itself, for instance regarding type defaulting on empty lists.

The implementation circumvents this problem by merely pretty printing the specifications for IGOR2, but lacks consequently the necessary feature of a real flexible pre- and postprocessing.

\(^4\)http://hackage.haskell.org/package/QuickCheck
\(^5\)http://hackage.haskell.org/package/hint
5.1 Summary

The goal was to present and inspect bidirectional transformations as well as inductive functional programming, aiming to explore the general compatibility when combining both. Although the results are of a preliminary nature and only sketch possible approaches, they may be carried on by further refining the theoretical aspects of BX with respect to IP, especially in the context of functional programs and IGOR2.

Concretely, an account of the concept of lenses to describe BX was given and then transferred to the context of inductive functional programming, leading to an examination of the problems encountered when synthesizing the put-part of a lens. The preprocessing tool used works adequately for this task.

5.2 Future Work

Eventual discoveries can be made with taking the black box approach further. While the two proposed strategies of reformulating specifications did not pan out as intended, there still may be potential in the notion of indexing or marking structures in a manner that is agreeable with IGOR2.

A whole other perspective is of course opened as soon as the black box approach
is discarded; possible work could then entail reconciling IGOR2’s constructor term systems and BX, followed by further extensions to the implementation.
A.1 Views with lost Information

A.1.1 \textit{init}

get := init, defined in Haskell’s prelude

Generated put:

\begin{verbatim}
1 put_init :: [a] -> ([a],[a])
2 put_init [a] = ([],[a])
3 put_init [a,b] = ([a],[a,b])
4 put_init [a,b,c] = ([a,b],[a,b,c])
5 put_init [a,b,c,d] = ([a,b,c],[a,b,c,d])
6 put_init [a,b,c,d,e] = ([a,b,c,d],[a,b,c,d,e])
7 put_init [a,b,c,d,e,f] = ([a,b,c,d,e],[a,b,c,d,e,f])
8 put_init [a,b,c,d,e,f,g] = ([a,b,c,d,e,f],[a,b,c,d,e,f,g])
9 put_init [a,b,c,d,e,f,g,h] = ([a,b,c,d,e,f,g],[a,b,c,d,e,f,g,h])
\end{verbatim}

Synthesized function:

\begin{verbatim}
1 put_init (a0 : a1) = (fun1 (a0 : a1), a0 : a1)
2 fun1 [_] = []
3 fun1 (a0 : (a1 : a2)) = a0 : fun1 (a1 : a2)
\end{verbatim}

A.1.2 \textit{halve}

get := halve, defined as
A.1 Specification

1. \( \text{halve} :: [a] \rightarrow [a] \)
2. \( \text{halve \ } xs = \text{take (length \ } xs \div 2) \ \text{xs} \)

Generated put:

1. \( \text{put\_halve} :: [a] \rightarrow ([a],[a]) \)
2. \( \text{put\_halve} \ [a] = ([],\ [a]) \)
3. \( \text{put\_halve} \ [a,b] = ([a],[a,b]) \)
4. \( \text{put\_halve} \ [a,b,c] = ([a],[a,b,c]) \)
5. \( \text{put\_halve} \ [a,b,c,d] = ([a,b],[a,b,c,d]) \)
6. \( \text{put\_halve} \ [a,b,c,d,e] = ([a,b],[a,b,c,d,e]) \)
7. \( \text{put\_halve} \ [a,b,c,d,e,f] = ([a,b,c],[a,b,c,d,e,f]) \)
8. \( \text{put\_halve} \ [a,b,c,d,e,f,g] = ([a,b,c],[a,b,c,d,e,f,g]) \)
9. \( \text{put\_halve} \ [a,b,c,d,e,f,g,h] = ([a,b,c,d],[a,b,c,d,e,f,g,h]) \)

Synthesized function:

1. \( \text{put\_halve} \ (a0 : a1) = (\text{fun1} \ (a0 : a1), a0 : a1) \)
2. \( \text{fun1} \ [\_] = [] \)
3. \( \text{fun1} \ (a0 : (a1 : a2)) = a0 : \text{fun1} \ (\text{fun26} \ (a0 : (a1 : a2))) \)
4. \( \text{fun154} \ [\_ , \_ , \_ ] = [] \)
5. \( \text{fun154} \ (\_ : (\_ : (a2 : (a3 : a4)))) = \)
   \( \quad \quad a2 : \text{fun154} \ (\text{error } "?a" : (\text{error } "?b" : (a3 : a4))) \)
6. \( \text{fun26} \ [\_ , \_ ] = [\text{error } "?a"] \)
7. \( \text{fun26} \ (a0 : (a1 : (a2 : a3))) = a1 : \text{fun154} \ (a0 : (a1 : (a2 : a3))) \)
A.2 Views as Permutation

A.2.1 \textit{swapCons}

get := \textit{swapCons}, defined as

\begin{verbatim}
1 swap_cons :: [a] \rightarrow [a]
2 swap_cons [] = []
3 swap_cons [a] = [a]
4 swap_cons [a,b] = [b,a]
5 swap_cons (x:y:ys) = (y : x : swap_cons ys)
\end{verbatim}

Generated put:

\begin{verbatim}
1 put_swap_cons :: [a] \rightarrow ([a],[a])
2 put_swap_cons [a] = ([a],[a])
3 put_swap_cons [a,b] = ([b,a],[a,b])
4 put_swap_cons [a,b,c] = ([b,a,c],[a,b,c])
5 put_swap_cons [a,b,c,d] = ([b,a,d,c],[a,b,c,d])
6 put_swap_cons [a,b,c,d,e] = ([b,a,d,c,e],[a,b,c,d,e])
7 put_swap_cons [a,b,c,d,e,f] = ([b,a,d,c,f,e],[a,b,c,d,e,f])
8 put_swap_cons [a,b,c,d,e,f,g] = ([b,a,d,c,f,e,g],[a,b,c,d,e,f,g])
9 put_swap_cons [a,b,c,d,e,f,g,h] = ([b,a,d,c,f,e,h,g],[a,b,c,d,e,f,g,h])
\end{verbatim}

Synthesized function:

\begin{verbatim}
1 put_swap_cons (a0 : a1) = (fun1 (a0 : a1), a0 : a1)
2 fun1 [a0] = [a0]
3 fun1 (a0 : (a1 : a2)) = a1 : (a0 : fun8 (a0 : (a1 : a2)))
4 fun8 (_, _) = []
5 fun8 (_ : (_ : (a2 : a3))) = fun1 (a2 : a3)
\end{verbatim}

A.2.2 \textit{swap_halves}

get := \textit{swap_halves}, defined as

\begin{verbatim}
1 swap_halves :: [a] \rightarrow [a]
2 swap_halves xs = let
3     mid = (length xs `div` 2)
4     in (drop mid xs) ++ (take mid xs)
\end{verbatim}
A.3 Indexed Views

get := sieve_ind, defined with sieve as

```haskell
data N = Z | S N deriving Show

sieve :: [a] -> [a]
sieve [] = []
sieve [a] = []
sieve (x:y:ys) = y : sv ys

sieve (x:y:ys) = y : sv ys

sieve_ind :: [Int] -> [N]
sieve_ind xs = sieve (fmap toPeano xs)
```
where the auxiliary function \texttt{toPeano} maps integers to \( N \).

Generated \texttt{put}:

\begin{verbatim}
1 put_sieve :: \([N] \rightarrow [N] \rightarrow [N]
2 put_sieve [(S Z)] [Z] = [(S Z)]
3 put_sieve [(S (S Z))] [(S Z),(S (S Z))] = [(S Z),(S (S Z))]
4 put_sieve [(S (S Z))] [(S Z),(S (S Z)),(S (S (S Z)))]
5   = [(S Z),
6       (S (S Z)),
7       (S (S (S Z)))
8   ]
9 put_sieve [(S (S Z)),(S (S (S (S Z)))))
10   [(S Z),
11      (S (S Z)),
12      (S (S (S Z))),
13      (S (S (S (S Z))))
14   ]
15   = [(S Z),(S (S Z)),
16      (S (S (S Z))),
17      (S (S (S (S Z))))
18   ]
\end{verbatim}

Synthesized function:

\begin{verbatim}
1 put_sieve_ind (S : _) (_ : a2) = S Z : a2
\end{verbatim}
Contents of Medium

Bidirectionalization Tasks/
  Permutation/
    SwapCons.hs
    SwapHalves.hs
  Lost_Information/
    Init.hs
    Halve.hs
  Indexed/
    SieveIndexed.hs

Formulation_Lens/
  Lens.agda

Preprocessing/
  readme.txt
  Main.hs
  Eval.hs
  IOPair.hs
  PrettyPr.hs

Igor2/
  Igor2.zip
  Igor2
Agda\(^1\) is functional programming language that supports dependent typing and theorem proving [12]. With a syntax similar to Haskell, it allows to express propositions as types under Curry-Howard (cf. [13]), used here for a short demonstration of lenses and sieve/put_sieve as referred by section 2.2.

C.1 Definition of Lens in Agda

A lens is a record entailing both \texttt{get} and \texttt{put} as well as a formulation of the first two laws.

\[
\text{module } \text{Lens where} \\
\text{open import } \text{Relation.Binary.PropositionalEquality} \\
\text{record } \text{Lens } (S \ V : \text{Set}) : \text{Set} \text{ where} \\
\text{field} \\
\text{get } : S \to V \\
\text{put } : S \to V \to S \\
\text{putGet } : \forall \{v \ s\} \to \text{get } (\text{put } s \ v) \equiv v \\
\text{getPut } : \forall \{s\} \to \text{put } s (\text{get } s) \equiv s \\
\text{open } \text{Lens public}
\]

\(^1\)\text{http://wiki.portal.chalmers.se/agda/pmwiki.php}
**C.2 sieve and putsieve**

`sieve` and `put_sieve` are encoded as indexed lists; the underscore in the name of `put_sieve` is omitted since this would represent an infix-notation within Agda.

---

**The Caveat:** `sieve` is restricted on `(length source / 2) < length view`

\[
\text{sieve} : \forall \{A : \text{Set}\} \{n : \mathbb{N}\} \rightarrow \text{List} \ A \ n \rightarrow \text{List} \ A \ (\lfloor n / 2 \rfloor)
\]

\[
\text{sieve}[\ [] = []
\]

\[
\text{sieve}(s ::[]) = []
\]

\[
\text{sieve}(s :: s_2 :: \text{src}) = s_2 ::\text{sieve} \ \text{src}
\]

---

`putsieve` does nothing too exciting for \(m > n/2\)

\[
\text{putsieve} : \forall \{A : \text{Set}\} \{n m : \mathbb{N}\} \rightarrow \text{List} \ A \ n \rightarrow \text{List} \ A \ m \rightarrow \text{List} \ A \ n
\]

\[
\text{putsieve} \ [\ [] = s
\]

\[
\text{putsieve}(x :: v) = []
\]

\[
\text{putsieve}(s :: []) (v :: \text{view}) = v :: []
\]

\[
\text{putsieve}(s :: s_2 :: \text{src}) (v :: \text{view}) = s :: v :: \text{putsieve} \ \text{src} \ \text{view}
\]

---

Two lemmas for each law, using congruence of the cons-operator and respective functions definition to show equivalence.

---

**Lemma for putGet**

\[
\text{putGet-lem} : \forall \{n : \mathbb{N}\} \{A : \text{Set}\}
\]

\[
(\text{src} : \text{List} \ A \ n) \rightarrow
\]

\[
(\text{view} : \text{List} \ A \ (\lfloor n / 2 \rfloor)) \rightarrow
\]

\[
\text{sieve}(\text{putsieve} \ \text{src} \ \text{view}) \equiv \text{view}
\]

\[
\text{putGet-lem} [\ [] = \text{refl}
\]

\[
\text{putGet-lem} (s ::[]) [] = \text{refl}
\]

\[
\text{putGet-lem} (s :: s_2 :: \text{src}) (v :: \text{view}) = \text{cong} (_ :: _ :: v) (\text{putGet-lem} \ \text{src} \ \text{view})
\]

---

**Lemma for getPut**

\[
\text{getPut-lem} : \forall \{n : \mathbb{N}\} \{A : \text{Set}\}
\]

\[
(\text{src} : \text{List} \ A \ n) \rightarrow
\]

\[
\text{putsieve} \ \text{src} (\text{sieve} \ \text{src}) \equiv \text{src}
\]

\[
\text{getPut-lem} [\ [] = \text{refl}
\]

\[
\text{getPut-lem} (x :: []) = \text{refl}
\]

\[
\text{getPut-lem} (s :: s_2 :: \text{src}) = \text{cong} (\lambda \ z \rightarrow s :: s_2 :: z) (\text{getPut-lem} \ \text{src})
\]

---

31
C.3 Instantiation of Lens-record

```haskell
-- Instantiating the record Lens
sieve-lens-inhabited : ∀{m n : ℕ} {A : Set} → Lens (List A n) (List A ⌊ n / 2⌋)
get sieve-lens-inhabited = sieve
put sieve-lens-inhabited = putsieve
putGet sieve-lens-inhabited {s} {v} = putGet-lem v s
getPut sieve-lens-inhabited {s} = getPut-lem s
```
BIBLIOGRAPHY


Eidesstattliche Erklärung

Ich erkläre hiermit gemäß § 17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Zeil, 28.09.2018

Sebastian Seufert