CHALMERS

Generalizing Semantic Bidirectionalization & Tracking Generated Expressions

Master of Science Thesis in Computer Science

SHAYAN NAJD JAVADIPOUR

Chalmers University of Technology
University of Gothenburg
Department of Computer Science & Engineering
Gothenburg, Sweden Feb. 2013

Generalizing Semantic Bidirectionalization & Tracking Generated Expressions

Shayan Najd Javadipour
© Shayan Najd Javadipour, February 2013.
Examiner: Professor Mary Sheeran

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Abstract

In programming, there are often pairs of functions running in opposite directions: the domain of one is the codomain of the other. Their functionalities are so closely related that it is possible to derive one from the implementation of the other. *Bidirectionalization* techniques address this concern. This thesis studies some of the theoretical and practical aspects of *bidirectionalization.*

As the theoretical part of this thesis, we generalize an existing *bidirectionalization* technique, known as *semantic bidirectionalization.* Our generalized algorithm scales well and lifts some of the restrictions set by the original algorithm.

As the practical part of this thesis, we focus on the problem of tracking expressions in the low-level generated code to their origins in the high-level code.

Acknowledgements

To begin with, I would like to thank Meng Wang for his immense support. He has always been ready to answer and discuss my questions at any time of the day, even when resting in a sauna after a long conference!

I would also like to express my deepest appreciations to Mary Sheeran for her invaluable guidance throughout this thesis.

While working on the thesis, I had the chance to join the Feldspar research project as a research assistant. Therefore, I would like to thank the members of the Feldspar project, particularly Emil Axelsson, for their help with the practical part of the thesis.

As a project assistant, I worked under the supervision of Niklas Broberg and David Sands. I am grateful for this experience, during which I learnt a great deal on implementation and transformation of programming languages. To Niklas, I owe my understanding of type systems.

I would like to offer my special thanks to Jean-Philippe Bernardy; the use of *obsever functions* in my proposed algorithm is a result of long discussions with him.

I would like to thank all the members of Function Programming research group at Chalmers. Especially, I would like to express my deepest gratitude to John Hughes for all his support and inspiring suggestions.

Last but not least, I would like to express my most sincere appreciations to Anna who never ceases to amaze me.

Shayan Najd, Gothenburg 2013/02/01

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In programming, there are often pairs of functions running in opposite directions: the domain of one is the codomain of the other. Zipping and unzipping, compressing and decompressing, serializing and deserializing, parsing and pretty printing are all instances of such pairs [CFH$^+$09]. Their functionalities are so closely related that it is possible to derive one from the implementation of the other. *Bidirectionalization* techniques address this concern. They have applications in a wide range of areas including software engineering, databases and programming languages. This thesis studies some of the theoretical and practical aspects of *bidirectionalization*.

As the theoretical part of this thesis, we generalize an existing *bidirectionalization* technique, known as *semantic bidirectionalization*. Our generalized algorithm scales well and lifts some of the restrictions set by the original algorithm.

As the practical part of this thesis, we focus on the problem of tracking expressions in the low-level generated code to their origins in the high-level code. *Feldspar* is a relatively complex domain-specific language embedded inside Haskell which is translated into *C* code. We apply the *semantic bidirectionalization* technique to enhance *Feldspar* with the ability to track the expressions in the low-level generated *C* code all the way back to their origins in the high-level *Haskell* code.

In this thesis, we first study the existing techniques (chapter 2) and contribute to the theory behind one of the existing methods (chapter 3). Later, in a practical case study, we apply *bidirectional transformation* (BX) techniques to design and implement a mechanism to track the expressions from the generated low-level code to their origin in the high-level code (chapter 4).

1.1 Semantic Bidirectionalization Revisited

In programming languages research, there are three major approaches [FMV12] in design and implementation of bidirectional transformations [CFH$^+$09]: the language-based approach and two bidirectionalization techniques. The former demands the programmer to code in a specific language designed to produce bidirectional programs; the existing programs should be rewritten in the new language. On the other hand, bidirectionalization techniques do not limit the programmer to code in a specific language; the programmer writes the program in the conventional language of choice and bidirectionalization mechanisms automatically derive the program in the opposite direction. There are two main distinct approaches to bidirectionalization:

1. *Syntactic-Bidirectionalization* [MHN$^+$07]: using the actual code describing the function to derive the function in the opposite direction

2. *Semantic-Bidirectionalization* [Voi09]: using the information provided by the type and run-time behavior of the function to derive the function in the opposite direction

The syntactic approach heavily depends on the actual syntax of the language that the function is written in and cannot handle syntactically complex programs. However, the semantic approach is decoupled from the syntax and it can bidirectionalize any function of specific polymorphic type. In this thesis, theory and practice surrounding semantic bidirectionalization are explored.

The original semantic bidirectionalization method [Voi09] can only bidirectionalize polymorphic functions with the following type signatures:

1. fully polymorphic

   $$f :: \forall a.[\,a\,] \rightarrow [\,a\,]$$

2. polymorphic function constrained with equality constraint

   $$f :: \forall a.\mathit{Eq}\ a \Rightarrow [\,a\,] \rightarrow [\,a\,]$$

3. polymorphic function constrained with an ordering constraint

   $$f :: \forall a.\mathit{Ord}\ a \Rightarrow [\,a\,] \rightarrow [\,a\,]$$

It also employs generic programming techniques to generalize these functions to work on algebraic data types in general [Voi09]:

$$\mathbf{class}\ (\mathit{Traversable}\ k, \mathit{Foldable}\ k', \mathit{Zippable}\ k') \Rightarrow \mathit{Generic}\ k\ k'\ \mathbf{where}\ \{\,\}$$

1. fully polymorphic

   $$f :: \forall k\ k'\ a.\mathit{Generic}\ k\ k' \Rightarrow k\ a \rightarrow k'\ a$$

2. polymorphic function constrained with equality constraint

   $$f :: \forall k\ k'\ a.(\mathit{Generic}\ k\ k', \mathit{Eq}\ a) \Rightarrow k\ a \rightarrow k'\ a$$

3. polymorphic function constrained with an ordering constraint

   $$f :: \forall k\ k'\ a.(\mathit{Generic}\ k\ k', \mathit{Ord}\ a) \Rightarrow k\ a \rightarrow k'\ a$$

For simplicity, in this thesis, we mainly explain the underlying theories and contributions using lists; the above mentioned generalization is orthogonal to the algorithm itself and can be applied at any stage.

The original method does not scale properly and provides a separate mechanism per type signature. In the third chapter of this thesis, the theoretical part, we introduce a new mechanism that generalizes the original semantic bidirectionalization technique. Our system scales very well; it does not need to provide a separate mechanism per type signature. Moreover, our system is general enough to bidirectionalize any higher-order polymorphic function having *observer functions* as their function arguments. *Observer functions* are polymorphic functions that have a monomorphic result type, e.g., $\forall a.a \to a \to Bool$. For example, in addition to functions with all the above type signatures, our system can bidirectionalize functions like the following:

$filter :: \forall a.(a \to Bool) \to [a] \to [a]$
$dropWhile :: \forall a.(a \to Bool) \to [a] \to [a]$
$takeWhile :: \forall a.(a \to Bool) \to [a] \to [a]$
$find\ :: \forall a.(a \to Bool) \to [a] \to Maybe\ a$
$partition :: (a \to Bool) \to [a] \to ([a],[a])$

For instance, consider the function *partition* that takes a predicate (a predicate is a function of the type $a \to Bool$) and splits the input list into two parts: the elements satisfying the predicate and the ones that do not. Given the input list as `"shayan"` and the predicate as (<`'j'`), the output of the function would be (`"haa"`,`"syn"`):

```
ghci > partition (<'j') "shayan"
("haa","syn")
```

Now, if one changes the output to (`"eaa"`,`"smn"`), by changing the characters `'h'` to `'e'` and `'y'` to `'m'`, our algorithm (the function bff_{Par}) would be able to map the changes back to the original source by bidirectionalizing the function *partition*:

```
ghci > (bff_Par partition (<'j') "shayan") ("eaa","smn")
Right "seaman"
```

The expression (bff_{Par} *partition* (<`'j'`) `"shayan"`) can be seen as a partial function that takes the two (potentially modified) parts and puts them together in the right order.

1.2 Tracking Generated Expressions

In the practical part of the thesis, we apply the semantic bidirectionalization techniques to track expressions in the low-level generated code to their origins at the high-level code. In particular, we enhance *Feldspar* [ACS+11] with the ability to track the expressions in the low-level generated *C* code to their origins in the high-level *Haskell* code. *Feldspar* is a relatively complex domain-specific language embedded inside *Haskell*; it generates *C* code to facilitate parallel programming for digital signal processing algorithms. A simple program in *Feldspar* looks like the following:

```

module TestFeldspar where

import qualified Prelude
import Feldspar
import Feldspar.Compiler

inc :: Data Int32 → Data Int32
inc x = x + 1

dec :: Data Int32 → Data Int32
dec x = x − 1

incAbs :: Data Int32 → Data Int32
incAbs a = condition (a < 0) (dec a) (inc a)

cCode :: IO ()
cCode = icompile incAbs
```

It defines three functions using *Feldspar*'s front-end (the imported module *Feldspar*): a function to increase the value of the input by one (*inc*), a function to decrease the value of the input by one (*dec*) and a function to increase the absolute value of the input number by one (*incAbs*). Then using the *Feldspar*'s back-end (the imported module *Feldspar.Compiler*), it defines an expression named *cCode* that compiles the *incAbs* function into the following *C* code:

```
/* The header files are ignored */

void test(int32_t v0, int32_t * out)
{
   if((v0 < 0))
   {
       (* out) = (v0 - 1);
   }
   else
   {
       (* out) = (v0 + 1);
   }
}
```

It defines a function (named *test*) in *C* that accepts an integer (of type *int32_t*) as an input and returns it with its absolute value increased by one. The body of the function represents the expression *incAbs* in line 15 in the high-level *Haskell* code. Moreover, the expression $v0 - 1$ in line 07 and $v0 + 1$ in line 11 of the low-level code represent the body of the functions *inc* and *dec* in the high-level code correspondingly. So far, there is no mechanism to indicate this connection. Therefore, if the generated *C* code, for instance, generates an error at run-time, it is hard to find the problematic expression in the high-level *Haskell* code.

As the practical part of this thesis (chapter 4), we make this connection explicit and enhance *Feldspar* with the possibility to track the generated expressions to their origin in the high-level *Haskell* code. By adding the pragma $\{-\#OPTIONS\_GHC - F - pgmF\ qapp\#-\}$ at the top of the *Haskell* source code, the implemented tracking system gets activated and it automatically annotates the generated expressions in the *C* code with the exact source locations of the corresponding top-level expressions in the *Haskell* code. For instance, by adding the mentioned pragma in the top of (line 01) the *TestFeldspar* module in the above (being defined in the file " /TestFeldspar.hs"), we get the following *C* code (*cCode*):

```
/* The header files are ignored */
void test (int32_t v0,int32_t * out)
{
  /* SrcLoc {srcFilename = "~/TestFeldspar.hs",
    srcLine = 15,srcColumn = 1} */
  if ((v0 < 0))
  {
    /* SrcLoc {srcFilename = "~/TestFeldspar.hs",
      srcLine = 12,srcColumn = 1} */
    (*out) = (v0 - 1);
  }
  else
  {
    /* SrcLoc {srcFilename = "~/TestFeldspar.hs",
      srcLine = 9,srcColumn = 1} */
    (*out) = (v0 + 1);
  }
}
```

In this *C* code, the connection between the expressions in the *C* code and their origin in the *Haskell* code is explicitly mentioned in the annotations (declared via *C* comments): the **if** block is annotated with a source location referring to the body expression of the high-level binding *incAbs*, the body expression of the first branch is annotated with a source location referring to the body expression of the high-level binding *dec* and the body expression of the second branch is annotated with a source location referring to the body expression of the high-level binding *inc*.

Chapter 2

Background

2.1 Invertible Programming

Invertible programming techniques [MMHT10, MHT04b, Wan10, MHT04a, HMT04] borrow the well-studied notion of invertibility from mathematics and use it to calculate the inverse functions. This way, the programmer only implements a function (f) in the desired direction and by calculating its inverse function (f^{-1}), the function in the opposite direction is derived.

$$\begin{array}{l} f \quad :: A \rightarrow B \\ f^{-1} :: B \rightarrow A \\ [\mathit{Left} - \mathit{Invertibility}] \\ \forall x :: A.\ (f^{-1} \circ f)\ x = x \\ [\mathit{Right} - \mathit{Invertibility}] \\ \forall x :: B.\ (f \circ f^{-1})\ x = x \end{array}$$

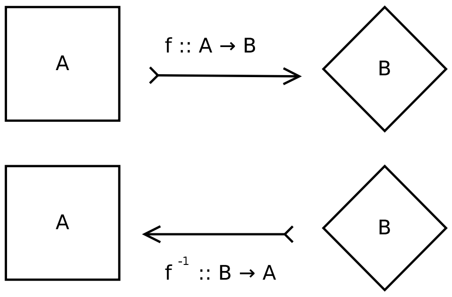


Figure 2.1: Invertible Programming

For example, in the following, the function *unzipp* is the inverse function of the function *zipp*.

```
zipp :: ∀a b.([a],[b]) → [(a,b)]
zipp ([],[]) = []
zipp ((x : xs),(y : ys)) = let
  ps = zipp (xs,ys)
  in (x,y) : ps
```

```
unzipp :: ∀a b.[(a,b)] → ([a],[b])
unzipp []             = ([],[])
unzipp ((x,y) : ps) = let
  (xs,ys) = unzipp ps
  in (x : xs,y : ys)
```

Limitations

A function is invertible if and only if it is possible to define an inverse function for it. Not all functions are invertible; there are fundamental restrictions for calculating the inverse function. In the following, we study these restrictions in more detail.

2.1.1 Invertibility in Mathematics

In mathematics, a function $f :: A \to B$, either partial or total, can be categorized as:

1. Injective (one-to-one)

   $$\mathit{isInjective}\ f : \forall x_1 :: A.\ \forall x_2 :: A.\ f\ x_1 = f\ x_2 \Rightarrow x_1 = x_2$$

2. Surjective (onto)

   $$\mathit{isSurjective}\ f : \forall y :: B.\ \exists x :: A.\ f\ x = y$$

3. Bijective (both one-to-one and onto)

   $$\mathit{isBijective}\ f : \forall x_1 :: A.\ \forall x_2 :: A.\ f\ x_1 = f\ x_2 \Leftrightarrow x_1 = x_2$$

4. Neither (non-injective non-surjective)

Among them, only injective functions are invertible: a function $f :: A \to B$ is invertible if and only if it is at least injective.

$$\mathit{isInjective}\ f \Leftrightarrow (\exists f^{-1} :: B \to A.\ (f^{-1} \circ f)\ x = x)$$

Limitations

In practice, functions are often non-injective and therefore they cannot be used for invertible programming. Fortunately, in some cases, bidirectional programming techniques can be used to approximate the result. In the next section (section 2.2), we explore bidirectional programming techniques.

Even if a function is not injective, it may be possible to define a partial inverse for it. This can be done by restricting the input domain. For example, having $f\ x = x^2$, by restricting the domain to the positive numbers, we have a partial inverse $f^{-1}\ x = \sqrt{x}$. If we allow multivalued inverse functions, it may be possible to define them without restricting the domain. Outputs of such inverse functions are called inverse image (preimage) of their corresponding value in the codomain of the original function. In a more general case, the inverse function theorem gives sufficient conditions for a function to be invertible. The invertibility of a binary relation is also a related concept in mathematics, since a function is a special form of a binary relation.

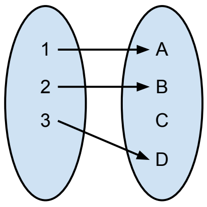
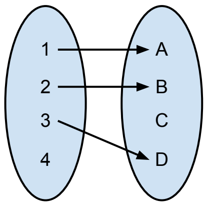
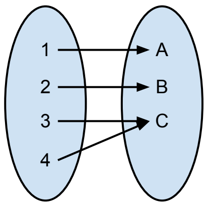
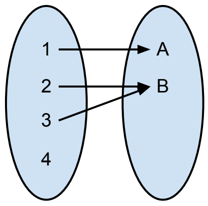
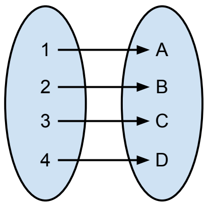
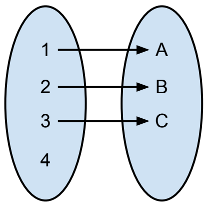
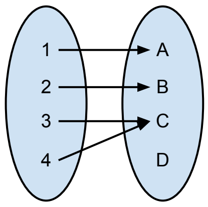
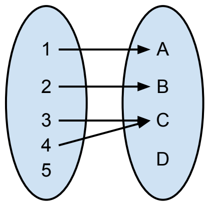
	Total	*Partial*
Injective		
Surjective		
Bijective		
None		

Table 2.1: Function Pairings

2.2 Bidirectional Programming

In bidirectional programming [CFH$^+$09] terminology, the pair of functions consists of a *forward* function (*get*) and a *backward* function (*put*). The input of the forward function is called the *source* and the output is called the *view*.

Consider the function $values :: \forall a.[(String, a)] \rightarrow [a]$ that returns the list of stored values in the input lookup table:

```
values :: ∀a.[(String,a)] → [a]
values [] = []
values ((i,x) : ps) = let
  xs = values ps
  in x : xs
```

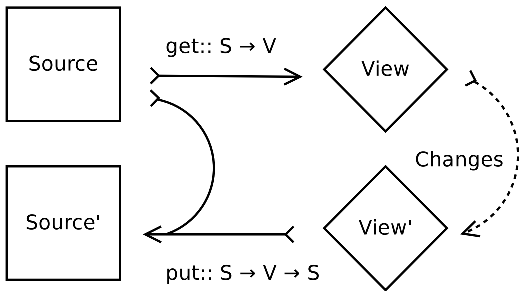


Figure 2.2: Bidirectional Programming

The function *values* can be viewed as the forward (get) function, a value of the type $[(String,a)]$ as the source and a value of the type $[a]$ as the view:

type *Source a* = [(*String*,*a*)]
type *View a* = [*a*]

forward :: ∀*a*.*Source a* → *View a*
forward = *values*

source :: *Source String*
source = [(`"#01"`,`"Keyboard"`),(`"#02"`,`"Mouse"`),(`"#03"`,`"Monitor"`)]

view :: *View String*
view = [`"Keyboard"`,`"Mouse"`,`"Monitor"`]

If the view changes ($view'$), bidirectional transformation can provide the backward function $save :: \forall a.[(String,a)] \to [a] \to [(String,a)]$ to save the changes to the values in the original lookup table:

view′ :: *View String*
view′ = [`"Keyboard"`, `"Speaker"` ,`"Monitor"`]

save :: ∀*a*.[(*String*,*a*)] → [*a*] → [(*String*,*a*)]
save source view′ = *zip* (*map fst source*) *view′*

backward :: ∀*a*.*Source a* → *View a* → *Source a*
backward = *save*

In comparison with the pair of *zipp* and *unzipp*, there are two points to notice:

Firstly, the *values* function is not injective, since at least two distinct values in the domain are mapped to the same value in the codomain.

$$[(\texttt{"0"},0)] \neq [(\texttt{"1"},0)] \wedge \mathit{values}\ [(\texttt{"0"},0)] = \mathit{values}\ [(\texttt{"1"},0)] = [0]$$

Also, the backward function additionally takes a lookup table as its first argument (the original source). The backward function uses this extra parameter to reconstruct (update) the source corresponding to the input updated view. Thus, the result depends on the input source.

$$\begin{aligned}
&\mathit{source}_1 = [(\texttt{"0"},0)]\\
&\mathit{source}_2 = [(\texttt{"1"},0)]\\
&\mathit{view} = [1]\\
&\mathit{result}_1 = \mathit{backward}\ \mathit{source}_1\ \mathit{view} = (1,1)\\
&\mathit{result}_2 = \mathit{backward}\ \mathit{source}_2\ \mathit{view} = (1,2)
\end{aligned}$$

Since the input original sources are different, for a single view, there are different results.

2.2.1 Correctness Laws

A bidirectional transformation is correct if the following properties hold [FMV12]:

Consistency–PutGet $get\ (put\ s\ v) = v$ (figure 2.3)

Acceptability–GetPut $put\ s\ (get\ s) = s$ (figure 2.4)

In addition to these laws, an optional *undoability* property is sometimes introduced:

Undoability–PutPut $put\ (put\ s\ v')\ (get\ s) = s$ (figure 2.5)

Note that we assume that the property $\forall x\ y.((x == y) = \mathit{True}) \Leftrightarrow (x = y)$ holds throughout the thesis and the operator == is yet another Haskell function and does not denote equality; the operator = is used to denote equality and binding.

Consistency property ensures that all the updates on a view are captured by the updated source.

Acceptability property states if there are no changes to the view, only the original source should be retrieved by the backwards function.

Undoability property states that the order of updates on a single source should not matter.

In practice, it is often allowed for the backward function to fail on certain inputs [FMV12]. In those cases, weakened versions of the consistency and undoability properties are introduced (the hypotheses test the definedness of specific function calls):

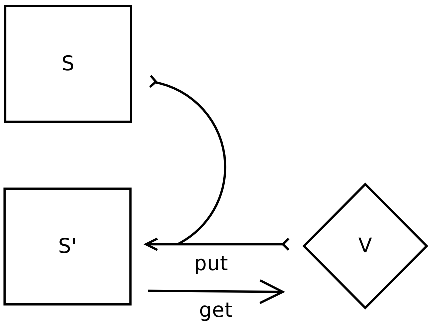


Figure 2.3: Consistency–PutGet Law

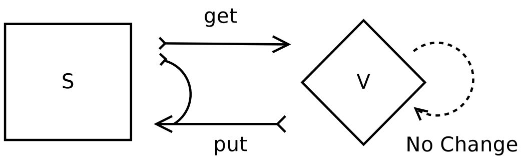


Figure 2.4: Acceptability–GetPut Law

Weakened Consistency–Partial PutGet $\dfrac{(put\ s\ v)\!\downarrow}{get\ (put\ s\ v) = v}$

Weakened Undoability–Partial PutPut $\dfrac{(put\ s\ v')\!\downarrow}{put\ (put\ s\ v')\ (get\ s) = s}$

Note that in all the mentioned properties, the input value of the forward function ranges over the actual domain in which the forward function is total:

$$\forall s. get\ s \neq \bot$$

There are two main approaches to bidirectional programming [FMV12]:

1. lenses and other language-based approaches
2. bidirectionalization

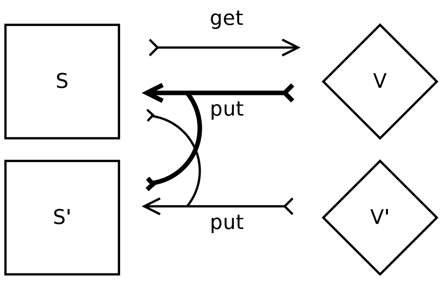


Figure 2.5: Undoability–PutPut Law

2.2.2 Lenses

In a language-based approach [FGM$^+$07], the programmer writes one single implementation in a special language and from that implementation both forward and backward functions are derived. That is, the programmer, by writing code in that language, is defining the forward and backward transformation at the same time.

Limitations

Although lenses are useful in practice, since they require the programmer to write the program in a specific language rather than a conventional language of choice, their application domain is limited. The existing programs written in conventional languages should be rewritten in order to form lenses.

2.2.3 Bidirectionalization

By employing bidirectionalization techniques, the programmer writes the forward function in a conventional language of choice and the backward function is mechanically derived.

There are two main distinct approaches to bidirectionalization:

1. *Syntactic-Bidirectionalization*: using the actual code describing the forward function to derive the backward function [MHN$^+$07]
2. *Semantic-Bidirectionalization*: using the information provided by the type and run-time behavior of the forward function to derive the backward function [Voi09]

Moreover, there is a technique that combines the two [VHMW10].

2.2.3.1 Syntactic-Bidirectionalization

Syntactic-bidirectionalization [MHN$^+$07] employs a syntax-directed transformation, over the actual code describing the forward function, to derive the backward function. The transformation is done in three steps:

1. The complement function – i.e. the function returning the parts of the input that are discarded by the forward function – is calculated
2. The complement function is tupled with the forward function. The tupled function is injective and hence invertible, since the information discarded by the forward function is provided by the complement function

3. the inverse function of the tupled function is calculated

Considering the function *values* from before as the forward function, syntactic bidirectionalization results in the following:

```
comp_values [] = []
comp_values ((i,x) : ps) = let
  is = comp_values ps
  in i : is

tupl_values [] = ([],[])
tupl_values ((i,x) : ps) = let
  (is,xs) = tupl_values ps
  in (i : is,x : xs)

invs_tupl_values ([],[]) = []
invs_tupl_values (i : is,x : xs) = let
  ps = invs_tupl_values (is,xs)
  in (i,x) : ps

backward_Syn :: ∀a.Source a → View a → Source a
backward_Syn source view′ = invs_tupl_values (comp_values source,view′)
```

where *comp_values* is the complement function and returns the list keys in the input lookup table that are discarded by the forward function. The function *tupl_values* tuples the forward function with the complement function. By swapping the patterns and corresponding expressions in *tupl_values* the inverse function *invs_tupl_values* is produced. The backward function applies the inverse function *invs_tupl_values* to the view and the complement that was discarded.

Limitations Since this technique involves calculation of the complement function and the inverse function of the tupled function, it is highly coupled with the definition of the language the program is written in [VHMW10, FMV12].

Quoting from [VHMW10]:

> [Syntactic-Bidirectionalization] can only deal with programs in a custom first-order language subject to linearity restrictions and absence of intermediate results between function calls.

In a first-order language, functions are no longer first-class citizens; it is not possible to abstract over functions. Linearity restriction (affine language) is a condition stating that each variable should only be used at most once. It means the program cannot duplicate values. For instance, it is impossible to define the function $dup\ x = x + x$. Absence of intermediate results (treeless) is another well-known restriction, discussed in [Wad88] for example.

2.2.3.2 Semantic-Bidirectionalization

Semantic-bidirectionalization [Voi09] technique targets polymorphic functions; it uses polymorphism to assign a unique index to each element in the input (indexes the elements of the input), executes the forward function over the indexed input and then studies the output to mimic the behavior of the forward function. Having Haskell extended with the support for *rank-2 types*, the corresponding backward function for a fully polymorphic function $f :: \forall a.[a] \to [a]$ is a higher-order function of the following type:

$$\begin{aligned}&\mathit{bff} :: (\forall a.[a] \to [a]) \to \\ &\quad (\forall a.Eq\ a \Rightarrow [a] \to [a] \to [a])\end{aligned}$$

The algorithm is best illustrated by an example. Considering the standard function *tail* $:: \forall a.[a] \to [a]$ as the forward function with the original source [`'c'`,`'a'`,`'a'`,`'b'`] and the modified view [`'a'`,`'a'`,`'d'`], the algorithm works as follows:

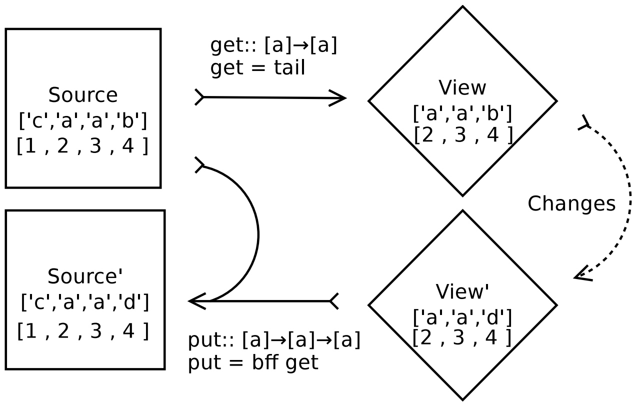


Figure 2.6: Semantic-Bidirectionalization, An Example

1. each element in the original source is uniquely indexed by numbers to form a mapping, from the unique indices to their corresponding source elements, called the *source mapping.*

   [(1,`'c'`),(2,`'a'`),(3,`'a'`),(4,`'b'`)]

2. the indices are extracted from the mapping and the forward function is applied to the indices

   tail [1,2,3,4] = [2,3,4]

3. the resulting indices

   [2,3,4]

are zipped with the modified view

`['a','a','d']`

to form the *view mapping*

[(2,`'a'`),(3,`'a'`),(4,`'d'`)]

4. it checks duplication in the view mapping by checking if any index is repeated more than once. If so, the corresponding values in the modified view should be the same. The duplication check on the view mapping [(2,'a'), (3,'a'),(4,'d')] finds no duplication since no index is repeated more than once.

5. the view mapping

   [(2,`'a'`),(3,`'a'`),(4,`'d'`)]

   is overwritten on the source mapping

   [(1,`'c'`),(2,`'a'`),(3,`'a'`),(4,`'b'`)]

   and results in

   [(1,`'c'`),(2,`'a'`),(3,`'a'`),(4,`'d'`)]

6. looking up the original indices

   [1,2,3,4]

   from the overwritten mapping

   [(1,`'c'`),(2,`'a'`),(3,`'a'`),(4,`'d'`)]

   results in the updated source

   `['c','a','a','d']`

By generic programming, the algorithm can be extended to work for any algebraic data types. [Voi09]

$\textbf{class}\ (\mathit{Traversable}\ k, \mathit{Foldable}\ k', \mathit{Zippable}\ k') \Rightarrow \mathit{Generic}\ k\ k'\ \textbf{where}\ \{\}$

$\mathit{bff} :: \forall k\ k'.\mathit{Generic}\ k\ k' \Rightarrow$
$\quad (\forall a.k\ a \to k'\ a) \to$
$\quad (\forall a.\mathit{Eq}\ a \Rightarrow k\ a \to k'\ a \to k\ a)$

For example, adopting the same technique for bidirectionalizing the function *values* from before, we get the higher-order function with the following type as the first approximation:

$\mathit{backward}_{\mathit{Sem}} :: (\forall a.[(\mathit{String},a)] \to [a]) \to$
$\quad (\forall a.\mathit{Eq}\ a \Rightarrow [(\mathit{String},a)] \to [a] \to [(\mathit{String},a)])$

Limitations The original technique only works for polymorphic functions f with the following types:

$$\textbf{class}\ (\mathit{Traversable}\ k, \mathit{Foldable}\ k', \mathit{Zippable}\ k') \Rightarrow \mathit{Generic}\ k\ k'\ \textbf{where}\ \{\,\}$$

1. fully polymorphic

   $$f :: \forall k\ k'\ a. \mathit{Generic}\ k\ k' \Rightarrow k\ a \rightarrow k'\ a$$

2. polymorphic function constrained with equality constraint

   $$f :: \forall k\ k'\ a. (\mathit{Generic}\ k\ k', \mathit{Eq}\ a) \Rightarrow k\ a \rightarrow k'\ a$$

3. polymorphic function constrained with an ordering constraint

   $$f :: \forall k\ k'\ a. (\mathit{Generic}\ k\ k', \mathit{Ord}\ a) \Rightarrow k\ a \rightarrow k'\ a$$

Moreover, semantic-bidirectionalization does not support shape change in the modified view, e.g., considering the *tail* example above, if the length of the list in the modified view changes, it cannot be bidirectionalized. The reason is the fact that mappings are not of the same length, thus they cannot be zipped together. In the next chapter, we generalize the underlying theory of semantic-bidirectionalization and, as a result, we expand the application domain.

2.2.3.3 Syntactic and Semantic Combined

As mentioned in the previous section, semantic-bidirectionalization rejects any updates to the shape of the view. This lack of "updateability" can be compensated by combining [VHMW10] the semantic-bidirectionalization with syntactic-bidirectionalization techniques.

Limitations Since this technique is a combination of semantic and syntactic bidirectionalization, it inherits limitations of both approaches. Therefore, in practice it has a small application domain.

Chapter 3

Semantic Bidirectionalization Revisited

3.1 Parametricity and Polymorphism

The Abstraction theorem (Parametricity) [Rey83] was originally introduced to capture the intuitive understanding that parametric polymorphic functions should behave independently (abstracted) from the type assigned to the type variables. It can be used to derive theorems (*free theorems*) for every function with a polymorphic type [Wad89]. For example, the function $get :: \forall a.[a] \rightarrow [a]$ should treat a list of Boolean values of the type $[Bool]$ in the same way as it treats a list of characters of the type $[Char]$. In other words, the function *get* has no information about the type of the elements of the input list and this lack of information (the abstraction) limits the function in the way it treats the argument; variables with parametric types cannot be generated and cannot be observed (e.g. pattern matched over) except by the input function arguments sharing the same type variables. To generate a value of the parametric type a inside a parametric polymorphic function, the function argument should result in a value of the same parametric type, i.e., the function argument should be of the type $... \rightarrow a$. Such a function is called the *generator* function [BJC10]. For example, the function argument in $scanr1 :: \forall a.(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]$ is a generator function.

Analogously, function arguments of type $... \rightarrow X$, where X is a (non-parametric) monomorphic type, are called the *observer* functions since by applying them to variables of parametric type, a value with a concrete type is produced that can be observed in the function. For example, the function argument in $nubBy :: \forall a.(a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$ is an observer function.

3.2 Original Algorithm

According to the abstraction theorem (refer to the previous section), a parametric polymorphic forward function $get :: \forall a.[a] \rightarrow [a]$, due to the lack of generator functions, can only do the following actions with the elements of the input list [Wad89]:

1. Rearranging
2. Dropping
3. Duplicating

Semantic bidirectionalization uses this fact to simulate the behavior of the forward function *get* by running it on a list of unique integers with the same length and examining the result. Each of the above mentioned effects of the function on the elements of the input list is revealed as follows:

1. *Rearranging*: the rearrangement of the numbers in the resulting list of integers with respect to the original list of integers

2. *Dropping*: missing numbers in the resulting list of integers with respect to the original list of integers

3. *Duplicating*: multiple equal elements in the resulting list of integers

Consider the following function that concatenates the tail of the input list with the mirror of its tail:

$$
\begin{aligned}
&\mathit{tailRevDup} :: \forall a.[\,a\,] \rightarrow [\,a\,]\\
&\mathit{tailRevDup}\ [\,] = [\,]\\
&\mathit{tailRevDup}\ x = \mathbf{let}\\
&\quad t = \mathit{tail}\ x\\
&\quad \mathbf{in}\ t +\!\!+ (\mathit{reverse}\ t)
\end{aligned}
$$

Consider the following original source:

$$
\begin{aligned}
&\mathit{Source}_{\mathit{Char}} = [\texttt{'c'},\texttt{'a'},\texttt{'a'},\texttt{'b'}]\\
&\mathit{View}_{\mathit{Char}} = \mathit{tailRevDup}\ \mathit{Source}_{\mathit{Char}}
\end{aligned}
$$

The resulting view $\mathit{View}_{\mathit{Char}}$ is `"aabbaa"`. Now, we apply the function to a list of unique numbers with the same size:

$$
\begin{aligned}
&\mathit{Source}_{\mathit{Int}} = [1,2,3,4]\\
&\mathit{View}_{\mathit{Int}} = \mathit{tailRevDup}\ \mathit{Source}_{\mathit{Int}}
\end{aligned}
$$

The resulting list of numbers $\mathit{View}_{\mathit{Int}}$ is [2,3,4,4,3,2]. The effect of the function *tailRevDup* on the source $\mathit{Source}_{\mathit{Char}}$ is revealed by studying the resulting list of numbers $\mathit{View}_{\mathit{Int}}$:

1. *Rearranging*: the rearrangement of the numbers in $\mathit{View}_{\mathit{Int}}$ with respect to $\mathit{Source}_{\mathit{Int}}$ exactly shows how corresponding characters in $\mathit{Source}_{\mathit{Char}}$ are rearranged.

2. *Dropping*: since number 1 is dropped, then it is possible to conclude the first character `'c'` is dropped by the function.

3. *Duplicating*: recurrence of numbers 2,3 and 4 indicates duplication of characters the first `'a'`, the second `'a'` and `'b'`.

3.2.1 Bidirectionalizing Fully Polymorphic Functions

In essence, the original semantic-bidirectionalization algorithm [Voi09] follows the above mentioned approach to bidirectionalize polymorphic functions. The original algorithm for bidirectionalizing fully polymorphic forward functions of type $\forall a.[\,a\,] \rightarrow [\,a\,]$ (e.g. *tailRevDup*) can be implemented as the following higher-order rank-2 function that takes the forward function as a function argument and returns the backward function as the result:

```
 {-# LANGUAGE Rank2Types #-}
import Data.Maybe
import Control.Monad
import qualified Data.List
import Data.Function


bff :: (∀a.[a] → [a]) →
  (∀a.Eq a ⇒ [a] → [a] → Either String [a])
bff get s v = do
    -- Step 1
  let ms = index s
    -- Step 2
  let is  = fst `map` ms
  let iv  = get is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms' = union mv ms
    -- Step 5.1
  unless (check ms')
    $ Left "Invalid modified view!"
    -- Step 6
  return $ lookupAll is ms'
```

As discussed before, it includes six steps (section 2.2.3.2). For a more illustrative and self-contained explanation, we combine the description of each step, the corresponding code and a step by step bidirectionalization of the function *tail* :: $\forall a.[a] \rightarrow [a]$ with s = `['c','a','a','b']` as the original source and v = `['a','a','d']` as the modified view (figure 2.6). The code is defined inside the *do notation* block to facilitate error handling using *Either Monad* and the monadic function *unless* [Wad95, Wad92].

Step 1: Indexing

Each element in the original source is uniquely indexed by the integers to form the *source mapping.*

$$\begin{aligned}&\mathit{index} :: \forall a.[a] \to [(\mathit{Int},a)]\\&\mathit{index}\ s = \mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s\end{aligned}$$

In the example above, after *step 1* we have:

$$\mathit{ms} = [(1,\texttt{'c'}),(2,\texttt{'a'}),(3,\texttt{'a'}),(4,\texttt{'b'})]$$

Step 2: Calculating View of Indices

The indices are extracted from the mapping and forward function is applied to the indices.

In the example above, after *step 2* we have:

$$\begin{aligned}&\mathit{is} = [1,2,3,4]\\&\mathit{iv} = [2,3,4]\end{aligned}$$

Step 3: Associating

The resulting view of indices are zipped with the input modified view v to form the *view mapping.* Since the two lists are zipped together, it is checked if they have the same length. The standard *zip* function in Haskell, imported in the *Prelude* module, can zip lists of different lengths; every element in the shorter list is paired with a corresponding element (the same position) in the larger list and the rest of the elements of the larger list are discarded. We expect all the elements in the two lists to be paired up. Therefore, lists of different lengths cannot be zipped together correctly.

$$\begin{aligned}&\mathit{assoc} :: \forall a\ b.[a] \to [b] \to [(a,b)]\\&\mathit{assoc} = \mathit{zip}\end{aligned}$$

In the example above, after *step 3* we have:

$$\begin{aligned}&(\mathit{length}\ v == \mathit{length}\ \mathit{iv}) = \mathit{True}\\&\mathit{mv} = [(2,\texttt{'a'}),(3,\texttt{'a'}),(4,\texttt{'d'})]\end{aligned}$$

Step 4: Duplication Check

The duplication check is performed to ensure that duplicated indices are assigned to the same value. Since this check needs to compare the value of the elements in the input modified view, the type of elements is constrained with an equality constraint. For example, having the forward function $\mathit{get}\ x = x +\!\!+\, x$ with the input source `"a"`, the modified view `"ab"` is indeed invalid since all the possible updates of the source, namely `"a"` or `"b"`, violate the consistency law.

$$\begin{array}{l} \mathit{validAssoc} :: \forall a\ b.(\mathit{Eq}\ a,\mathit{Eq}\ b) \Rightarrow [(a,b)] \rightarrow \mathit{Bool} \\ \mathit{validAssoc}\ \mathit{mv} = \mathit{and}\ [\neg\ (i == j) \vee x == y \mid (i,x) \leftarrow \mathit{mv},(j,y) \leftarrow \mathit{mv}] \end{array}$$

In the example above, after *step 4* we have:

$$\mathit{validAssoc}\ \mathit{mv} = \mathit{True}$$

Step 5: Union

The mappings are unified with the view mapping having the highest priority.

$$\begin{array}{l} \mathit{union} :: \forall a\ b.\mathit{Eq}\ a \Rightarrow [(a,b)] \rightarrow [(a,b)] \rightarrow [(a,b)] \\ \mathit{union} = \mathit{unionBy}\ ((==)\ \text{`}\mathit{on}\text{`}\ \mathit{fst}) \end{array}$$

In the example above, after *step 5* we have:

$$\mathit{ms}' = [(2,\texttt{'a'}),(3,\texttt{'a'}),(4,\texttt{'d'}),(1,\texttt{'c'})]$$

Step 5.1: Union Validity Checking

The output of the union is checked to be valid. In the fully polymorphic case (*bff*), the check always passes. It is used as a place holder; soon, by extending the algorithm to bidirectionalize polymorphic functions constrained with equality or ordering, we need to introduce an actual validation phase here.

$$\begin{array}{l} \mathit{check} :: \forall a\ b.[(a,b)] \rightarrow \mathit{Bool} \\ \mathit{check}\ \_ = \mathit{True} \end{array}$$

In the example above, after *step 5.1* we have:

$$\mathit{check}\ \mathit{ms}' = \mathit{True}$$

Step 6: Looking Up

The actual updated source is formed by looking up the original indices from the unified mapping.

$$\begin{array}{l} \mathit{lookupAll} :: \forall a\ b.\mathit{Eq}\ a \Rightarrow [a] \rightarrow [(a,b)] \rightarrow [b] \\ \mathit{lookupAll}\ \mathit{is}\ \mathit{mp} = \mathit{map}\ (\mathit{fromJust}.\mathit{flip}\ \mathit{lookup}\ \mathit{mp})\ \mathit{is} \end{array}$$

Finally, in the example above, after *step 6* we have:

$$\mathit{result} = [\texttt{'c'},\texttt{'a'},\texttt{'a'},\texttt{'d'}]$$

Correctness

The bidirectionalization via the function *bff* forms a valid bidirectional transformation with respect to BX laws:

Theorem 1 (Consistency of *bff*)

The bidirectional transformation formed by bff is consistent:

$$\mathit{bff}\ \mathit{get}\ s\ v = \mathit{Right}\ s' \Rightarrow \mathit{get}\ s' = v$$

□

PROOF.

Refer to the proof of theorem 2 in the original paper [Voi09].

□

Theorem 2 (Acceptability of *bff*)

The bidirectional transformation formed by bff is acceptable:

$$\mathit{bff}\ \mathit{get}\ s\ (\mathit{get}\ s) = \mathit{Right}\ s$$

□

PROOF.

Refer to the proof of theorem 1 in the original paper [Voi09].

□

3.2.2 Bidirectionalizing Functions with an Equality Constraint

The algorithm was originally extended [Voi09] to work for forward functions with an equality constraint, namely bff_{Eq} :

$$\begin{aligned}\mathit{bff}_{Eq} &:: (\forall a.Eq\ a \Rightarrow [\,a\,] \to [\,a\,]) \to \\ &\quad\ (\forall a.Eq\ a \Rightarrow [\,a\,] \to [\,a\,] \to [\,a\,])\end{aligned}$$

For example, in order to bidirectionalize the function $nub :: Eq\ a \Rightarrow [\,a\,] \rightarrow [\,a\,]$, we have to use bff_{Eq} instead of *bff*. The function *nub* is imported from the module *Data.List* in *Haskell's* standard library and it removes duplication from the input list.

For a function like *nub*, it is not enough to use *bff*; even the types do not match, since *nub* has an equality constraint on its type. To illustrate the problem, we consider a version of *bff*, with only its type changed to include the equality constraint for the function argument:

$$\begin{aligned}&bff_{v1} :: (\forall a.Eq\ a \Rightarrow [\,a\,] \rightarrow [\,a\,]) \rightarrow \\ &\quad (\forall a.(Eq\ a, Eq\ a) \Rightarrow [\,a\,] \rightarrow [\,a\,] \rightarrow Either\ String\ [\,a\,])\end{aligned}$$

The following example fails for bff_{v1} :

```
ghci > bff_v1 nub "aa" "b"
Left "Modified view of wrong length!"
```

Here, the original view would be "a" and the user modified it to "b". bff_{v1} works as follows:

Step 1

$ms = [\,(1,\texttt{'a'}),(2,\texttt{'a'})\,]$

Step 2

$is = [\,1,2\,]$
$iv = [\,1,2\,]$

Step 3

$(length\ v == length\ iv) = False$

The length of the view of the indices (length 2) is not equal to the length of the input modified view (length 1). It fails and returns the error message as the result (wrapped by *Left*).

The main problem is that *nub* is capable of observing equality between the elements of the input list (via the (==) operator provided by the witness of the equality constraint) and hence behaves according to that observation. Naive indexing of original source values with a list of unique numbers ignores the fact that some elements are equal to each other and hence applying *nub* to a list of unique indices can no longer mimic its effect on the actual source. In other words, if the equality of elements can be observed in a mapping (*mp*), indices should be equal whenever their corresponding elements are equal:

$\forall (i,x),(j,y) \in mp.\ i == j \Leftrightarrow x == y$

To solve this issue, the original work [Voi09] uses the same algorithm as *bff* but with a different indexing function (the function $index_{Eq}$ instead of *index*) and a stronger union check $check_{Eq}$.

The new indexing function $index_{Eq}$ assigns equal indices to equal elements; for example having the source `"caab"`, the new indexing function $index_{Eq}$ results in the following:

```
ghci > index_Eq "caab"
[(1,'c'),(2,'a'),(2,'a'),(3,'b')]
```

While the original indexing function results in a wrong mapping:

```
ghci > index "caab"
[(1,'c'),(2,'a'),(3,'a'),(4,'b')]
```

The original work [Voi09], uses the *State Monad* [Wad92] to implement the new indexing function $index_{Eq}$. For simplicity of the presentation, we sacrifice efficiency and redefine the original algorithm as follows:

```
index_Eq :: ∀a.Eq a ⇒ [a] → [(Int,a)]
index_Eq s = index_Eq s [] 0
index_Eq :: ∀a.Eq a ⇒ [a] → [(Int,a)] → Int → [(Int,a)]
index_Eq [] mp _ = mp
index_Eq (x : xs) mp i = let
  (i',ix) = case (find ((== x).snd) mp) of
    Just (j,_) → (i,j)
    Nothing → (i + 1,i + 1)
  in index_Eq xs (mp ++ [(ix,x)]) i'
```

If an equal element is already indexed, then it assigns the same index, otherwise it assigns a new index.

In addition to a new indexing function, we need to introduce a new mechanism to check validity of the mapping after union. First, let us demonstrate the necessity by an example.

Considering bff_{v2} being the same as bff_{v1} but with the new indexing function $index_{Eq}$, we have:

```
ghci > bff_v2 nub "ab" "aa"
"aa"
```

It certainly violates the *PutGet* law (section 2.2.1):

$$nub\ (bff_{v2}\ \ nub\ \texttt{"ab"}\ \texttt{"aa"}) = \texttt{"a"} \neq \texttt{"aa"}$$

It boils down to the very same fact that in presence of an equality operator as an observer function, equal elements in the mappings should have equal indices and vise versa. So far, we enforced this property in generating mappings ($index_{Eq}$); we also need to make sure this correspondence between indices and values in the mapping still holds after union. We encourage the reader to refer to the original paper [Voi09] in which, as a part of the proof of consistency of the bidirectionalization formed via bff_{Eq} , the necessity of this correspondence is explained.

To validate the unified mapping (the mapping after the union), we provide a new function $check_{Eq}$:

```
check_Eq :: ∀a.Eq a ⇒ [(Int,a)] → Bool
check_Eq mp = and [(i == j) == (x == y) | (i,x) ← mp,(j,y) ← mp]
```

Replacing the old validity check function *check* in bff_{v2} with the new version $check_{Eq}$, we finally get bff_{Eq} to bidirectionalize polymorphic functions of type $\forall a.Eq\ a \Rightarrow [a] \rightarrow [a]$:

```
bff_Eq :: (∀a.Eq a ⇒ [a] → [a]) →
  (∀a.Eq a ⇒ [a] → [a] → Either String [a])
bff_Eq get s v = do
    -- Step 1
  let ms = index_Eq s
    -- Step 2
  let is  = fst `map` ms
  let iv  = get is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms' = union mv ms
    -- Step 5.1
  unless (check_Eq ms')
    $ Left "Invalid modified view!"
    -- Step 6
  return $ lookupAll is ms'
```

By running bff_{Eq} with the invalid modified view from the example above, this time, we get an error message which correctly stops us from updating the source with an invalid change:

```
ghci > bff_Eq nub "ab" "aa"
Left "Invalid modified view!"
```

Correctness

The bidirectionalization via the function bff_{Eq} forms a valid bidirectional transformation with respect to BX laws:

Theorem 3 (Consistency of bff_{Eq})

The bidirectional transformation formed by bff_{Eq} *is consistent:*

$$\mathit{bff}_{Eq}\ \mathit{get}\ s\ v = \mathit{Right}\ s' \Rightarrow \mathit{get}\ s' = v$$

□

Proof.

Refer to the proof of theorem 4 in the original paper [Voi09].

□

Theorem 4 (Acceptability of bff_{Eq})

The bidirectional transformation formed by bff_{Eq} *is acceptable:*

$$\mathit{bff}_{Eq}\ \mathit{get}\ s\ (\mathit{get}\ s) = \mathit{Right}\ s$$

□

Proof.

Refer to the proof of theorem 3 in the original paper [Voi09].

□

3.2.3 Bidirectionalizing Functions with an Ordering Constraint

To bidirectionalize a forward function with an ordering constraint, i.e. a function of type $\forall a. Ord\ a \Rightarrow [a] \rightarrow [a]$, the original paper introduces a third function bff_{Ord}. Like for bff_{Eq}, bff_{Ord} needs to have its own separate mechanism for indexing ($index_{Ord}$) and checking the validity of the unified mapping ($check_{Ord}$). A function of type $\forall a. Ord\ a \Rightarrow [a] \rightarrow [a]$ can check for ordering of two elements in its arguments via the operator ($\leqslant$). In addition, it can check for equality via ($==$). That is because the type class *Eq* is defined as a super-class of the type class *Ord*:

class *Eq a* ⇒ *Ord a* **where** ...

The function *compare* introduced by an ordering constraint *Ord* can check both equality and ordering of the elements.

With such a powerful observational ability, it is more tricky for the indexing algorithm to calculate indices that follow the same ordering as their corresponding elements. For example, indexing `['c','a','a','b']` with [1,2,2,3] is not correct anymore, since for the elements 'b' and 'c' the function can observe `'b'` < `'c'` while for their corresponding indices (3 and 1 respectively), the function observes otherwise, i.e., $3 \not< 1$. A correct indexing for `['c','a','a','b']` would be [3,1,1,2]. The original paper uses *Applicative Functors* [MP08] to implement such an indexing system. For simplicity, we redefine it as follows:

```
$index_{Ord}$ :: $\forall a. Ord\ a \Rightarrow [a] \rightarrow [(Int,a)]$
$index_{Ord}$ $s$ = let
  $s' = sort\ s$
  $mp = index_{Eq}\ s'$
  in $[(fst.fromJust\ \$\ find\ ((== e).snd)\ mp, e) \mid e \leftarrow s]$
```

It first sorts the source list, employs $index_{Eq}$ for indexing the sorted list and then assigns indices to the elements of the original source (unsorted list) by looking up indices associated with each element in the mapping of the sorted list.

The same story repeats for checking validity of the mapping after union. The algorithm should check that indices compare to each other in the same way as their corresponding elements do.

```
$check_{Ord}$ :: $\forall a. Ord\ a \Rightarrow [(Int,a)] \rightarrow Bool$
$check_{Ord}$ $mp = and\ [compare\ i\ j == compare\ x\ y \mid (i,x) \leftarrow mp, (j,y) \leftarrow mp]$
```

Finally, bff_{Ord} [1] is formed by modifying the type of *bff* to include *Ord* constraint, replacing *index* with index_{Ord} and *check* with check_{Ord} :

```
bff_Ord :: (∀a.Ord a ⇒ [a] → [a]) →
  (∀a.(Eq a,Ord a) ⇒ [a] → [a] → Either String [a])
bff_Ord get s v = do
    -- Step 1
  let ms = index_Ord s
    -- Step 2
  let is  = fst ‘map‘ ms
  let iv  = get is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms′ = union mv ms
    -- Step 5.1
  unless (check_Ord ms′)
    $ Left "Invalid modified view!"
    -- Step 6
  return $ lookupAll is ms′
```

Correctness

The bidirectionalization via the function bff_{Ord} forms a valid bidirectional transformation with respect to BX laws:

Theorem 5 (Consistency of bff_{Ord})

The bidirectional transformation formed by bff_{Ord} is consistent:

$$\mathit{bff}_{Ord}\ \mathit{get}\ s\ v = \mathit{Right}\ s' \Rightarrow \mathit{get}\ s' = v$$

□

[1] The type class *Ord* inherits the type class *Eq*. Therefore, we could reduce the constraint $(Eq\ a, Ord\ a)$ to $Ord\ a$.

PROOF.

Refer to the proof of theorem 6 in the original paper [Voi09].

□

Theorem 6 (Acceptability of bff_{Ord})

The bidirectional transformation formed by bff_{Ord} is acceptable:

$$\mathit{bff}_{Ord}\ \mathit{get}\ s\ (\mathit{get}\ s) = \mathit{Right}\ s$$

□

PROOF.

Refer to the proof of theorem 5 in the original paper [Voi09].

□

3.3 Generalized Algorithm

As discussed in the previous section, the original technique is heavily based on a mechanism to generate unique indices and for every different type signature (different constraint) it has to provide a new indexing mechanism and a new validity checking function. Hence, the original method does not scale properly; it fails to bidirectionalize functions such as $\mathit{filter} :: \forall a.(a \to \mathit{Bool}) \to [a] \to [a]$.

3.3.1 Generalizing to Higher-Order Format

In order to gain generality, we rewrite constrained polymorphic functions as their equivalent high-order functions where constraints are translated to dictionaries. Dictionaries are normal function arguments witnessing the constraints.

For example, a function with equality constraint can be rewritten as follows:

$$\begin{array}{lll} f :: \forall a.\mathit{Eq}\ a \Rightarrow T & \hookrightarrow & f' :: \forall a.(a \to a \to \mathit{Bool}) \to T \\ f = f'\ (==) & & \end{array}$$

In the same way, a function with an ordering constraint can be rewritten as follows:

$$\begin{array}{lll} f :: \forall a.\mathit{Ord}\ a \Rightarrow T & \hookrightarrow & f' :: \forall a.(a \to a \to \mathit{Ordering}) \to T \\ f = f'\ \mathit{compare} & & \end{array}$$

A more general form that subsumes the two (and much more) can be expressed as the following, where ψ denotes a type class constraint that introduces (only) binary observer functions and its corresponding monomorphic type X which contains the *sum* of every possible observation:

$$f :: \forall a.\psi\ a \Rightarrow T \quad \hookrightarrow \quad f' :: \forall a.(a \to a \to X) \to T$$

For example, a function with an ordering constraint can also be rewritten as follows:

$$f :: \forall a.Ord\ a \Rightarrow T \quad \hookrightarrow \quad f' :: \forall a.(a \to a \to (Bool,Bool)) \to T$$
$$f = f'\ (\lambda x\ y \to (x == y, x < y))$$

Now, we try to rewrite the existing algorithms from the previous section into this general form.

For example, bff_{Ord} can be rewritten as follows:

```
bff_OrdBy :: (∀a.(a → a → Ordering) → [a] → [a]) →
  (∀a.Eq a ⇒ (a → a → Ordering)
    → [a] → [a] → Either String [a])
bff_OrdBy get_By obs_X s v = do
    -- Step 1
  let ms = index_OrdBy obs_X s
    -- Step 2
  let is  = fst `map` ms
  let iv  = get_By compare is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms′ = union mv ms
    -- Step 5.1
  unless (check_OrdBy obs_X ms′)
    $ Left "Invalid modified view!"
    -- Step 6
  return $ lookupAll is ms′
```

Firstly, the type of the function is modified so that the observer function is passed as an explicit function argument (obs_X). Note that the observer function for indices is the *compare* function provided by the instance declaration of the type class *Ord* for the type *Int* in the module *Prelude*. For simplicity, the equality constraint required for duplication checking is not rewritten. All the other (internal) functions that share the same constraint (*Ord*) are rewritten accordingly, namely:

1. $get \hookrightarrow get_{By}$
2. $index_{Ord} \hookrightarrow index_{OrdBy}$
3. $check_{Ord} \hookrightarrow check_{OrdBy}$

In the same way, bff_{Eq}, $index_{Eq}$, $index_{Ord}$, $check_{Eq}$ and $check_{Ord}$ are rewritten as follows:

```
bff_EqBy :: (∀a.(a → a → Bool) → [a] → [a]) →
  (∀a.Eq a ⇒ (a → a → Bool)
    → [a] → [a] → Either String [a])
bff_EqBy get_By obs_X s v = do
    -- Step 1
  let ms = index_EqBy obs_X s
    -- Step 2
  let is  = fst `map` ms
  let iv  = get_By (==) is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms' = union mv ms
    -- Step 5.1
  unless ( check_EqBy obs_X ms')
    $ Left "Invalid modified view!"
    -- Step 6
  return $ lookupAll is ms'
```

```
index_EqBy :: ∀a.(a → a → Bool) → [a] → [(Int,a)]
index_EqBy obs_X s = index_EqBy obs_X s [] 0
index_EqBy :: ∀a.(a → a → Bool) → [a] → [(Int,a)]
                → Int → [(Int,a)]
```

```
$index_{EqBy}\ \_\ [\,]\ mp\ \_ = mp$
$index_{EqBy}\ obs_X\ (x : xs)\ mp\ i = \mathbf{let}$
  $(i',ix) = \mathbf{case}\ (find\ ((obs_X\ x).snd)\ mp)\ \mathbf{of}$
    $Just\ (j,\_) \rightarrow (i,j)$
    $Nothing \rightarrow (i+1,i+1)$
  $\mathbf{in}\ index_{EqBy}\ obs_X\ xs\ (mp +\!\!+ [(ix,x)])\ i'$
```

```
$index_{OrdBy}\ :: \forall a.(a \rightarrow a \rightarrow Ordering) \rightarrow [a] \rightarrow [(Int,a)]$
$index_{OrdBy}\ obs_X\ s = \mathbf{let}$
  $s' = sortBy\ obs_X\ s$
  $mp = index_{EqBy}\ (\lambda x\ y \rightarrow obs_X\ x\ y == EQ)\ s'$
  $\mathbf{in}\ [(fst.fromJust\ \$$
    $find\ ((\lambda x \rightarrow obs_X\ e\ x == EQ).snd)\ mp,e)$
     $\mid e \leftarrow s]$
```

```
$check_{EqBy}\ :: \forall a.(a \rightarrow a \rightarrow Bool) \rightarrow [(Int,a)] \rightarrow Bool$
$check_{EqBy}\ obs_X\ mp = and$
  $[(i == j) == (obs_X\ x\ y) \mid (i,x) \leftarrow mp,(j,y) \leftarrow mp]$
```

```
$check_{OrdBy}\ :: \forall a.(a \rightarrow a \rightarrow Ordering) \rightarrow [(Int,a)] \rightarrow Bool$
$check_{OrdBy}\ obs_X\ mp = and$
  $[compare\ i\ j == obs_X\ x\ y \mid (i,x) \leftarrow mp,(j,y) \leftarrow mp]$
```

Correctness

Since we followed the standard dictionary translation that is automatically done by Haskell compilers (for more details refer to the theory of qualified types [Jon95]), the rewriting should not have changed the behavior of the functions:

Theorem 7 (Equivalence of bff_{Eq} and bff_{EqBy})

For every function $obs_X :: a \rightarrow a \rightarrow Bool$ such that $\forall x\ y.obs_X\ x\ y = x == y$ and for every pair of functions $f :: Eq\ a \Rightarrow [a] \rightarrow [a]$ and $f_{By} :: (a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$ such that $\forall x.f\ x = f_{By}\ obs_X\ x$, the following property holds:

$$\forall s\ v.bff_{Eq}\ f\ s\ v = bff_{EqBy}\ obs_X\ f_{By}\ s\ v$$

□

Proof.

No proof is included, as this transformation (dictionary translation) is well-known and standard; for more details refer to [Jon95].

□

Theorem 8 (Equivalence of bff_{Ord} and bff_{OrdBy})

For every function $obs_X :: a \to a \to Ordering$ such that $\forall x\ y. obs_X\ x\ y = compare\ x\ y$ and for every pair of functions $f :: Ord\ a \Rightarrow [a] \to [a]$ and $f_{By} :: (a \to a \to Ordering) \to [a] \to [a]$ such that $\forall x. f\ x = f_{By}\ obs_X\ x$, the following property holds:

$$\forall s\ v. \mathit{bff}_{Ord}\ f\ s\ v = \mathit{bff}_{OrdBy}\ obs_X\ f_{By}\ s\ v$$

□

Proof.

No proof is included, as this transformation (dictionary translation) is well-known and standard, for more details refer to [Jon95].

□

3.3.2 General Validity Check

Intuitively, by comparing the definitions of $check_{EqBy}$ and $check_{OrdBy}$, we can derive a more general property:

```
$check_{EqBy} :: \forall a. (a \to a \to Bool) \to [(Int,a)] \to Bool$
$check_{EqBy}\ obs_X\ mp = and$
  $[\ (==)\ i\ j == obs_X\ x\ y \mid (i,x) \leftarrow mp, (j,y) \leftarrow mp]$
```

```
$check_{OrdBy} :: \forall a. (a \to a \to Ordering) \to [(Int,a)] \to Bool$
$check_{OrdBy}\ obs_X\ mp = and$
  $[\ compare\ i\ j == obs_X\ x\ y \mid (i,x) \leftarrow mp, (j,y) \leftarrow mp]$
```

Condition 1 (Map Invariant)

A valid mapping $mp :: [(I,X)]$ *must satisfy the following property:*

$\forall (i_1,x_1),(i_2,x_2) \in mp.\ obs_X\ x_1\ x_2 = obs_I\ i_1\ i_2$

where $obs_X :: X \rightarrow X \rightarrow Z$ *is the observer function for the elements provided as an input and* $obs_I :: I \rightarrow I \rightarrow Z$ *is the equivalent observer function for indices that are provided by the indexing mechanism.*

□

It can be implemented as follows:

```
$check_{IBy} :: \forall x\ i\ a. Eq\ x \Rightarrow$
  $(i \rightarrow i \rightarrow x) \rightarrow (a \rightarrow a \rightarrow x) \rightarrow [(i,a)] \rightarrow Bool$
$check_{IBy}\ obs_I\ obs_X\ mp = and$
  $[(i\ `obs_I`\ j) == (x\ `obs_X`\ y) \mid (i,x) \leftarrow mp,(j,y) \leftarrow mp]$
```

For example, the validity check in presence of an order constraint ($check_{Ord}$) can be implemented using $check_{IBy}$:

```
$check_{Ord} :: \forall a. Ord\ a \Rightarrow [(Int,a)] \rightarrow Bool$
$check_{Ord} = check_{IBy}\ compare\ compare$
```

3.3.3 General Algorithm with Parametric Indexing

In the same fashion, it is possible to abstract over the indexing function. The mappings produced by this function should respect the *Map Invariant*. Also, the produced mappings should respect the rule of valid association (*validAssoc*); otherwise an unchanged view may fail the valid association test and hence violate the *GetPut* law.

Since the sole functionality of the indexing function is to assign indices to the elements of the input list, we expect the produced mappings to contain the elements of the original source without any change. We capture this property as follows:

Condition 2 (Input Preservation of Indexing Functions)

A valid indexing function index must satisfy the following property:

$\forall s.\ map\ snd\ (index\ s) = s$

□

Combining all the three mentioned validity properties for an indexing function, we define the condition for a valid indexing function as follows:

Condition 3 (Valid Indexing Function)

A valid indexing function index must satisfy all of the following properties:

1. *all the mapping ms = index s generated by the valid indexing function index should respect the* Map Invariant *condition*
2. *all the mapping ms = index s generated by the valid indexing function index should pass the association validity check*
3. *the valid indexing function index should respect the* Input Preservation *condition*

□

A function to check this condition can be implemented as follows:

```
validIndexing :: ∀x a i.(Eq x,Eq a,Eq i) ⇒
  ((a → a → x) → [a] → [(i,a)]) →
  (i → i → x) → (a → a → x) → [a] → Bool
validIndexing index_By obs_I obs_X s = let
  ms = index_By obs_X s
  in (check_IBy obs_I obs_X ms) ∧
     (validAssoc ms) ∧
     (map snd ms == s)
```

By abstracting over the indexing function in bff_{OrdBy} (or bff_{EqBy}), replacing the check function $check_{IBy}$ and putting a guard to check the validity of the mapping produced by the indexing function, we derive the following function:

```
bff_IBy :: ∀x i.(Eq x,Eq i) ⇒ (i → i → x) →
  (∀a.(a → a → x) → [a] → [(i,a)]) →
  (∀a.(a → a → x) → [a] → [a]) →
  (∀a.Eq a ⇒
     (a → a → x) → [a] → [a] → Either String [a])
bff_IBy obs_I index_By get_By obs_X s v = do
    -- Step 1
  let ms = index_By obs_X s
  unless (validIndexing index_By obs_I obs_X s)
     $ Left "Invalid indexing!"
  -- Step 2
let is  = fst `map` ms
let iv  = get_By (obs_I) is
  -- Step 3
unless (length v == length iv)
  $ Left "Modified view of wrong length!"
let mv = assoc iv v
  -- Step 4
unless (validAssoc mv)
  $ Left "Inconsistent duplicated values!"
  -- Step 5
let ms' = union mv ms
  -- Step 5.1
unless (check_IBy obs_I obs_X ms')
  $ Left "Invalid modified view!"
  -- Step 6
return $ lookupAll is ms'
```

This function subsumes *bff*,bff_{Eq} and bff_{Ord} (in the higher-order equivalent format), given the appropriate indexing functions.

Correctness

So far, we have rewritten the algorithm in a way that it is only parametric on the indexing function. It should work flawlessly with fully polymorphic functions, polymorphic functions constrained with an equality or ordering constraint (in the higher-order equivalent format). Now, the main concern is whether this algorithm works correctly with all the other functions with the type signature $\forall a.(a \to a \to X) \to [a] \to [a]$. In order to prove that the bidirectional transformation formed via the function bff_{IBy} respects the BX laws (section 2.2.1), we adopt the proof technique proposed in the original paper [Voi09]. It uses free theorems (refer to [Wad89] and [Voi09]) to reason about the functions. The only difference between the original system and our system is that in the original system the free theorem for functions explicitly specifies the observer functions $(==)$ and $(\leqslant)$ in its body whereas our system uses a function parameter that could be instantiated to $(==)$, $(\leqslant)$ or any other observer function. In the following, we prove that the bidirectional transformation formed via bff_{IBy} respects the *GetPut* law. We omit the proof for the *PutGet* law, since it follows the same style of reasoning (refer to appendix A in the original paper [Voi09]); it introduces no new proof techniques and it involves more steps of trivial reasoning.

Theorem 9 (Consistency of bff_{IBy})
Given a valid indexing function, the bidirectional transformation formed by bff_{IBy} *is consistent:*

$$\begin{aligned}&(\mathit{validIndexing}\ \mathit{index}_{By}\ \ \mathit{obs}_I\ \ \mathit{obs}_X\ \ s) \Rightarrow\\ &\quad \mathit{bff}_{IBy}\ \ \mathit{obs}_I\ \ \mathit{index}_{By}\ \ \mathit{get}_{By}\ \ \mathit{obs}_X\ \ s\ v = \mathit{Right}\ s' \Rightarrow \mathit{get}_{By}\ \ \mathit{obs}_X\ \ s' = v\end{aligned}$$

□

Proof.

We omit the proof for the *PutGet* law, since it follows the same style of reasoning as the *GetPut* law, theorem 10 (also, refer to appendix A in the original paper [Voi09]).

□

Theorem 10 (Acceptability of bff_{IBy})

Provided a valid indexing function, the bidirectional transformation formed by bff_{IBy} *is acceptable:*

$$\begin{aligned}&(\mathit{validIndexing}\ \mathit{index}_{By}\ \ \mathit{obs}_I\ \ \mathit{obs}_X\ \ s) \Rightarrow\\ &\quad \mathit{bff}_{IBy}\ \ \mathit{obs}_I\ \ \mathit{index}_{By}\ \ \mathit{get}_{By}\ \ \mathit{obs}_X\ \ s\ (\mathit{get}_{By}\ \ \mathit{obs}_X\ \ s) = \mathit{Right}\ s\end{aligned}$$

□

Proof.

Refer to the Appendix A.

□

3.4 General Indexing Mechanisms

So far, we studied rewriting by which we could abstract over the indexing function and we offered a uniform way of checking validity of mappings. In this section, we introduce a general algorithm to produce correct mappings in presence of an arbitrary observer function.

3.4.1 Self-Indexing System

Integers could be easily used to index elements with a polymorphic type constrained by *Eq* or *Ord*, since the type *Int* itself is an instance of *Eq* and *Ord* type classes. Numbers cannot be used to index elements in presence of a constraint which the type *Int* does not instantiate. The main quest is to find the proper index type that works in presence of an arbitrary constraint (an arbitrary observer function). One more obvious answer is to use the elements to index themselves. Such an indexing function can simply be defined as follows:

$$\begin{array}{l}
\mathit{index}_{GS}\ :: \forall a.[a] \to [(a,a)] \\
\mathit{index}_{GS}\ s = \mathit{zip}\ s\ s
\end{array}$$

Unfortunately, we cannot define the new algorithm (bff_{GS}) using bff_{IBy}:

$$\mathit{bff}_{GS}\ \mathit{get}_{By}\ f\ s\ v \neq \mathit{bff}_{IBy}\ f\ (\mathit{const}\ \mathit{index}_{GS})\ \mathit{get}_{By}\ f\ s\ v$$

The type of the new indexing function $\mathit{const}\ \mathit{index}_{GS}$ is too specific. Rewriting its type using the equality constraint [SPJCS08], we get the following type:

$$\begin{array}{l}
\mathit{const}\ \mathit{index}_{GS}\ :: \forall a\ b.b \to [a] \to [(a,a)] \\
\quad = \\
\mathit{const}\ \mathit{index}_{GS}\ :: \forall a\ b\ i.(i{\sim}a) \Rightarrow b \to [a] \to [(i,a)]
\end{array}$$

That is while the expected type can be viewed as a type equivalent to $\forall a\ b\ i.b \to [a] \to [(i,a)]$. The two types do not unify since the actual type has the extra equality constraint $i{\sim}a$ indicating the type of the index is the same as the type of the element.

This type mismatch is not an issue at all; we just need to manually put the new index function inside bff_{IBy} and set the observer function for indices (obs_I) equal to the observer function for the elements (obs_X), i.e. **let** $\mathit{obs}_I = \mathit{obs}_X$. We remove the valid indexing-function check *validIndexing*, since the indexing function *index* is a valid indexing function and the output of the check *validIndexing* would always pass:

$$\begin{array}{l}
\mathit{bff}_{GS}\ :: \forall x.(\mathit{Eq}\ x) \Rightarrow \\
\quad (\forall a.(a \to a \to x) \to [a] \to [a]) \to \\
\quad (\forall a.\mathit{Eq}\ a \Rightarrow \\
\qquad (a \to a \to x) \to [a] \to [a] \to \mathit{Either}\ \mathit{String}\ [a]) \\
\mathit{bff}_{GS}\ \mathit{get}_{By}\ \mathit{obs}_X\ s\ v = \mathbf{do} \\
\quad \mathbf{let}\ \mathit{obs}_I = \mathit{obs}_X \\
\qquad \text{-- Step 1} \\
\quad \mathbf{let}\ \mathit{ms} = \mathit{index}_{GS}\ s \\
\qquad \text{-- validIndexing is removed} \\
\qquad \text{-- Step 2} \\
\quad \mathbf{let}\ \mathit{is} = \mathit{fst}\ `\mathit{map}`\ \mathit{ms}
\end{array}$$

```
let iv  = get_By obs_I is
  -- Step 3
unless (length v == length iv)
  $ Left "Modified view of wrong length!"
let mv = assoc iv v
  -- Step 4
unless (validAssoc mv)
  $ Left "Inconsistent duplicated values!"
  -- Step 5
let ms' = union mv ms
  -- Step 5.1
unless (check_IBy obs_I obs_X ms')
  $ Left "Invalid modified view!"
  -- Step 6
return $ lookupAll is ms'
```

Correctness

Since bff_{GS} is built on top of bff_{IBy}, to prove its correctness we need to prove that its indexing function index_{GS} is a valid indexing function:

Theorem 11 (Validity of index_{GS})

The indexing function index_{GS} is a valid indexing function:

$$\mathit{validIndexing}\ (\lambda\_ \to \mathit{index}_{GS})\ \mathit{obs}_X\ \ \mathit{obs}_X\ \ s = \mathit{True}$$

$\square$

Proof.

$$
\begin{aligned}
& \mathit{validIndexing}\ (\lambda\_ \to \mathit{index}_{GS})\ \mathit{obs}_X\ \ \mathit{obs}_X\ \ s \\
=\ & \{\text{-definition of validIndexing -}\} \\
& (\mathit{check}_{IBy}\ \ \mathit{obs}_X\ \ \mathit{obs}_X\ \ (\mathit{zip}\ s\ s)) \wedge \\
& (\mathit{validAssoc}\ (\mathit{zip}\ s\ s)) \wedge \\
& (\mathit{map}\ \mathit{snd}\ (\mathit{zip}\ s\ s) == s) \\
=\ & \{\text{-specification of zip -}\} \\
& (\mathit{check}_{IBy}\ \ \mathit{obs}_X\ \ \mathit{obs}_X\ \ (\mathit{zip}\ s\ s)) \wedge \\
& (\mathit{validAssoc}\ (\mathit{zip}\ s\ s)) \\
=\ & \{\text{-specification of validAssoc -}\} \\
& (\mathit{check}_{IBy}\ \ \mathit{obs}_X\ \ \mathit{obs}_X\ \ (\mathit{zip}\ s\ s))
\end{aligned}
$$

= {-specification of checkIBy -}
  True

□

Theorem 12 (Consistency of bff_{GS})

The bidirectional transformation formed by bff_{GS} is consistent:

$$\mathit{bff}_{GS}\ \ \mathit{get}_{By}\ \ \mathit{obs}_X\ \ s\ v = \mathit{Right}\ s' \Rightarrow \mathit{get}_{By}\ \ \mathit{obs}_X\ \ s' = v$$

□

Proof.

By theorems 9 and 11, we conclude the bidirectional transformation formed by bff_{GS} is consistent.

□

Theorem 13 (Acceptability of bff_{GS})

The bidirectional transformation formed by bff_{GS} is acceptable:

$$\mathit{bff}_{GS}\ \ \mathit{get}_{By}\ \ \mathit{obs}_X\ \ s\ (\mathit{get}_{By}\ \ \mathit{obs}_X\ \ s) = \mathit{Right}\ s$$

□

Proof.

By theorems 10 and 11, we conclude that the bidirectional transformation formed by bff_{GS} is acceptable.

□

Unfortunately, bff_{GS} does not respect the undoability (PutPut) law [2].

[2]The undoability law is often considered [FMV12] an optional property of a bidirectional transformation.

Consider the following example where the bidirectional transformation formed via bff_{GS} breaks the undoability law:

```
ghci > :{
ghci | let get = tail
ghci | put       = bff_GS  (const get) (const $ const ())
ghci | s         = [0,1,2]
ghci | v         = [0,0]
ghci | Right put_s_v = put s v
ghci | get_s     = get s
ghci | in put (put_s_v) (get_s) == Right s
ghci | :}
False
```

Where *put_s_v* has the value of [0,0,0], *get_s* has the value of [1,2] and *put (put_s_v) (get_s)* has the value of *Left* `"Inconsistent duplicated values!"`.

The main problem is due to the fact that the equal indices originating from the different positions in the source are indistinguishable from the equal indices from the same origin. For example, having the source `"aa"` and the view `"aa"`, in a system in which values index themselves, we can assign two indistinguishable, yet different, semantics to the forward function, namely *reverse* or *id*. Therefore, the duplication check rejects inconsistent changes to equal values, whether or not they are from the same origin. Arguably, we would like to keep track of the origin of the elements. In the next section, we enhance bff_{GS} with the ability to track the origin of the elements.

3.4.2 Uniquely Self-Indexing System

In this system, like the system described in the previous section, each element indexes itself. In addition, each element is indexed with a unique number which makes it possible to keep track of the origin of the elements. The system can be viewed as a combination of the indexing function *index*, borrowed from *bff*, and using the corresponding original values themselves in observations, borrowed from bff_{GS}. The observer function for indices is implemented as follows:

```
obs_I = obs_X `on` (fromJust.(flip lookup ms))
```

where *ms* is the source mapping.

Putting these all together, bff_{GUS} is implemented by rewriting bff_{IBy} as follows:

```
bff_GUS :: ∀x.(Eq x) ⇒
  (∀a.(a → a → x) → [a] → [a]) →
  (∀a.Eq a ⇒
    (a → a → x) → [a] → [a] → Either String [a])
```

```
bff_GUS get_By obs_X s v = do
    -- Step 1
  let ms = index s
  let obs_I = obs_X `on` (fromJust.(flip lookup ms))
    -- validIndexing is removed
    -- Step 2
  let is  = fst `map` ms
  let iv  = get_By obs_I is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms' = union mv ms
    -- Step 5.1
  unless (check_IBy obs_I obs_X ms')
    $ Left "Invalid modified view!"
    -- Step 6
  return $ lookupAll is ms'
```

Having each element additionally indexed with its position, we expect the new system to respect the undoability law. We can try the counter-example from before:

```
ghci > :{
ghci | let get = tail
ghci | put     = bff_GUS (const get) (const $ const ())
ghci | s       = [0,1,2]
ghci | v       = [0,0]
ghci | Right put_s_v = put s v
ghci | get_s   = get s
ghci | in put (put_s_v) (get_s) == Right s
ghci | :}
True
```

Where *put_s_v* has the value of [0,0,0], *get_s* has the value of [1,2] and emphput (put_s_v) (get_s) has the value of *Right* [0,1,2].

Correctness

Since bff_{GUS} is built on top of bff_{IBy}, to prove its correctness we need to prove that its indexing function *index* is a valid indexing function:

Theorem 14 (Validity of *index*)
The indexing function index in bff_{GUS} *is a valid indexing function:*

$$\mathit{validIndexing}\ (\lambda\_ \rightarrow \mathit{index})\ \mathit{obs}_I\ \ \mathit{obs}_X\ \ s = \mathit{True}$$

where

$$\mathit{obs}_I\ = \mathit{obs}_X\ \text{`}\mathit{on}\text{`}\ (\mathit{fromJust}.(\mathit{flip}\ \mathit{lookup}\ (\mathit{index}\ s)))$$

□

Proof.

$$\begin{array}{l} \quad \mathit{validIndexing}\ (\lambda\_ \rightarrow \mathit{index})\ \mathit{obs}_I\ \ \mathit{obs}_X\ \ s \\ =\ \ \text{\{-definition of validIndexing -\}} \\ \quad (\mathit{check}_{IBy}\ \ \mathit{obs}_I\ \ \mathit{obs}_X\ \ (\mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s)) \wedge \\ \quad (\mathit{validAssoc}\ (\mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s)) \wedge \\ \quad (\mathit{map}\ \mathit{snd}\ (\mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s) == s) \\ =\ \ \text{\{-specification of zip -\}} \\ \quad (\mathit{check}_{IBy}\ \ \mathit{obs}_I\ \ \mathit{obs}_X\ \ (\mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s)) \wedge \\ \quad (\mathit{validAssoc}\ (\mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s)) \\ =\ \ \text{\{-specification of validAssoc -\}} \\ \quad \mathit{check}_{IBy}\ \ \mathit{obs}_I\ \ \mathit{obs}_X\ \ (\mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s) \\ =\ \ \text{\{-definition of obsI and Index -\}} \\ \quad \mathit{check}_{IBy}\ \ (\mathit{obs}_X\ \text{`}\mathit{on}\text{`}\ (\mathit{fromJust}.(\mathit{flip}\ \mathit{lookup}\ (\mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s)))) \\ \qquad \mathit{obs}_X\ \ (\mathit{zip}\ [1\,..\,\mathit{length}\ s]\ s) \\ =\ \ \text{\{-definition of checkIBy -\}} \\ \quad \mathit{True} \end{array}$$

□

Theorem 15 (Consistency of bff_{GUS})

The bidirectional transformation formed by bff_{GUS} *is consistent:*

$$\mathit{bff}_{GUS}\ \ \mathit{get}_{By}\ \ \mathit{obs}_X\ \ s\ v = \mathit{Right}\ s' \Rightarrow \mathit{get}_{By}\ \ \mathit{obs}_X\ \ s' = v$$

□

Proof.

By theorems 9 and 14, we conclude the bidirectional transformation formed by bff_{GUS} is consistent.

□

Theorem 16 (Acceptability of bff_{GUS})

The bidirectional transformation formed by bff_{GUS} *is acceptable:*

$$\mathit{bff}_{GUS}\ \mathit{get}_{By}\ \mathit{obs}_X\ s\ (\mathit{get}_{By}\ \mathit{obs}_X\ s) = \mathit{Right}\ s$$

□

Proof.

By theorems 10 and 14, we conclude the bidirectional transformation formed by bff_{GUS} is acceptable.

□

3.5 Arity of the Observer Function

So far, we only considered the forward function with an observer function of the type $a \to a \to X$ where a is the polymorphic type of the elements of the source and X is a concrete type. Our algorithm can also work for observer functions of any fixed arity denoted as $a \to \ldots \to a \to X$. The conditions necessary for a valid indexing function remain the same with the difference that now the map invariant (check_{IBy}) has to check the correspondence between indices and the elements slightly differently:

Condition 4 (Arity-Generic Map Invariant)

A valid mapping $\mathit{mp} :: [(I,X)]$ *must satisfy the following property:*

$$\forall \overline{(i,x)} \in \mathit{mp}.\ \mathit{obs}_X\ \overline{x} = \mathit{obs}_I\ \overline{i}$$

where $\mathit{obs}_X :: \overline{X} \to X$ *is the observer function for the elements provided as an input and* $\mathit{obs}_I :: \overline{I} \to X$ *is the equivalent observer function for indices provided by the indexing mechanism.*

□

For example, the equivalent version of check_{IBy} for observer functions of arity one can be implemented as follows:

```
$\mathit{check}^1_{IBy}$ :: $\forall x\ i\ a. Eq\ x \Rightarrow$
  $(i \to x) \to (a \to x) \to [(i,a)] \to \mathit{Bool}$
$\mathit{check}^1_{IBy}\ \mathit{obs}_I\ \mathit{obs}_X\ \mathit{mp} = \mathit{and}$
  $[(\mathit{obs}_I\ i) == (\mathit{obs}_X\ x) \mid (i,x) \leftarrow \mathit{mp}]$
```

3.5.1 Arity-Generic Data Type

In order to implement an arity-generic version of $check_{IBy}$, we first need to model the types of the form $a \rightarrow ... \rightarrow a \rightarrow X$. As the first step, we rewrite the type in the uncurried format $(a, ..., a) \rightarrow X$. The type $\underbrace{(a,\ ...,\ a)}_{\times n}$ forms a homogeneous tuple of arity n. A homogeneous tuple is a tuple whose every element has the same type and it is isomorphic to a vector of length n whose elements are of the polymorphic type a. A vector is a finite list whose length is known at the type level. For example, the type (a,a) is isomorphic to a vector of length 2 with elements of type a, denoted as *Vect* 2 *a*. In order to express length of a vector at the type level, we use *Peano* encoding, expressed as the following simple ADT declaration:

```
data Nat =
  Zero
  | Succ Nat
```

For example, using the Peano encoding, number 2 is encoded as *Succ* (*Succ Zero*) and the type *Vect* 2 *a* is rewritten as *Vect* (*Succ* (*Succ Zero*)) *a*. Cognoscenti will recognize a problem here–the data constructors *Zero* and *Succ* are used as type constructors! Thanks to a recent extension to Haskell (via the flag $-XDataKinds$ in *GHC*) [YWC+12], it is possible to promote simple ADT declarations to the type level, i.e., the data constructors in an ADT are promoted to act as type constructors and the type constructor of an ADT is promoted to act as a kind constructor. Consider the following definition of vectors using generalized algebraic data types (GADTs):

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}

infixr 5 :::
data Vect :: Nat → ∗ → ∗where
  Nil :: Vect Zero a
  (:::) :: a → Vect n a → Vect (Succ n) a
```

In the above, the first parameter of the type *Vect* is set to be of the (promoted) kind *Nat*.

We also derive the type class *Functor* for the type *Vect* as follows:

```
instance Functor (Vect n) where
  fmap _ Nil = Nil
  fmap f (x ::: xs) = f x ::: fmap f xs
```

Now, the size of a vector is only accessible at the type level. In order to be able to use this value at the term level, we use *singleton* types [EW12]. A *singleton* type has only one (besides bottom) inhabitant. A singleton for the Peano numeric type *Nat* can be expressed as follows:

```
data Sing_Nat :: Nat → ∗where
  Zero_Sing :: Sing_Nat Zero
  Succ_Sing :: Sing_Nat n → Sing_Nat (Succ n)
```

We also derive the type class *Show* for this type.

In order to get values of the type *Nat* out of the singleton type $Sing_{Nat}$, we define the following overloaded function using scoped type variables (via the extension $-XScopedTypeVariables$ in *GHC*) [PJS04]:

```
{-# LANGUAGE ScopedTypeVariables #-}

class SingI (n :: Nat) where
  sing :: Sing_Nat n

instance SingI Zero where
  sing = Zero_Sing

instance SingI n ⇒ SingI (Succ n) where
  sing = let
    n = (sing :: Sing_Nat n)
    in Succ_Sing n
```

Consider the following examples where the overloaded function *sing* is used to get the value of a singleton type:

```
ghci> sing :: SingNat 'Zero
SZero
ghci> sing :: SingNat ('Succ 'Zero)
SSucc SZero
```

Prefixing a constructor with a single quote is used for explicit promotion.

Now, we are ready to model the isomorphism with vectors by the following type class:

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE FlexibleContexts #-}
```

$$\begin{array}{l}
\textbf{class}\ (\mathit{SingI}\ (\mathit{Size}\ t)) \Rightarrow \mathit{Vect}_{\mathit{Iso}}\ \ (t :: * \to *)\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Size}\ t :: \mathit{Nat} \\
\quad \mathit{toVect} :: \forall a.t\ a \to \mathit{Vect}\ (\mathit{Size}\ t)\ a \\
\quad \mathit{fromVect} :: \forall a.\mathit{Vect}\ (\mathit{Size}\ t)\ a \to t\ a
\end{array}$$

Besides the size of the isomorphic vector, the isomorphism provides the pair of functions *toVect* and *fromVect* to handle conversions in either direction. Since, the relation is isomorphism[3], the functions should be each other's inverse functions:

Condition 5

For all members of the type class $\mathit{Vect}_{\mathit{Iso}}$ *, the function toVect is an inverse function of fromVect and vice versa (left and right invertibility):*

$$\mathit{toVect} \circ \mathit{fromVect} = \mathit{fromVect} \circ \mathit{toVect} = \mathit{id}$$

□

The isomorphism states that during the conversions no information is lost, i.e., a type carries exactly the same information as its isomorphic vector does.

Using the singleton type $\mathit{Sing}_{\mathit{Nat}}$, we can demote (bringing a value from the type level to the term level) the type-level information about the size of a vector to the term level:

$$\begin{array}{l}
\mathit{size} :: \forall a\ t.(\mathit{SingI}\ (\mathit{Size}\ t), \mathit{Vect}_{\mathit{Iso}}\ \ t) \Rightarrow \\
\qquad t\ a \to \mathit{Sing}_{\mathit{Nat}}\ \ (\mathit{Size}\ t) \\
\mathit{size}\ \_ = \mathit{sing}
\end{array}$$

3.5.2 Arity-Generic Observations

In order to implement an arity-generic version of $\mathit{check}_{\mathit{IBy}}$, we need to compute a matrix in n dimensions, containing all the possible combinations of n copies of the source mapping where n is the arity of the observer function. Since the only source of values of the polymorphic type is the source list, this matrix forms all the possible permutations of the inputs that the observer function can get. By applying the pair of the observer function for indices (obs_I) and the observer function for elements (obs_X) to the pairs in elements of the matrix, we get a matrix filled with Boolean values. A mapping is valid if all of the elements of this Boolean matrix are of the value *True*.

[3]isomorphism is a bijective homomorphism relation

For example, consider the source list `"ab"` and the source mapping `[(1,'a'),(2,'b')]`. With an observer function of arity one, the computed matrix would be the following:

```
[(1,'a'),(2,'b')]
```

With an observer function of arity three, the computed matrix would be the following:

```
[[(1,'a'),[(1,'a'),[(1,'a')]]]
,[(1,'a'),[(1,'a'),[(2,'b')]]]
,[(1,'a'),[(2,'b'),[(1,'a')]]]
,[(1,'a'),[(2,'b'),[(2,'b')]]]
,[(2,'b'),[(1,'a'),[(1,'a')]]]
,[(2,'b'),[(1,'a'),[(2,'b')]]]
,[(2,'b'),[(2,'b'),[(1,'a')]]]
,[(2,'b'),[(2,'b'),[(2,'b')]]]]
```

We implement a function to calculate this matrix as follows:

```
perm :: Sing_Nat (Succ m) → [(i,a)] →
  [Vect (Succ m) (i,a)]
perm (Succ_Sing Zero_Sing) ms = (:::Nil) `map` ms
perm (Succ_Sing (Succ_Sing n)) ms = join
  [((i,x):::) `map` (perm (Succ_Sing n) ms) | (i,x) ← ms]
```

As the final step, we need to apply the resulting matrix to the pair of observer functions:

```
check_GBy :: ∀t a x s.
  (Vect_Iso t,Size t∼Succ s,Eq x) ⇒
  (t Int → x) → (t a → x) → [(Int,a)] → Bool
check_GBy obs_I obs_X ms = let
  vs = perm (size (⊥ :: t Int)) ms
  in and
    [obs_I (fromVect (fmap fst z)) ==
       obs_X (fromVect (fmap snd z)) | z ← vs]
```

Here, we added the extra constraint *Size t∼Succ s* to make sure that the arity of the observer function is at least one.

3.5.3 Bidirectionalization with Arity-Generic Observations

In order to use the arity-generic function check_{GBy} in a version of bff_{GUS} with arity-generic observer functions, we need to provide an arity-generic version of the function *on* defined in the module *Data.Function*:

```
onG :: Vect_Iso t ⇒
  (t b → c) → (a → b) → (t a → c)
onG f f′ = f ∘ fromVect ∘ (fmap f′) ∘ toVect
```

Finally, a version of bff_{GUS} with an arity-generic observer function, denoted as bff^{a-*}_{GUS}, is implemented by replacing the functions *on* and check_{IBy} with their equivalent arity-generic versions, namely the functions *onG* and check_{GBy}:

```
bff^{a−*}_GUS :: ∀x t s.
  (Vect_Iso t, Eq x, Size t∼Succ s) ⇒
  (∀a.(t a → x) → [a] → [a]) →
  (∀a.Eq a ⇒
  (t a → x) → [a] → [a] → Either String [a])
bff^{a−*}_GUS get_By obs_X s v = do
    -- Step 1
  let ms = index s
  let obs_I = onG obs_X (fromJust.(flip lookup ms))
    -- Step 2
  let is  = fst ‘map‘ ms
  let iv  = get_By obs_I is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms′ = union mv ms
    -- Step 5.1
  unless (check_GBy obs_I obs_X ms′)
    $ Left "Invalid modified view!"
    -- Step 6
  return $ lookupAll is ms′
```

To be able to bidirectionalize a function with bff^{a-*}_{GUS}, we need to derive the type class Vect_{Iso}. Unfortunately, due to lack of (closed-) type functions[4] in Haskell, we cannot

[4]also, it is not possible to instantiate a type class with a partially applied type synonym

derive a constructor class (a type class with higher kind) [Jon93] for homogeneous structures. For instance, we cannot derive the type class *Functor* for homogeneous tuples; we need something like the following pseudocode, where Λ introduces the type-level abstraction:

```
instance Functor (Λ a → (a,a)) where
  fmap (x₁,x₂) = (f x₁,f x₂)
```

To bypass this restriction, we wrap each tuple in a corresponding isomorphic **newtype** declaration:

```
newtype ()¹ a = ()¹ a
newtype ()² a = ()² (a,a)
newtype ()³ a = ()³ (a,a,a)
  -- ...
```

Now, we can derive the type class Vect_{Iso} for these declarations:

```
instance Vect_Iso ()¹ where
  type Size ()¹ = Succ Zero
  toVect (()¹ x) = x ::: Nil
  fromVect (x ::: Nil) = ()¹ x
instance Vect_Iso ()² where
  type Size ()² = Succ (Succ Zero)
  toVect (()² (x₁,x₂)) =
              x₁ ::: x₂ ::: Nil
  fromVect (x₁ ::: x₂ ::: Nil) =
              ()² (x₁,x₂)
instance Vect_Iso ()³ where
  type Size ()³ = Succ (Succ (Succ Zero))
  toVect (()³ (x₁,x₂,x_3)) =
              x₁ ::: x₂ ::: x_3 ::: Nil
  fromVect (x₁ ::: x₂ ::: x_3 ::: Nil) =
              ()³ (x₁,x₂,x_3)
```

Using the homogeneous tuples, we can encode the original function bff_{GUS} using the function bff^{a-*}_{GUS} as follows:

```
uncurry' :: ∀a b.(a → a → b) → (()² a → b)
uncurry' f (()² (x₁,x₂)) = f x₁ x₂


curry' :: ∀a b.(()² a → b) → (a → a → b)
curry' f x₁ x₂ = f (()² (x₁,x₂))
```

$$
\begin{array}{l}
\acute{bff}_{GUS} \; :: \forall x.(Eq\ x) \Rightarrow \\
\quad (\forall a.(a \to a \to x) \to [\,a\,] \to [\,a\,]) \to \\
\quad (\forall a.Eq\ a \Rightarrow \\
\qquad (a \to a \to x) \to [\,a\,] \to [\,a\,] \to Either\ String\ [\,a\,]) \\
\acute{bff}_{GUS}\ \ get_{By}\ \ obs_X = \textbf{let} \\
\quad \acute{get}_{By}\ \ :: \forall a.(()^2\ a \to x) \to [\,a\,] \to [\,a\,] \\
\quad \acute{get}_{By}\ \ f = get_{By}\ \ (curry'\ f) \\
\quad \acute{obs}_X\ \ = uncurry'\ obs_X \\
\quad \textbf{in}\ bff^{a-*}_{GUS}\ \ \acute{get}_{By}\ \ \acute{obs}_X
\end{array}
$$

In the same way, it is possible to define a version to bidirectionalize a forward function with an observer function of arity one:

$$
\begin{array}{l}
uncurry^1 :: \forall a\ b.(a \to b) \to (()^1\ a \to b) \\
uncurry^1\ f\ (()^1\ x) = f\ x
\end{array}
$$

$$
\begin{array}{l}
curry^1 :: \forall a\ b.(()^1\ a \to b) \to (a \to b) \\
curry^1\ f\ x = f\ (()^1\ x)
\end{array}
$$

$$
\begin{array}{l}
bff^{a-1}_{GUS}\ \ :: \forall x.(Eq\ x) \Rightarrow \\
\quad (\forall a.(a \to x) \to [\,a\,] \to [\,a\,]) \to \\
\quad (\forall a.Eq\ a \Rightarrow \\
\qquad (a \to x) \to [\,a\,] \to [\,a\,] \to Either\ String\ [\,a\,]) \\
bff^{a-1}_{GUS}\ \ get_{By}\ \ obs_X = \textbf{let} \\
\quad \acute{get}_{By}\ \ :: \forall a.(()^1\ a \to x) \to [\,a\,] \to [\,a\,] \\
\quad \acute{get}_{By}\ \ f = get_{By}\ \ (curry^1\ f) \\
\quad \acute{obs}_X\ \ = uncurry^1\ obs_X \\
\quad \textbf{in}\ bff^{a-*}_{GUS}\ \ \acute{get}_{By}\ \ \acute{obs}_X
\end{array}
$$

3.6 Generics

So far, for simplicity we used lists as the polymorphic data structures whose elements change in the view and are put back by the backward function (*put*). The original paper [Voi09] introduces a simple (data-) generic programming technique to apply the bidirectionalization algorithm to put back changes to elements of any ADT (modulo deriving specific type classes). In this section, for simplicity, we rewrite the original generic algorithm [Voi09] in our own preferred style. We only study the generic algorithm for *bff* and bff^{a-*}_{GUS} , as the same idea can simply be used for the other versions.

Having the type of a data structure deriving the *Traversable* type class, we will be able to extract all the (polymorphic) elements of the data structure as a list by the function *toList*. For example, in the following we define an ADT representing a polymorphic binary tree structure and we use the GHC extensions to automatically derive the *Traversable* type class for us (and the other required super-classes, namely *Functor* and *Foldable*):

```
{-# LANGUAGE DeriveTraversable #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveFunctor #-}


data Tree a = Node a (Tree a) (Tree a)
  | Leaf a
  deriving (Functor,Foldable,Traversable,Eq)
```

Consider the following example where applying the function *toList* extracts the elements of the structure and returns them as a list:

```
ghci > toList (Node 'a' (Leaf 'b') (Leaf 'c'))
"abc"
```

Now, we define a generic function which replaces the polymorphic elements (elements of the data structure that are of the polymorphic type) of a data structure with the corresponding elements of the input list. Since the traversal scheme is fixed, the correspondence is set by the order in which nodes are visited. The generic function first assigns a running series of integers starting at zero to the polymorphic elements of the structure and then uses the numbers as indices to extract the corresponding element from the input list:

```
fromList :: ∀k a b.Traversable k ⇒ k a → [b] → k b
fromList s lst = let
  indices _ = do i ← Control.Monad.State.get
    Control.Monad.State.put (i + 1)
    return i
  si = Control.Monad.State.evalState
      (Data.Traversable.mapM indices s) 0
  in fmap (lst!!) si
```

Consider the following example where applying the function *fromList* replaces the elements of the structure with the corresponding elements from the input list:

```
ghci > fromList (Node 'a' (Leaf 'b') (Leaf 'c')) [1,2,3]
Node 1 (Leaf 2) (Leaf 3)
```

We also define a generic function to compare (the shape of) two data structures regardless of the value of their polymorphic elements. For that purpose, we set every element of the structure to the same value and then check if they are equal:

```
(==_Shape) :: ∀k a.(Eq (k a),Foldable k,Functor k) ⇒ k a → k a → Bool
(==_Shape) x y = case (toList x) of
  [] → x == y
  (a : _) → ((==) ‘on‘ fmap (const a)) x y
```

In the following example, we use the function $(==_{Shape})$ to compare two data structures regardless of the value of their polymorphic elements:

```
ghci > (==_Shape) (Node 'a' (Leaf 'b') (Leaf 'c'))
  (Node 'd' (Leaf 'e') (Leaf 'f'))
True
```

Now, we use these generic functions to define the generic version of *bff*. We need to extract the polymorphic data from the data structures, let the original version *bff* take care of the bidirectionalization and then put the updated source elements from the list into the original data structure. There are also two subtle points to notice. First, the forward function *get* works on specific polymorphic data structures that can be anything other than lists. Therefore, we need to pass the shape of the original structure as an additional parameter (can be viewed as context) to the forward function (hence *s* in the expression *get* (*fromList* *s* *x*) is fixed). Also, since we consider any polymorphic data structure, the equality of length no longer guarantees equality of the shape of the data structures (as it did for lists), therefore we have to add an additional check to reject the modified view with shape changes.

```
bff^{d-*} :: ∀k k'.
  (Functor k',Foldable k',Traversable k) ⇒
  (∀a.k a → k' a) →
  (∀a.(Eq a,Eq (k' a)) ⇒ k a → k' a → Either String (k a))
bff^{d-*} get s v = do
  let s^{list} = toList s
  let v^{list} = toList v
  let get^{list} :: ∀a.[a] → [a]
      get^{list} x = toList $ get (fromList s x)
  unless ((==_Shape) (get s) v)
    $ Left "Modified view off wrong shape!"
  ś^{list} ← bff get^{list} s^{list} v^{list}
  return $ fromList s ś^{list}
```

Likewise, we extend our generalize function bff^{a-*}_{GUS} using generic programming techniques:

```
$bff^{a/d-*}_{GUS}$ :: ∀x k k′ t s.
  (Vect$_{Iso}$ t,Functor k′,Foldable k′
  ,Size t∼Succ s,Traversable k,Eq x) ⇒
  (∀a.(t a → x) → k a → k′ a) →
  (∀a.(Eq a,Eq (k′ a)) ⇒
    (t a → x) → k a → k′ a →
    Either String (k a))
$bff^{a/d-*}_{GUS}$ get$_{By}$ obs$_X$ s v = do
  let s$^{list}$ = toList s
  let v$^{list}$ = toList v
  let get$^{list}_{By}$ :: ∀a.(t a → x) → [a] → [a]
    get$^{list}_{By}$ obs x = toList $ get$_{By}$ obs (fromList s x)
  unless ((==$_{Shape}$) (get$_{By}$ obs$_X$ s) v)
    $ Left "Modified view off wrong shape!"
  ś$^{list}$ ← $bff^{a-*}_{GUS}$ get$^{list}_{By}$ obs$_X$ s$^{list}$ v$^{list}$
  return $ fromList s ś$^{list}$
```

In the same way, we can extend the function bff^{a-1}_{GUS} using generic programming techniques:

```
$bff^{a-1/d-*}_{GUS}$ :: ∀k k′ x.
  (Functor k′,Foldable k′,Traversable k,Eq x) ⇒
  (∀a.(a → x) → k a → k′ a) →
  (∀a.(Eq a,Eq (k′ a)) ⇒
    (a → x) → k a → k′ a → Either String (k a))
$bff^{a-1/d-*}_{GUS}$ get$_{By}$ obs$_X$ s v = do
  let s$^{list}$ = toList s
  let v$^{list}$ = toList v
  let get$^{list}$ :: ∀a.(a → x) → [a] → [a]
    get$^{list}$ obs x = toList $ get$_{By}$ obs (fromList s x)
  unless ((==$_{Shape}$) (get$_{By}$ obs$_X$ s) v)
    $ Left "Modified view off wrong shape!"
  ś$^{list}$ ← $bff^{a-1}_{GUS}$ get$^{list}$ obs$_X$ s$^{list}$ v$^{list}$
  return $ fromList s ś$^{list}$
```

To enable bidirectionalization of functions like *partition* :: ∀a.(a → *Bool*) → [a] → ([a],[a]), we need to modify the function $bff^{a-1/d-*}_{GUS}$ slightly. That is because the type ([a],[a]) cannot derive the type classes *Functor* and *Foldable* directly.

```
data PairList a = PairList ([a],[a])
  deriving (Functor,Foldable,Eq)
```

```
$bff_{Par}$ :: $\forall k\ x.$
  ($Traversable\ k$,$Eq\ x$) $\Rightarrow$
  ($\forall a.(a \to x) \to k\ a \to ([a],[a])) \to$
  ($\forall a.(Eq\ a) \Rightarrow$
    $(a \to x) \to k\ a \to ([a],[a]) \to Either\ String\ (k\ a))$
$bff_{Par}\ get_{By}\ \ obs_X\ s\ v$ = let
    $get^{PL}_{By}$ :: $\forall a.(a \to x) \to k\ a \to PairList\ a$
    $get^{PL}_{By}\ obs\ x = PairList\ \$\ get_{By}\ \ obs\ x$
    $v\_PL = PairList\ v$
  in $bff^{a-1/d-*}_{GUS}\ \ get^{PL}_{By}\ obs_X\ s\ v\_PL$
```

To see other examples of how these functions are used in practice, refer to the next chapter where we use these generic functions to bidirectionalize code generating functions.

In general, we can bidirectionalize any function with a type signature isomorphic to the type of the function argument in the function $bff^{a/d-*}_{GUS}$. For example, an uncurried forward function can be bidirectionalized by the following:

```
$uncurBff^{a/d-*}_{GUS}$ :: $\forall x\ k\ k'\ t\ s.$
  ($Vect_{Iso}\ \ t$,$Functor\ k'$,$Foldable\ k'$
  ,$Size\ t{\sim}Succ\ s$,$Traversable\ k$,$Eq\ x$) $\Rightarrow$
  ($\forall a.((t\ a \to x),k\ a) \to k'\ a) \to$
  ($\forall a.(Eq\ a,Eq\ (k'\ a)) \Rightarrow$
    $(t\ a \to x) \to k\ a \to k'\ a \to$
    $Either\ String\ (k\ a))$
$uncurBff^{a/d-*}_{GUS}\ \ uncurGet_{By}$ = let
  $get_{By}$ :: $\forall a.(t\ a \to x) \to k\ a \to k'\ a$
  $get_{By}\ = curry\ uncurGet_{By}$
  in $bff^{a/d-*}_{GUS}\ get_{By}$
```

3.7 Demonstration

In order to measure how well our algorithm performs in practice, we consider the total number of functions in the *Prelude* module of *Haskell 2010* that our algorithm can bidirectionalize. The figure 3.1 displays the distribution of all the functions in the *Prelude* module based on their types; the methods of the type classes are excluded. The data for the following graphs are attached as an appendix (Appendix C).

Our algorithm can bidirectionalize 40% of the polymorphic functions in the *Prelude* module (figure 3.2). It is 20% improvement in the total number of the functions bidirectionalizable by our algorithm compared to the original algorithm [Voi09]. In total, we can bidirectionalize 30% of all the functions defined in the *Prelude* module.

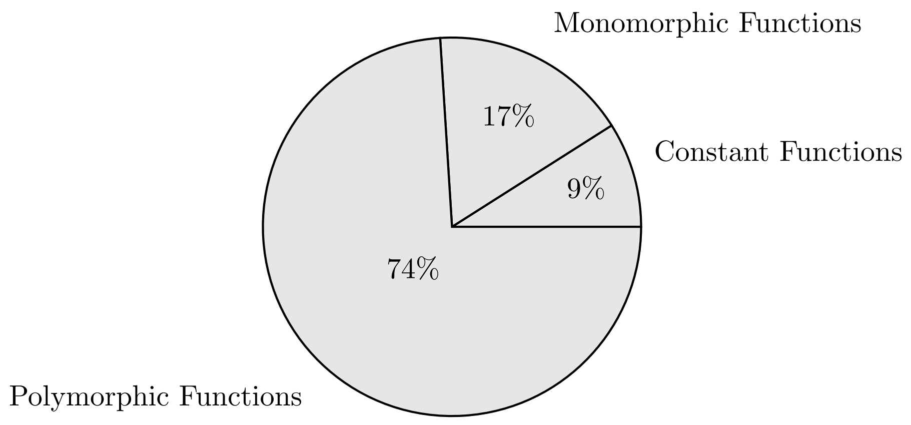


Figure 3.1: Prelude Functions

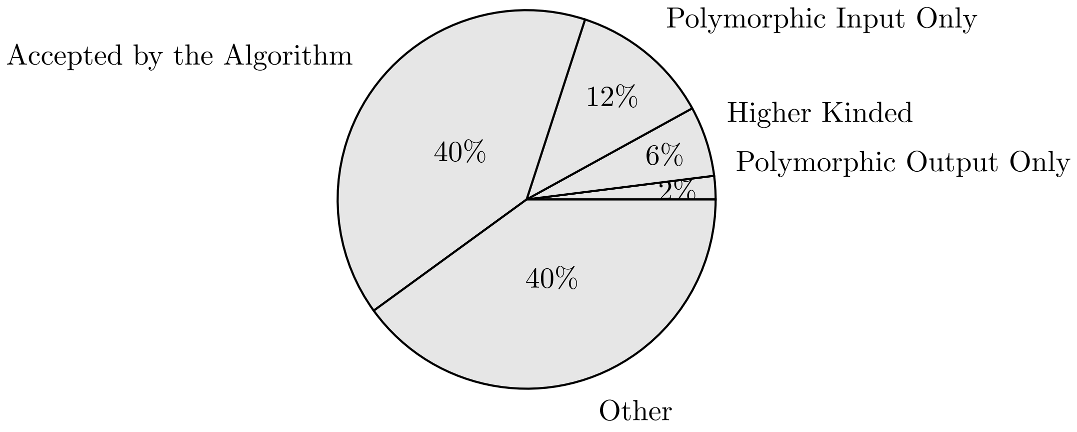


Figure 3.2: Polymorphic Functions in Prelude

Our algorithm can bidirectionalize polymorphic forward functions with 20% of the type classes declared in the *Prelude* module.

3.8 Future Work

One potential improvement to the system is to extend the algorithm to bidirectionalize the higher-order functions with generator functions, e.g. a function argument of the type $a \to a \to a$. In fact, we already have sketched an algorithm that does this for us. The key idea is to keep track of the origin of the generated elements. For example, having the generator function $(+\!\!+)$, we force the underlying algorithm to use the generator function $\lambda(i,x)\ (j,y) \to (i:+:j, x +\!\!+ y)$ where the constructor $:+:$ allows us to store the indices of the elements forming generated elements.

The other potential extension is to combine our work with the syntactic-bidirectionalization approach (section 2.2.3.1) in the same way as the original semantic-bidirectionalization algorithm was combined with syntactic-bidirectionalization (section 2.2.3.3).

Theoretically, it is interesting to find the limitations of semantic-bidirectionalization formally. Also, how this approach relates to the other existing BX techniques is a question worth investigating.

Chapter 4

Tracking Generated Expressions

4.1 Tracking Generated Expressions

In order to track generated expressions, we employ the semantic-bidirectionalization technique. The idea of applying bidirectionalization techniques to put back the results of analyses on the generated expressions was originally proposed in *Wang*'s Ph.D. thesis (section 4.4 of [Wan10]). However, he applied syntactic bidirectionalization [MHN+07]. As described before (section 2.2.3.1), syntactic bidirectionalization limits the programmer to program in a syntactically restricted language. Hence, in this thesis, we propose applying semantic bidirectionalization technique to lift these restrictions.

The problem of tracking generated expressions can be modeled as follows:

The transformation function of type $Exp_{\uparrow} \rightarrow Exp_{\downarrow}$ transforms the values of type $Exp_{\uparrow}$ representing the high-level abstract syntax tree (AST) to the values of type $Exp_{\downarrow}$ representing the low-level AST. Having a high-level expression $exp_{\uparrow} :: Exp_{\uparrow}$ and the corresponding low-level expression $exp_{\downarrow} :: Exp_{\downarrow}$, we want for every subexpression $sub_{\downarrow}$ of $exp_{\downarrow}$ to be able to find the subexpressions of $exp_{\uparrow}$ that $sub_{\downarrow}$ is originally derived from.

Our solution can be sketched as the following steps:

1. the expressions in the high-level are initially annotated with a *Boolean* flag of the value *False*
2. the annotations are preserved throughout the transformations, from the high-level to the low-level
3. whenever we want to find the origin of an expression, we change its annotation flag to the value *True*
4. we use semantic-bidirectionalization to automatically track and update the annotation flags of the corresponding expressions in the high-level code

For example, consider two languages: untyped lambda calculus as the low-level language and untyped lambda calculus extended with **let** expressions (a syntactic sugar) as the high-level language. The abstract syntax of the two languages (with annotations) are presented as the following algebraic data type declarations:

```
data Exp↑ ann =
  Var↑ String
  | Abs↑ String (Exp↑ ann)
  | App↑ (Exp↑ ann) (Exp↑ ann)
  | Ann↑ ann (Exp↑ ann)
  | Let String (Exp↑ ann) (Exp↑ ann)
  deriving (Functor,Foldable,Traversable,Eq)
```

$$\begin{array}{l}
\textbf{data}\ \mathit{Exp}_{\downarrow}\ \mathit{ann} = \\
\quad \mathit{Var}_{\downarrow}\ \mathit{String} \\
\quad \mid \mathit{Abs}_{\downarrow}\ \mathit{String}\ (\mathit{Exp}_{\downarrow}\ \mathit{ann}) \\
\quad \mid \mathit{App}_{\downarrow}\ (\mathit{Exp}_{\downarrow}\ \mathit{ann})\ (\mathit{Exp}_{\downarrow}\ \mathit{ann}) \\
\quad \mid \mathit{Ann}_{\downarrow}\ \mathit{ann}\ (\mathit{Exp}_{\downarrow}\ \mathit{ann}) \\
\quad \textbf{deriving}\ (\mathit{Functor}, \mathit{Foldable}, \mathit{Traversable}, \mathit{Eq})
\end{array}$$

The transformation function (*desugar*) simply desugars the **let** expressions:

$$\begin{array}{l}
\mathit{desugar} :: \forall \mathit{ann}.\mathit{Exp}_{\uparrow}\ \mathit{ann} \to \mathit{Exp}_{\downarrow}\ \mathit{ann} \\
\mathit{desugar}\ (\mathit{Var}_{\uparrow}\ x) \quad = \mathit{Var}_{\downarrow}\ x \\
\mathit{desugar}\ (\mathit{Abs}_{\uparrow}\ x\ \ e) = \mathit{Abs}_{\downarrow}\ x\ (\mathit{desugar}\ e) \\
\mathit{desugar}\ (\mathit{App}_{\uparrow}\ e_1\ e_2) = \mathit{App}_{\downarrow}\ (\mathit{desugar}\ e_1)\ (\mathit{desugar}\ e_2) \\
\mathit{desugar}\ (\mathit{Ann}_{\uparrow}\ a\ e) = \mathit{Ann}_{\downarrow}\ a\ (\mathit{desugar}\ e) \\
\mathit{desugar}\ (\mathit{Let}\ x\ e_1\ e_2) = \mathit{App}_{\downarrow}\ (\mathit{Abs}_{\downarrow}\ x\ (\mathit{desugar}\ e_2))\ (\mathit{desugar}\ e_1)
\end{array}$$

In the following, in order to improve clarity of the presentation, we use quasiquotations [Mai07]. The text wrapped in the Oxford brackets $[\,\mathit{qType} \mid \mathit{term} \mid]$ is interpreted as a term of the type *Type*. Annotations are presented as superscripts, where the annotation *False* is presented as an empty circle $\circ$ and *True* as a filled circle $\bullet$.

Consider the following high-level expression:

$$\begin{array}{l}
\mathit{exp}_{\uparrow} :: \mathit{Exp}_{\uparrow}\ \mathit{Bool} \\
\mathit{exp}_{\uparrow} = [\,\mathit{qExp}_{\uparrow} \mid \lambda x \to \textbf{let}\ \mathit{id} = (\lambda y \to y)\ \textbf{in}\ \mathit{id}\ x \mid]
\end{array}$$

It represents a function that takes an input and then applies the identity function, defined in the local **let** binding, to the input. Applying the transformation function *desugar* to the high-level expression $\mathit{exp}_{\uparrow}$ results in the following low-level expression:

$$\begin{array}{l}
\mathit{exp}_{\downarrow} :: \mathit{Exp}_{\downarrow}\ \mathit{Bool} \\
\mathit{exp}_{\downarrow} = [\,\mathit{qExp}_{\downarrow} \mid \lambda x \to (\lambda \mathit{id} \to \mathit{id}\ x)\ (\lambda y \to y) \mid]
\end{array}$$

For tracking back the low-level subexpression $[\,\mathit{qExp}_{\downarrow} \mid \lambda y \to y \mid]$, our algorithm works as the following steps:

1. every subexpression in the high-level expression $\mathit{exp}_{\uparrow}$ is annotated with the Boolean value $False$:

$$\begin{array}{l}
\mathit{exp}^{\mathit{Ann}}_{\uparrow} :: \mathit{Exp}_{\uparrow}\ \mathit{Bool} \\
\mathit{exp}^{\mathit{Ann}}_{\uparrow} = [\,\mathit{qExp}_{\uparrow} \mid \\
\quad \lambda^{\circ}\ x \to \\
\quad\quad \textbf{let}^{\circ} \\
\quad\quad\quad \mathit{id} = \lambda^{\circ}\ y \to y^{\circ} \\
\quad\quad \textbf{in}\ (\mathit{id}^{\circ}\ x^{\circ})^{\circ} \mid]
\end{array}$$

2. the annotations are preserved throughout the transformations, from the high-level to the low-level and result in the following low-level expression:

```
$exp_{\downarrow}^{Ann} :: Exp_{\downarrow}\ Bool$
$exp_{\downarrow}^{Ann} = [\,qExp_{\downarrow}\ |$
  $\lambda^{\circ}\ x \rightarrow$
    $((\lambda id \rightarrow$
      $(id^{\circ}\ x^{\circ})^{\circ})$
      $(\lambda^{\circ}\ y \rightarrow y^{\circ}))^{\circ}\ |]$
```

3. we set the annotation of the subexpression that we want to track (the whole subexpression on the last line) to $True$:

```
$exp_{\downarrow}^{Ann} :: Exp_{\downarrow}\ Bool$
$exp_{\downarrow}^{Ann} = [\,qExp_{\downarrow}\ |$
  $\lambda^{\circ}\ x \rightarrow$
    $((\lambda id \rightarrow$
      $(id^{\circ}\ x^{\circ})^{\circ})$
      $(\lambda^{\bullet}\ y \rightarrow y^{\circ}))^{\circ}\ |]$
```

4. we use the generic function bff_{Gen} (section 3.6) to automatically update the corresponding annotations in the high-level expression:

```
$ghci > bff_{Gen}\ desugar\ exp_{\uparrow}^{Ann}\ exp_{\downarrow}^{Ann}$
$Right\ [\,qExp_{\uparrow}\ |$
  $\lambda^{\circ}\ x \rightarrow$
    $\mathbf{let}^{\circ}$
      $id = \lambda^{\bullet}\ y \rightarrow y^{\circ}$
    $\mathbf{in}\ (id^{\circ}\ x^{\circ})^{\circ}\ |]$
```

The Boolean flag of the corresponding subexpression (the third line) is updated.

Semantic-bidirectionalization requires the transformation function to be polymorphic on the type of the annotations. It is a natural demand, as we do not expect the transformation function to be able to generate or observe the annotations; the content of the annotations should be kept abstract and the transformation function should be agnostic towards the annotations. The original proposal [Wan10] requires the transformation function and the data types to be polymorphic on the type of the annotations from the beginning; it is well-suited for fresh developments. In the following sections, we explore the design space and propose solutions to bypass this restriction.

Another difficulty in tracking generated expressions in *Feldspar* is to locate the exact source location of the high-level expressions. Since *Feldspar* is an embedded domain specific language, there is no corresponding parse tree for an abstract syntax tree and hence there is no explicit connection between the expressions and the actual code in a source file. Later in this chapter, we describe the framework that we have designed to address this problem.

4.2 Annotations in Data

We enumerate three distinct ways to store annotations in an ADT declaration, based on where in an ADT [1] annotations are stored:

1. *Product-Annotations*: each node in the ADT contains an annotation, e.g.:

   ```
   data Exp ann = Var ann String
      | Abs ann String (Exp ann)
      | App ann (Exp ann) (Exp ann)
      | Let ann String  (Exp ann) (Exp ann)
   ```

2. *Sum-Annotations*: annotations are carried in a separate wrapper node, e.g.:

   ```
   data Exp ann = Var String
      | Abs String (Exp ann)
      | App (Exp ann) (Exp ann)
      | Let String  (Exp ann) (Exp ann)
      | Ann ann    (Exp ann)
   ```

3. *Recursion-Annotations*: each recursion is annotated, e.g.:

   ```
   type Exp ann = (ann,Exp')
   data Exp' ann = Var String
      | Abs String (ann,Exp')
      | App (ann,Exp') (ann,Exp')
      | Let String (ann,Exp') (ann,Exp')
   ```

The original proposal [Wan10] uses mutually recursive definitions which is equivalent to recursion-annotations:

[1]An ADT can be viewed as recursive sum of products

type $Exp\ ann = (ann, Exp')$
data $Exp'\ ann = Var\ String$
  $\mid Abs\ String\ (Exp\ ann)$
  $\mid App\ (Exp\ ann)\ (Exp\ ann)$
  $\mid Let\ String\ \ (Exp\ ann)\ (Exp\ ann)$

Using product-annotations and recursion-annotations, the annotations are scattered all over the ADT, while, in sum-annotations, annotations are all carried in separate nodes; the definition of the other language constructs are not polluted by the annotations. Also, product-annotations and recursion-annotations, unlike sum-annotations, come with the guarantee that every expression in the abstract syntax is annotated; using product-annotations and recursion-annotations, it is impossible to construct an expression without providing an annotation for it.

4.3 Preserving the Annotations

Regardless of the way annotations are stored in the data types, we expect the transformation function to preserve the annotations. The original proposal [Wan10] does not specify the preservation formally. Therefore, we propose the following simple condition:

Condition 6 (Annotation Preservation)

Whenever a value of an annotated type (e.g. Exp ann) is deconstructed by pattern-matching, the annotations should be transferred (injected) to the value of the selected expression (e.g. the body expression of a case or function alternative).

□

Consider the function *desugar* from the previous section where sum-annotations are used. Due to the following function alternative, the function *desugar* satisfies the condition:

$desugar\ (Ann_{\uparrow}\ \boxed{a}\ e) = Ann_{\downarrow}\ \boxed{a}\ (desugar\ e)$

4.3.1 Towards Annotation Preservation

The original proposal [Wan10] expects all the functions and the data types related to the transformation to be able to preserve and carry the annotations. We propose a simple algorithm based on sum-annotations to transform an incapable system to be able to preserve and carry the annotations.

First, we define a type class to model data types that can store annotations:

```
{-# LANGUAGE TypeFamilies #-}

class Inj t where
  type Ann t
  inj :: Ann t → t → t

class Inj t ⇒ Annotatable t where
  prj :: t → Maybe ((Ann t,t))
```

Members of this type class should respect the following condition:

Condition 7 (Valid Annotatable Type)

For the type T to be a valid instance of the Annotatable type class the following property should hold:

$$\forall t :: T. \forall ann :: Ann\ T.\ proj\ (inj\ ann\ t) = Just\ ann$$

For example, consider the two data types $Exp_{\uparrow}$ and $Exp_{\downarrow}$ from before. They can be valid members of the *Annotatable* type class:

```
instance Inj (Exp↑ ann) where
  type Ann (Exp↑ ann) = ann
  inj = Ann↑

instance Inj (Exp↓ ann) where
  type Ann (Exp↓ ann) = ann
  inj = Ann↓

instance Annotatable (Exp↑ ann) where
  prj (Ann↑ x e) = Just (x,e)
  prj _ = Nothing

instance Annotatable (Exp↓ ann) where
  prj (Ann↓ x e) = Just (x,e)
  prj _ = Nothing
```

We also define the function *preserve* to preserve annotation through a given transformation:

```
preserve :: (Annotatable t↑,Inj t↓,Ann t↑~Ann t↓) ⇒
  t↑ → (t↑ → t↓) → t↓
preserve e↑ f = case prj e↑ of
  Just (ann,é↑) → inj ann (f é↑)
  Nothing → f e↑
```

The first input of the function *preserve* is the scrutiny and the second input is the body of the case expression abstracted over its scrutiny.

To transform an incapable system to be able to preserve and carry the annotations, we sketch our algorithm as the following steps:

1. for each related data type, following the sum-annotation style, we add a new separate data constructor to carry the annotations
2. for each data type, we derive the type class *Annotatable*
3. to get rid of nested patterns, we apply the standard transformations defined in Haskell's language report to flatten nested patterns (a necessary step for most Haskell compilers) and transform all the other possible syntactic forms of pattern matchings to case expressions
4. we apply the function *preserve* anywhere a value of the annotated type is deconstructed

These transformations are simple enough that they can be done automatically by a preprocessor.

For example, consider the following optimization function that transforms application of an identity function to any expression e to the expression e itself.

```
data Exp′ =
  Var′ String
  | Abs′ String Exp′
  | App′ Exp′  Exp′
opt :: Exp′ → Exp′
opt (App′ (Abs′ x1 (Var′ x2)) e) | x1 == x2 = e
opt e = e
```

Our algorithm works as the following steps:

Step 1: Making data annotatable

```
data Exp' ann =
  Var' String
  | Abs' String (Exp' ann)
  | App' (Exp' ann) (Exp' ann)
  | Ann ann (Exp' ann)
  deriving (Functor,Foldable,Traversable,Eq)
```

Step 2: Deriving *Annotatable* type class

```
instance Inj (Exp' ann) where
  type Ann (Exp' ann) = ann
  inj = Ann

instance Annotatable (Exp' ann) where
  prj (Ann x e) = Just (x,e)
  prj _ = Nothing
```

Step 3: Flattening the nested patterns

```
opt' :: Exp' → Exp'
opt' e0 = case e0 of
  App' e1 e    → case e1 of
    Abs' x1 e2 → case e2 of
      Var' x2 → if (x1 == x2)
                then e
                else e0
      _         → e0
    _           → e0
  _             → e0
```

Step 4: Applying *preserve*

```
$opt' :: Exp'\ ann \to Exp'\ ann$
$opt'\ e_0 = preserve\ e_0\ \$\ \lambda\_x_0 \to$ **case** $\_x_0$ **of**
  $App'\ e_1\ e \quad \to preserve\ e_1\ \$\ \lambda\_x_1 \to$ **case** $\_x_1$ **of**
    $Abs'\ x_1\ e_2 \to preserve\ e_2\ \$\ \lambda\_x_2 \to$ **case** $\_x_2$ **of**
      $Var'\ x_2 \to$ **if** $(x_1 == x_2)$
                **then** $e$
                **else** $e_0$
      $\_ \qquad \to e_0$
    $\_ \qquad \to e_0$
  $\_ \qquad \to e_0$
```

4.4 Injecting Annotations

In standalone (non-embedded) languages, while parsing, we can gather source location information corresponding to each node inside the AST. In embedded languages, there is no parsing phase and therefore, the information about the exact location of expressions in the original source code is lost. To avoid this problem and to recover the lost information, we add a preprocessing phase to inject this information inside the data object representing the embedded program. Proprecessing embedded DSL seems unnecessary, since it violates the very reason for embedding in the first place; one of the main reasons behind embedding is to avoid writing parsers, pretty printers and the like but using a preprocessor forces us to do so. Anyhow, since we are dealing with the standard Haskell code itself, there are standard parsers that can be borrowed. Arguably, preprocessing an object language embedded in a host language seems reasonable if the standard parser of the host language itself is borrowed. It does not add any unnecessary burden.

For this purpose, we developed a tool named QuickAnnotate [2]. QuickAnnotate can be used by adding the pragma $\{-\#OPTIONS\_GHC - F - pgmF\ qapp\#-\}$ at the top of the source file in GHC Haskell and it automatically injects the source locations to expressions in the top-level bindings. The injection is done by applying the overloaded function $injLoc :: \forall a.Locatable\ a \Rightarrow Loc \to a \to a$ to the corresponding source location and the top-level expressions. For example, having the following code:

```
1 :
2 : **module** *Test* **where**
3 : *exp1* = "test"
4 : *exp2* = *Just* 1
```

[2]http://hackage.haskell.org/package/QuickAnnotate

After the preprocessing, we get the following code where the body expression of every top-level biding is annotated with the source location:

```
1 :
2 : module Test where
3 : exp1 = injLoc (SrcLoc {srcFilename = "~/Test.hs"
    ,srcLine = 03,srcColumn = 1}) $ "test"
4 : exp2 = injLoc (SrcLoc {srcFilename = "~/Test.hs"
    ,srcLine = 04,srcColumn = 1}) $ Just 1
```

The *injLoc* function is overloaded and uses the following GHC extensions to provide default instances:

```
{-#LANGUAGE FlexibleInstances#-}
{-#LANGUAGE IncoherentInstances#-}
{-#LANGUAGE OverlappingInstances#-}
```

It is defined by the following type class:

```
class Locatable a where
  injLoc :: Loc → a → a
```

Then we set the overloaded function *injLoc* to act as the identity function for the default cases:

```
instance Locatable a where
  injLoc _ = id
```

In case, *injLoc* is applied to a function, we wrap the function in a way that only the output of the function is annotated.

```
instance Locatable b ⇒ Locatable (a → b) where
  injLoc l f = λx → injLoc l (f x)
```

Finally, to make it actually inject annotations into values of a specific type, we need to provide an instance of *Locatable* type class for that specific type. For example in the above, if we would like to inject source locations into top-level expressions of type *String*, it can be achieved as follows:

```
instance Locatable ([Char]) where
  injLoc loc d = d ++ " at " ++ (show loc)
```

Using this simple type-level programming, the programmer can customize the way Quick-Annotate annotates each type. For example, consider the data type *Exp′* from the previous section. In order to inject the source locations into the top-level expression of type *Exp′*, derive the type class *Locatable* as follows:

```
instance Locatable (Exp′ Loc) where
  injLoc loc e = inj loc e
```

4.5 Demonstration

4.5.1 Tracking Generated Expressions in Pico-Feldspar

In order to demonstrate how our solution enables tracking the generated expressions in EDSLs, we have developed the EDSL *Pico-Feldspar* in Haskell from scratch. *Pico-Feldspar* is translated to *C*. We have developed *Pico-Feldspar* initially without annotations and then refactored the code using our algorithm to enhance the system with the ability to carry and preserve annotations. Then, we applied semantic bidirectionalization to track the generated expressions. The code is attached (Appendix B) with the exact refactorings highlighted in gray.

Pico-Feldspar, as the name suggests, is a tiny subset of *Feldspar* that, unlike *Feldspar*, uses normal GADTs to define the data types, including the abstract syntax tree. *Feldspar* uses the library *Syntactic* [Axe12] to define extensible data types [Swi08]. Having extensible data types, the task of introducing and preserving annotations is trivial [BH11]. Not all EDSLs in Haskell are implemented via extensible data types. Therefore, to have a realistic demonstration of our algorithm, it was critically important to experiment with an EDSL without extensible data types.

4.5.2 Tracking Generated Expressions in Feldspar

We also applied our technique to enhance *Feldspar* with the ability to track the expressions in the low-level generated *C* code all the way back to their origins at the high-level *Haskell* code. Starting from the version 0.5.0.1, our system has been part of the *Feldspar*'s official released version. In *Feldspar*, the emphasis in the implementation of the tracking system has been on simplicity and usability. It is often enough to push down the source-locations of the top-level bindings from the high-level *Haskell* code to the low-level *C* code. Also, it is not often necessary to annotate every single subexpression in the *C* code; often one annotation per block of code suffices.

One of the main application domains of *Feldspar* is digital signal processing. Therefore, to demonstrate some of the main features of *Feldspar* in interaction with our tracking system, we implemented a simple image processing algorithm in *Feldspar*. The algorithm first converts the input colored image to a grayscale image and then converts the grayscale image to a black-and-white (binary) image.

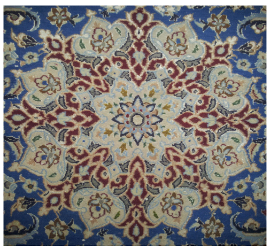


Figure 4.1: Colored

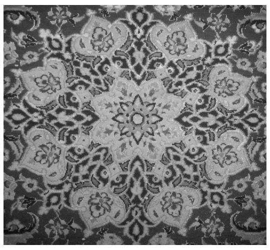


Figure 4.2: Grayscale

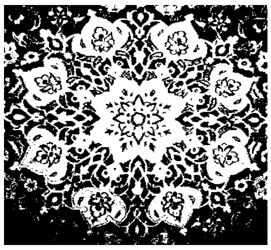


Figure 4.3: B&W

For example, the figure 4.2 and the figure 4.3 illustrate the grayscale and black-and-white versions of the photo in the figure 4.1, correspondingly.

The code for the algorithm is as follows:

```
{-#OPTIONS_GHC -F -pgmF qapp#-}
module IP where
import qualified Prelude as P
import Feldspar
import Feldspar.Vector
  -- Conversion from grayscale to black and white
toBW :: Vector (Data Int32) → Vector (Data Int32)
toBW = map (λx → condition (x < 127) 1 0)
    -- threshold is set to 127
  -- The standard red channel grayscale coefficient
redCoefficient :: Data Float
redCoefficient = 0.30
  -- The standard green channel grayscale coefficient
greenCoefficient :: Data Float
greenCoefficient = 0.59
  -- The standard blue channel grayscale coefficient
blueCoefficient :: Data Float
blueCoefficient =  0.11
  -- Conversion from RGB to grayscale
rgbToGray :: Data Int32 → Data Int32 →
  Data Int32 → Data Int32
rgbToGray r g b = truncate $
  (i2f r) * redCoefficient
   + (i2f g) * greenCoefficient
   + (i2f b) * blueCoefficient
  -- Conversion from colored to grayscale
toGray :: Vector (Data Int32) → Vector (Data Int32)
toGray v = forLoop ((length v) `div` 3) Empty
  (λi acc → let
    b = i * 3
    in acc ++ indexed 1
      (const $ rgbToGray (v ! b) (v ! (b + 1)) (v ! (b + 2))))
  -- Conversion from colored to black and white
fromColoredtoBW :: Vector (Data Int32) →
  Vector (Data Int32)
fromColoredtoBW = toBW.toGray
```

A colored image is modelled as a vector of integers in which every three consecutive number represent the color of a single pixel in the RGB (Red-Green-Blue) format. We use *Netpbm* format to store images [3]. To transform the input image to the grayscale format, the function *toGray* uses the function *rgbToGray* to compute the weighted sum of each pixel's color channels. The weights are the standard constant grayscale coefficients, representing human perception of colors. In order to transform a grayscale image into the black-and-white format, we do *thresholding* by a fixed threshold. Our threshold is fixed and suitable for photos with rather low lightness. This way, if the grayscale value of a pixel is less than the threshold, its color is set to black and otherwise to white. Notice that our preprocessor is enabled by the pragma at the first line. When we compile the function *fromColoredtoBW*, we get the *C* code where code blocks are annotated with source-locations of their origins (refer to Appendix D). Although parts of the code are fused together, the result of our tracking system is reasonably acceptable.

4.6 Related and Future Work

Traditionally, in order to track generated expressions, some of the nodes in the high-level AST have been annotated with source location information and the main transformation has been designed, with some ad-hoc heuristics, to preserve and transfer these annotations from the input to the output. For instance, Haskell-Src-Exts[4] [Bro12], in its interface AST (*Language.Haskell.Exts.Syntax*), carries source location information for top-level declarations, bindings and lambda expressions. This solution also is used in tracking changes made by macros in *Lisp*, *Scheme* and *Racket* [CF07, THSAC+11]. They may assign hash codes to some nodes and use hash tables to store the related information. We considered the existing techniques "ad-hoc", as they do not provide clear specifications by which validity of an implemented transformation could be verified; they do not give clear answers to some key questions facing the implementors:

[3]http://netpbm.sourceforge.net/doc/ppm.html

[4]the most popular package for parsing and manipulating Haskell code extended with GHC and other extensions [BFS04, Bro05]

1. what should be the information that the nodes in the high-level AST are annotated with? It is practically impossible to annotate every single node in the high-level AST with its own specific source location information.
2. which nodes in the high-level AST should be annotated?
3. what should be done with the annotations during the forward transformation? how is this behavior specified?
4. how should the generated nodes be annotated?
5. what happens if a node is removed?
6. what should be done if a node is duplicated?
7. what happens to the annotations, if two or more nodes in the high-level AST are combined to form one or more nodes in the low-level AST?
8. when we want to track a node in the low-level AST, how do we use its annotation to find the related nodes in the high-level AST?
9. how do we guarantee that we are tracking back to the right origins?
10. what should be done if an annotation is not valid or understandable by the tracking algorithm?

As was originally mentioned in the GRACE meeting notes [CFH$^+$09], questions of this kind are the subject of the interdisciplinary study bidirectional transformations (BX). We modeled the problem of tracking generated expressions as a BX system and proposed a solution by employing the semantic bidirectionalization technique. The laws governing validity of a bidirectional transformation (mainly acceptability) provide a form of specification to steer our design. Our implementation respects these laws (chapter 3). In addition, we force the annotations to be kept polymorphic (abstract) and as we discussed before (sections 3.1 and 3.2), this will provide us with the additional guarantee that the annotations cannot be generated or observed arbitrarily. Finally, we specified a simple and practical condition for the transformation to preserve the annotations and we designed an algorithm to transform an existing system to a system satisfying this condition.

Now, we can answer the above questions as follows:

1. We use Boolean values to annotate every single node in an AST and we exactly know which nodes are selected. Hence, we know the exact origins. If the system requires extra information available only for a few nodes, e.g. source-locations for the top-level bindings, and if an origin node lacks the information, we move up in the tree to find the first node with the required information. It is an approximation method to compensate for the lack of information and in our experiments with *Feldspar*, this method seems to work well in practice.

2. In our solution, every single node in the high-level AST should be annotated with Boolean values.

3. We specified the condition of *annotation preservation* and also designed an algorithm (section 4.3.1) to transform forward functions to respect this condition.

4. Since we keep the annotations abstract during the forward transformations, the generated nodes cannot have annotations. If they are generated by pattern matching on values of annotated data types, then our algorithm wraps the generated nodes in a node carrying the annotations of deconstructed values.

5. If the node is not destructed, its annotation is lost as expected. It is not possible to select a removed node, hence its annotation is not needed. Note that this behavior still satisfies the annotation preservation condition, since the removed node is not destructed. If the node is destructed, then its annotation is injected into the selected expression.

6. Semantic-bidirectionalization can detect duplication. What to do in that case is a design decision that should be made depending on the use-case scenario. One solution is to give priority to the value *True*, i.e., even if only one single node of many duplicated ones is selected, the system assumes the others selected.

7. The resulting node is wrapped in a node carrying the annotations of the ones that are deconstructed. The others carry their own annotations and are copied as is.

8. We set its annotation to *True* and let the semantic-bidirectionalization algorithm put back the changes in the source. In this updated source, the annotation of the origins are set to *True*.

9. The BX laws governing the semantic-bidirectionalization provides us with the guarantee.

10. Since we keep the annotations abstract, it is impossible to come up with new and not understandable annotations.

The work [Wan10] has been one of our main inspirations. However, in [Wan10], the syntactic approach has been used to map back the error messages referring to generated expressions. As mentioned earlier, the syntactic approach can only bidirectionalize functions in a restricted subset (section 2.2.3.1) of a functional language, *Haskell* in particular. The author in [Wan10] argues that these restrictions are not too restrictive in practice. Our solution employs the semantic approach. One point in using the semantic bidirectionalization is that the technique does not need to access the source code of the forward function. While our solution uses the semantic bidirectionalization technique, the annotation preservation algorithm needs to access and transform the source code of the forward function. Nevertheless, unlike the syntactic bidirectionalization [MHN$^+$07], our algorithm does not set restrictions on the language under transformation. In fact, our algorithm can work with any program written in *Haskell*. For example, a solution based on syntactic bidirectionalization cannot (while our solution can) track the output of a compile function that duplicates an expression; a duplicating function is not affine (section 2.2.3.1). One instance of such a duplicating compile function is one that compiles case expressions[5].

Our algorithm transforms a system with *closed* data type; there are solutions based on *open* [Swi08, Axe12, BH11] data types demanding much less effort and changes to the existing code. For instance, if the code is written in their proposed encoding [BH11], the annotations are carried in an orthogonal procedure, i.e., the main transformation is totally agnostic towards the annotations.

It would be interesting to use code profilers for the generated *C* code and use our system to track this data to profile the high-level *Haskell* code. These feedback information would help the programmers to refine their high-level code without the need to examine the generated spaghetti code.

One other interesting potential is to use our mechanism to make programs *resource-aware*. That is, we pass the feedback information to the program itself, a subject related to the study of *self-optimizing code* and *incremental computing* [Car02, Aca05, WGW11].

[5]for instance refer to the rule g of the section 3.17.3 of the language report of Haskell-2010

Chapter 5

Conclusion

Semantic-bidirectionalization was originally introduced as three distinct higher-order functions that could bidirectionalize forward functions with specific type signatures. We started by refactoring the original algorithm in order to unify these mechanisms. In the process, we identified the conditions that should be respected to form a lawful bidirectional transformation. We introduced an abstract system parametric over the indexing function and proved the soundness of the system with respect to BX laws. Then, we introduced a general indexing function working in the presence of an arbitrary observer function. At the end, we applied generic programming techniques to extend our system to bidirectionalize forward functions with arbitrary polymorphic data structures as the source and the view. We demonstrated that our algorithm can bidirectionalize 40% of the functions defined in the *Prelude* module. We had 20% improvement compared to the original technique.

As the practical part, we started with the problem of tracking generated expressions. We applied semantic-bidirectionalization techniques to track the expressions from the AST of the generated code to the AST of the original code. Our algorithm required the system to preserve and carry annotation data. First, we specified what we meant by preservation of the annotation data and then we sketched an algorithm to transform an incapable system to be able to preserve and carry the annotations. Since in embedded languages, the parsing phase is omitted, the information about the exact location of expressions in the original source code is lost. In order to track generated expressions in an embedded language all the way up to their exact source location in the high-level source file, we needed to recover this lost information. For this purpose, we developed a preprocessor (QuickAnnotate) to inject the source location information into the expressions of the right type using type-level programming. For testing and demonstrating our solution, we developed Pico-Feldspar. Finally, we enhanced *Feldspar* language with the ability to track the generated *C* expressions to their origins in the Haskell code; this mechanism is fully implemented and now it is a part of the released version of *Feldspar*.

It was a long journey; the student learned a great deal on subjects surrounding bidirectional transformation, language transformations and equational reasoning. The student studied some of the well-respected bidirectionalization techniques and improved one of these by designing a new algorithm; designed and developed tools to transform programs; and finally he practiced equational reasoning by proving correctness of his proposed algorithm. By working closely on the *Feldspar* project, the student had the opportunity to study the novel embedding techniques in the project. More noticeably, through this thesis, the student improved his skill in technical writing and critical thinking; the student learnt how to patiently observe an existing model, extract the underlying properties, formulate them and refine the model based on these newly found properties.

Bibliography

[Aca05] Umut A Acar. *Self-adjusting computation.* PhD thesis, Princeton University, 2005.

[ACS+11] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of feldspar an embedded language for digital signal processing. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[Axe12] Emil Axelsson. A generic abstract syntax model for embedded languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 323–334, New York, NY, USA, 2012. ACM.

[BFS04] Niklas Broberg, Andreas Farre, and Josef Svenningsson. Regular expression patterns. In *Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 67–78, New York, NY, USA, 2004. ACM.

[BH11] Patrick Bahr and Tom Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, pages 83–94, New York, NY, USA, 2011. ACM.

[BJC10] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 125–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[Bro05] Niklas Broberg. Haskell server pages through dynamic loading. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05, pages 39–48, New York, NY, USA, 2005. ACM.

[Bro12] Niklas Broberg. haskell-src-exts: Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer. `http://hackage.haskell.org/package/haskell-src-exts`, September 2012.

[Car02] Magnus Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 26–35, New York, NY, USA, 2002. ACM.

[CF07] Ryan Culpepper and Matthias Felleisen. Debugging macros. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 135–144, New York, NY, USA, 2007. ACM.

[CFH$^+$09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[EW12] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Symposium on Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM.

[FGM$^+$07] 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 Transactions on Programming Languages and Systems*, 29(3), May 2007.

[FMV12] Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three complementary approaches to bidirectional programming. In *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming*, SSGIP'10, pages 1–46, Berlin, Heidelberg, 2012. Springer-Verlag.

[HMT04] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '04, pages 178–189, New York, NY, USA, 2004. ACM.

[Hug95] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 53–96, London, UK, UK, 1995. Springer-Verlag.

[Jon93] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 52–61, New York, NY, USA, 1993. ACM.

[Jon95] Mark P. Jones. *Qualified types: theory and practice.* Cambridge University Press, New York, NY, USA, 1995.

[Mai07] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM.

[MHN$^+$07] 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.

[MHT04a] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 2–20, Berlin, Heidelberg, 2004. Springer-Verlag.

[MHT04b] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *In Seventh International Conference on Mathematics of Program Construction*, MPC 2004, pages 289–313, Berlin, Heidelberg, 2004. SpringerVerlag.

[MMHT10] Kazutaka Matsuda, Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. A grammar-based approach to invertible programs. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 448–467, Berlin, Heidelberg, 2010. Springer-Verlag.

[MP08] Conor Mcbride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, January 2008.

[PJS04] S. Peyton Jones and M. Shields. Lexically-scoped type variables. *Submitted to ICFP*, 2004.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. *Proceedings of the IFIP 9th World Computer Congress*, pages 513–523, 1983.

[SPJCS08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM.

[Swi08] Wouter Swierstra. Functional pearl: Data types a la carte. *Journal of Functional Programming*, 18(4):423, 2008.

[THSAC$^+$11] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.

[VHMW10] 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.

[Voi09] 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.

[Wad88] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the 2nd European Conference on Programming Languages and Systems*, ESOP'88, pages 231–248, Amsterdam, The Netherlands, 1988. North-Holland Publishing Co.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.

[Wan10] Meng Wang. *Bidirectional Programming and its application.* PhD thesis, University of Oxford, 2010.

[WGW11] Meng Wang, Jeremy Gibbons, and Nicolas Wu. Incremental updates for efficient bidirectional transformations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 392–403, New York, NY, USA, 2011. ACM.

[YWC$^+$12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

Appendices

Appendix A

PROOF.[Acceptability of bff_{IBy} (theorem 10)]

```
    $\mathit{bff}_{IBy}\ obs_I\ index_{By}\ get_{By}\ obs_X\ s\ (get_{By}\ obs_X\ s)$
  = {-definition of $\mathit{bff}_{IBy}$ and the premiss -}
    do
      let $ms = index_{By}\ obs_X\ s$
      $unless\ (length\ (get_{By}\ obs_X\ s) ==$
              $length\ (get_{By}\ obs_I\ (fst\ `map`\ ms)))$
        $ Left "Modified view of wrong length!"
      let $mv = zip\ (get_{By}\ obs_I\ (fst\ `map`\ ms))$
                $(get_{By}\ obs_X\ s)$
      $unless\ (validAssoc\ mv)$
        $ Left "Inconsistent duplicated values!"
      let $ms' = union\ mv\ ms$
      $unless\ (check_{IBy}\ obs_I\ obs_X\ ms')$
        $ Left "Invalid modified view!"
      return $ lookupAll (fst `map` ms) ms'
  = {-lemma 1, premiss (input preservation) and specification of unless -}
    do
      let $ms = index_{By}\ obs_X\ s$
      let $mv = zip\ (get_{By}\ obs_I\ (fst\ `map`\ ms))$
                $(get_{By}\ obs_X\ s)$
      $unless\ (validAssoc\ mv)$
        $ Left "Inconsistent duplicated values!"
      let $ms' = union\ mv\ ms$
      $unless\ (check_{IBy}\ obs_I\ obs_X\ ms')$
        $ Left "Invalid modified view!"
      return $ lookupAll (fst `map` ms) ms'
  = {-lemma 2, premiss (input preservation) -}
  do
    let ms = index_By  obs_X s
    let mv = get_By (obs_X `on` snd) ms
    unless (validAssoc mv)
      $ Left "Inconsistent duplicated values!"
    let ms' = union mv ms
    unless (check_IBy  obs_I obs_X ms')
      $ Left "Invalid modified view!"
    return $ lookupAll (fst `map` ms) ms'
=  {-replacing mv everywhere with its definition -}
  do
    let ms = index_By  obs_X s
    unless (validAssoc (get_By (obs_X `on` snd) ms))
      $ Left "Inconsistent duplicated values!"
    let ms' = union (get_By (obs_X `on` snd) ms) ms
    unless (check_IBy  obs_I obs_X ms')
      $ Left "Invalid modified view!"
    return $ lookupAll (fst `map` ms) ms'
=  {-lemma 3, premiss (validAssoc ms = True) and specification of unless -}
  do
    let ms = index_By  obs_X s
    let ms' = union (get_By (obs_X `on` snd) ms) ms
    unless (check_IBy  obs_I obs_X ms')
      $ Left "Invalid modified view!"
    return $ lookupAll (fst `map` ms) ms'
=  {-lemma 4 and replacing ms' everywhere with its definition -}
  do
    let ms = index_By  obs_X s
    unless (check_IBy  obs_I obs_X ms)
      $ Left "Invalid modified view!"
    return $ lookupAll (fst `map` ms) ms
=  {-premiss (map invariant) -}
  do
    let ms = index_By  obs_X s
    return $ lookupAll (fst `map` ms) ms
=  {-lemma 5 and premiss (input preservation) -}
  Right s
```

□

Lemma 1 *Let X, A and I be types; let $get_{By} :: \forall t.(t \to t \to X) \to [t] \to [t]$, $obs_X :: A \to A \to X$ and $obs_I :: I \to I \to X$ be functions; and let $s :: [A]$ and $ms :: [(I,A)]$. If we have $s = map\ snd\ ms$ (input preservation), then*

$$(length\ (get_{By}\ obs_X\ s) == length\ (get_{By}\ obs_I\ (fst\ \text{`}map\text{`}\ ms))) = True$$

$\square$

Proof.

$$
\begin{aligned}
& length\ (get_{By}\ obs_X\ s) == \\
& length\ (get_{By}\ obs_I\ (fst\ \text{`}map\text{`}\ ms)) \\
= \ & \{\text{-premiss (input preservation) -}\} \\
& length\ (get_{By}\ obs_X\ (snd\ \text{`}map\text{`}\ ms)) == \\
& length\ (get_{By}\ obs_I\ (fst\ \text{`}map\text{`}\ ms)) \\
= \ & \{\text{-free theorem -}\} \\
& length\ (get_{By}\ obs_X\ (snd\ \text{`}map\text{`}\ ms)) == \\
& length\ (map\ fst\ (get_{By}\ (obs_X\ \text{`}on\text{`}\ snd)\ ms)) \\
= \ & \{\text{-free theorem -}\} \\
& length\ (map\ snd\ (get_{By}\ (obs_X\ \text{`}on\text{`}\ snd)\ ms)) == \\
& length\ (map\ fst\ (get_{By}\ (obs_X\ \text{`}on\text{`}\ snd)\ ms)) \\
= \ & \{\text{-free theorem (length x = length (map g x)) -}\} \\
& length\ (get_{By}\ (obs_X\ \text{`}on\text{`}\ snd)\ ms) == \\
& length\ (map\ fst\ (get_{By}\ (obs_X\ \text{`}on\text{`}\ snd)\ ms)) \\
= \ & \{\text{-free theorem (length x = length (map g x)) -}\} \\
& length\ (get_{By}\ (obs_X\ \text{`}on\text{`}\ snd)\ ms) == \\
& length\ (get_{By}\ (obs_X\ \text{`}on\text{`}\ snd)\ ms) \\
= \ & \\
& True
\end{aligned}
$$

$\square$

Lemma 2 *Let X, A and I be types; let $get_{By} :: \forall t.(t \to t \to X) \to [t] \to [t]$, $obs_X :: A \to A \to X$ and $obs_I :: I \to I \to X$ be functions; and let $s :: [A]$ and $ms :: [(I,A)]$. If we have $s = map\ snd\ ms$ (input preservation), then*

$$zip\ (get_{By}\ obs_I\ (fst\ \text{`}map\text{`}\ ms))\ (get_{By}\ obs_X\ s) = get_{By}\ (obs_X\ \text{`}on\text{`}\ snd)\ ms$$

$\square$

Proof.

$$\begin{aligned}
& \mathit{zip}\ (\mathit{get}_{By}\ \mathit{obs}_I\ (\mathit{fst}\ \text{`}\mathit{map}\text{`}\ \mathit{ms})) \\
& \quad (\mathit{get}_{By}\ \mathit{obs}_X\ s) \\
= \ & \text{\{-premiss (input preservation) -\}} \\
& \mathit{zip}\ (\mathit{get}_{By}\ \mathit{obs}_I\ (\mathit{fst}\ \text{`}\mathit{map}\text{`}\ \mathit{ms})) \\
& \quad (\mathit{get}_{By}\ \mathit{obs}_X\ (\mathit{snd}\ \text{`}\mathit{map}\text{`}\ \mathit{ms})) \\
= \ & \text{\{-free theorem - two times -\}} \\
& \mathit{zip}\ (\mathit{map}\ \mathit{fst}\ (\mathit{get}_{By}\ (\mathit{obs}_X\ \text{`}\mathit{on}\text{`}\ \mathit{snd})\ \mathit{ms})) \\
& \quad (\mathit{map}\ \mathit{snd}\ (\mathit{get}_{By}\ (\mathit{obs}_X\ \text{`}\mathit{on}\text{`}\ \mathit{snd})\ \mathit{ms})) \\
= \ & \text{\{-specification of zip -\}} \\
& \text{\{-i.e. zip (map fst x) (map snd x) = x -\}} \\
& \mathit{get}_{By}\ (\mathit{obs}_X\ \text{`}\mathit{on}\text{`}\ \mathit{snd})\ \mathit{ms}
\end{aligned}$$

□

Lemma 3 *Let X, A and I be types; let $\mathit{get}_{By} :: \forall t.(t \to t \to X) \to [t] \to [t]$ and $\mathit{obs}_X :: A \to A \to X$ be functions; and let $\mathit{ms} :: [(I,A)]$. If we have $\mathit{validAssoc}\ \mathit{ms} = \mathit{True}$, then*

$$\mathit{validAssoc}\ (\mathit{get}_{By}\ (\mathit{obs}_X\ \text{`}\mathit{on}\text{`}\ \mathit{snd})\ \mathit{ms}) = \mathit{True}$$

□

Proof.

By free theorems and the premiss $\mathit{validAssoc}\ \mathit{ms} = \mathit{True}$

□

Lemma 4 *Let X, A and I be types; let $\mathit{get}_{By} :: \forall t.(t \to t \to X) \to [t] \to [t]$ and $\mathit{obs}_X :: A \to A \to X$ be functions; and let $\mathit{ms} :: [(I,A)]$. We have*

$$\mathit{union}\ (\mathit{get}_{By}\ (\mathit{obs}_X\ \text{`}\mathit{on}\text{`}\ \mathit{snd})\ \mathit{ms})\ \mathit{ms} = \mathit{ms}$$

□

Proof.

By free theorems and the specification of union

□

Lemma 5 *Let A and I be types; and let $\mathit{ms} :: [(I,A)]$. For some $s :: [A]$, if $s = \mathit{map}\ \mathit{snd}\ \mathit{ms}$ (input preservation), then we have*

$$\mathit{lookupAll}\ (\mathit{fst}\ \text{`}\mathit{map}\text{`}\ \mathit{ms})\ \mathit{ms} = s$$

□

PROOF.

$$
\begin{array}{ll}
 & \mathit{lookupAll}\ (\mathit{fst}\ \text{`}\mathit{map}\text{`}\ \mathit{ms})\ \mathit{ms} \\
= & \text{\{-Specification of lookupAll -\}} \\
 & \mathit{snd}\ \text{`}\mathit{map}\text{`}\ \mathit{ms} \\
= & \text{\{-premiss (Input Preservation) -\}} \\
 & s
\end{array}
$$

□

Appendix B

Module Feldspar

This module is used as a front-end to the Feldspar language. It re-exports from the internal modules.

```
 {-# LANGUAGE DataKinds #-}
module Feldspar (module Feldspar.FrontEnd.Interface
  ,Data,Int32,Num (..),String) where
import qualified Prelude
import Prelude (String,Num (..))


import Feldspar.FrontEnd.Interface
import qualified Feldspar.FrontEnd.AST as AST
import qualified Feldspar.Types as Types


import Feldspar.Annotations ()


type Data a = AST.Data a ( String )
type Int32 = Types.Int32
```

Module Annotations

The module containing the type classes and the functions defined in chapter 4 (refer to 4) to facilitate injecting, projecting and preserving the annotations.

```
{-# LANGUAGE TypeFamilies #-}
module Annotations where
import qualified Prelude
import Prelude (Maybe (..))
```

```
  -- injecting annotations into data
class Inj t where
  type Ann t
  inj :: Ann t → t → t
```

```
  -- projecting the stored annotations
class Inj t ⇒ Annotatable t where
  prj :: t → Maybe ((Ann t,t))
```

```
  -- preserving the annotations
preserve :: ∀ t↑ t↓.
  (Annotatable t↑,Inj t↓
  ,Ann t↑∼Ann t↓) ⇒
  t↑ → (t↑ → t↓) → t↓
preserve e↑ f = case prj e↑ of
  Just (ann,é↑) → inj ann (f é↑)
  Nothing → f e↑
```

Module BX

This module contains the code for our semantic bidirectionalization algorithm, described in the chapter 3 (refer to 3).

```
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveTraversable #-}
{-# LANGUAGE Rank2Types #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE FlexibleContexts #-}
```

```
module BX where
import Data.Foldable (Foldable (..),toList)
import Data.Traversable (Traversable (..))
import Control.Monad (unless,join)
import qualified Prelude
import Prelude (String,Bool (..),Int,Eq (..),Either (..)
  ,Functor (..),Monad (..),Maybe (..),($),(!!),(∨),(+),(.)
  ,¬,fst,⊥,snd,map,lookup,length,and,zip
  ,const,flip,mapM)
import Control.Monad.State (get,put,evalState)
import Data.Function (on)
import Data.List (unionBy)
```

```
fromJust :: Maybe a → a
fromJust (Just x) = x
fromJust Nothing = ⊥
```

```
index :: ∀ a.[a] → [(Int,a)]
index s = zip [1 .. length s] s
```

```
assoc :: ∀ a b.[a] → [b] → [(a,b)]
assoc = zip
```

```
validAssoc :: ∀ a b.(Eq a,Eq b) ⇒
  [(a,b)] → Bool
validAssoc mv = and
  [¬ (i == j) ∨ x == y | (i,x) ← mv,(j,y) ← mv]
```

```
union :: ∀ a b.Eq a ⇒
  [(a,b)] → [(a,b)] → [(a,b)]
union = unionBy ((==) ‘on‘ fst)
```

```
lookupAll :: ∀ a b.Eq a ⇒
  [a] → [(a,b)] → [b]
lookupAll is mp = map (fromJust.flip lookup mp) is
```

```
data Nat =
  Zero
  | Succ Nat
```

```
infixr 5 :::
data Vect :: Nat → ∗ → ∗ where
  Nil :: Vect Zero a
  (:::) :: a → Vect n a → Vect (Succ n) a
```

```
instance Functor (Vect n) where
  fmap _ Nil = Nil
  fmap f (x ::: xs) = f x ::: fmap f xs
```

```
data Sing_Nat :: Nat → ∗ where
  Zero_Sing :: Sing_Nat Zero
  Succ_Sing :: Sing_Nat n → Sing_Nat (Succ n)
```

```
class SingI (n :: Nat) where
  sing :: Sing_Nat n
```

```
instance SingI Zero where
  sing = Zero_Sing
```

```
instance SingI n ⇒ SingI (Succ n) where
  sing = let
    n = (sing :: Sing_Nat n)
    in Succ_Sing n
```

```
class (SingI (Size t)) ⇒ Vect_Iso (t :: ∗ → ∗) where
  type Size t :: Nat
  toVect :: ∀ a.t a → Vect (Size t) a
  fromVect :: ∀ a. Vect (Size t) a → t a
```

```
size :: ∀ a t.(SingI (Size t), Vect_Iso t) ⇒
      t a → Sing_Nat (Size t)
size _ = sing
```

```
perm :: Sing_Nat (Succ m) → [(i,a)] →
  [Vect (Succ m) (i,a)]
perm (Succ_Sing Zero_Sing) ms = (:::Nil) ‘map‘ ms
perm (Succ_Sing (Succ_Sing n)) ms = join
  [((i,x):::) ‘map‘ (perm (Succ_Sing n) ms) | (i,x) ← ms]
```

```
check_GBy :: ∀ t a x s.
  (Vect_Iso t,Size t∼Succ s,Eq x) ⇒
  (t Int → x) → (t a → x) → [(Int,a)] → Bool
check_GBy obs_I obs_X ms = let
      vs = perm (size (⊥ :: t Int)) ms
  in and
     [obs_I (fromVect (fmap fst z)) ==
        obs_X (fromVect (fmap snd z))
     | z ← vs]
```

```
onG :: Vect_Iso t ⇒
  (t b → c) → (a → b) → (t a → c)
onG f f′ = f.fromVect.(fmap f′).toVect
```

```
newtype ()^1 a = ()^1 a
newtype ()^2 a = ()^2 (a,a)
newtype ()^3 a = ()^3 (a,a,a)
```

```
instance Vect_Iso ()^1 where
  type Size ()^1 = Succ Zero
  toVect (()^1 x) = x ::: Nil
  fromVect (x ::: Nil) = ()^1 x
```

```
instance Vect_Iso ()^2 where
  type Size ()^2 = Succ (Succ Zero)
  toVect (()^2 (x_1,x_2)) =
              x_1 ::: x_2 ::: Nil
  fromVect (x_1 ::: x_2 ::: Nil) =
              ()^2 (x_1,x_2)
```

```
instance Vect_Iso ()^3 where
  type Size ()^3 = Succ (Succ (Succ Zero))
  toVect (()^3 (x_1,x_2,x_3)) =
              x_1 ::: x_2 ::: x_3 ::: Nil
  fromVect (x_1 ::: x_2 ::: x_3 ::: Nil) =
              ()^3 (x_1,x_2,x_3)
```

```
fromList :: ∀ k a b.Traversable k ⇒
  k a → [b] → k b
fromList s lst = let
  indices _ = do
    i ← get
    put (i + 1)
    return i
  si = evalState
       (Data.Traversable.mapM indices s) 0
  in fmap (lst!!) si
```

```
(==_Shape) :: ∀ k a.(Eq (k ()),Foldable k,Functor k) ⇒
  k a → k a → Bool
(==_Shape) = (==) `on` fmap (const ())
```

```
bff^{a−*}_{GUS} :: ∀ x t s.
  (Vect_Iso t,Eq x,Size t∼Succ s) ⇒
  (∀ a.(t a → x) → [a] → [a]) →
  (∀ a.Eq a ⇒
  (t a → x) → [a] → [a] → Either String [a])
bff^{a−*}_{GUS} get_By obs_X s v = do
    -- Step 1
  let ms = index s
  let obs_I = onG obs_X (fromJust.(flip lookup ms))
    -- Step 2
  let is  = fst `map` ms
  let iv  = get_By obs_I is
    -- Step 3
  unless (length v == length iv)
    $ Left "Modified view of wrong length!"
  let mv = assoc iv v
    -- Step 4
  unless (validAssoc mv)
    $ Left "Inconsistent duplicated values!"
    -- Step 5
  let ms′ = union mv ms
    -- Step 5.1
  unless (check_GBy obs_I obs_X ms′)
    $ Left "Invalid modified view!"
    -- Step 6
  return $ lookupAll is ms′
```

```
bff^{a/d-*}_{GUS} :: ∀ x k k′ t s.
  (Vect_{Iso} t,Functor k′,Foldable k′,Eq (k′ ())
  ,Size t∼Succ s,Traversable k,Eq x) ⇒
  (∀ a.(t a → x) → k a → k′ a) →
  (∀ a.(Eq a) ⇒
     (t a → x) → k a → k′ a → Either String (k a))
bff^{a/d-*}_{GUS} get_{By} obs_{X} s v = do
  let s^{list}  = toList s
  let v^{list}  = toList v
  let get^{list}_{By} :: ∀ a.(t a → x) → [a] → [a]
      get^{list}_{By} obs x = toList $ get_{By} obs (fromList s x)
  unless ((==_{Shape}) (get_{By} obs_{X} s) v)
     $ Left "Modified view of wrong shape!"
  ś^{list} ← bff^{a-*}_{GUS} get^{list}_{By} obs_{X} s^{list} v^{list}
  return $ fromList s ś^{list}
```

Module Feldspar.Compiler

This module is used as a front-end to the Feldspar compiler. It re-exports from the internal modules.

```
module Feldspar.Compiler (icompile,scompile,IO) where
import Feldspar.Compiler.Compiler
```

Module Feldspar.Types

This module contains the declaration of the built-in types in Pico-Feldspar. It also includes the code defining singlton types and the utility functions for promotion and demotion of the built-in types.

```
 {-# LANGUAGE GADTs #-}
 {-# LANGUAGE DataKinds #-}
 {-# LANGUAGE KindSignatures #-}
 {-# LANGUAGE ScopedTypeVariables #-}
module Feldspar.Types where
import qualified Prelude
import Prelude (Eq (..))
```

```
  -- the built-in types
  -- it is usually used in the promoted form
data Types = Int32 | Bool
  deriving Eq
```

```
  -- a GADT representation of a singleton type for
  -- the built-in types
data SingTypes :: Types → ∗where
  SInt32 :: SingTypes Int32
  SBool :: SingTypes Bool
```

```
  -- overloaded function to demote singletons
class SingT (n :: Types) where
  sing :: SingTypes n
instance SingT Int32 where
  sing = SInt32
instance SingT Bool where
  sing = SBool
```

```
  -- coversion from singleton types to the original
toTypes :: SingTypes n → Types
toTypes SInt32 = Int32
toTypes SBool = Bool
```

```
  -- overloaded function to demote singletons
  -- to the original
getType :: ∀ k n a.SingT n ⇒ k n a → Types
getType _ = toTypes (sing :: SingTypes n)
```

```
  -- an overloaded function to facilitate demotion
  -- using the type of the argument of a function
getTypeF :: ∀ k n a r.SingT n ⇒
  (k n a → r) → SingTypes n
getTypeF _ = sing :: SingTypes n
```

Module Feldspar.Annotations

In this module, the type classes defined in the module *Annotations* are derived for the main data types in Pico-Feldspar.

```
 {-# LANGUAGE TypeFamilies #-}
 {-# LANGUAGE FlexibleInstances #-}
module Feldspar.Annotations
  (module Annotations) where
import qualified Prelude as P
import Prelude (map,Int,Maybe (..),String)
```

```
import qualified QuickAnnotate as QA
import Feldspar.FrontEnd.AST
import Feldspar.BackEnd.AST
import Annotations (Inj (..),Annotatable (..),preserve)
import Feldspar.BackEnd.Pretty (Pretty (..))
```

```
import Control.Monad.State
import Text.PrettyPrint (text)
```

```
instance QA.Annotatable (Data a String) where
  annotate loc d = inj loc d
```

```
instance Inj (Data a ann) where
  type Ann (Data a ann) = ann
  inj x = Ann x
```

```
instance Annotatable (Data a ann) where
  prj (Ann x e) = Just (x,e)
  prj _ = Nothing
```

```
instance Inj (Exp_C ann) where
  type Ann (Exp_C ann) = ann
  inj x = Ann_ExpC x
```

```
instance Annotatable (Exp_C ann) where
  prj (Ann_ExpC x e) = Just (x,e)
  prj _ = Nothing
```

```
instance Inj (Stmt ann) where
  type Ann (Stmt ann) = ann
  inj x = Ann_Stmt x
```

```
instance Annotatable (Stmt ann) where
  prj (Ann_Stmt x e) = Just (x,e)
  prj _ = Nothing
```

```
instance Inj (Func ann) where
  type Ann (Func ann) = ann
  inj ann (Func x ps stmts) =
    Func x ps (inj ann `map` stmts)
```

```
instance Inj t ⇒ Inj [t] where
  type Ann [t] = Ann t
  inj x = map (inj x)
```

```
instance (Inj t1,Inj t2,Ann t1∼Ann t2) ⇒
  Inj (t1,t2) where
  type Ann (t1,t2) = Ann t1
  inj x (e1,e2) = (inj x e1,inj x e2)
```

```
instance Inj t ⇒ Inj (State Int t) where
  type Ann (State Int t) = Ann t
  inj x = fmap (inj x)
```

```
instance Inj r ⇒ Inj (a → r) where
  type Ann (a → r) = Ann r
  inj ann f = λx → inj ann (f x)
```

```
instance Pretty String where
  pretty = text
```

Module Feldspar.AnnotationUtils

This module provides a set of utilities to work with annotations, e.g., removing all the annotations from an AST *stripAnn* or annotating every single node in an AST with the value *False*.

```
 {-# LANGUAGE GADTs #-}
 {-# LANGUAGE FlexibleInstances #-}
module Feldspar.AnnotationUtils where
import qualified Prelude as P
import Prelude (Maybe (..),map,($),(.))
```

```
import Feldspar.FrontEnd.AST
import Feldspar.BackEnd.AST
```

```
import Annotations (Inj (..))
import Feldspar.Annotations ()
```

```
  -- removing all the annotations
stripAnn :: Data a ann → Data a ann'
stripAnn (Var      x) = Var    x
stripAnn (Lit_Int  i) = Lit_Int i
stripAnn (Lit_Bool b) = Lit_Bool b
stripAnn (Not      e) = Not (stripAnn e)
stripAnn (Add     e1 e2) =
  Add (stripAnn e1) (stripAnn e2)
stripAnn (Sub     e1 e2) =
  Sub (stripAnn e1) (stripAnn e2)
stripAnn (Mul     e1 e2) =
  Mul (stripAnn e1) (stripAnn e2)
stripAnn (Eq_Int  e1 e2) =
  Eq_Int (stripAnn e1) (stripAnn e2)
stripAnn (LT_Int e1 e2) =
  LT_Int (stripAnn e1) (stripAnn e2)
stripAnn (And     e1 e2) =
  And (stripAnn e1) (stripAnn e2)
stripAnn (If e1 e2 e3) =
  If (stripAnn e1) (stripAnn e2) (stripAnn e3)
stripAnn (Ann _ e) = stripAnn e
```

```
  -- annotating each node in the output with False
markAllF :: ∀ a ann ann' r.
  (r ann' → r P.Bool) →
  (Data a ann → r ann') →
  (Data a P.Bool → r P.Bool)
markAllF markAllr f = markAllr.f.stripAnn
```

```
  -- annotating each node with False
markAll :: ∀ a ann.Data a ann
          → Data a P.Bool
markAll (Var      x) = Ann P.False $
  Var     x
markAll (Lit_Int  i) = Ann P.False $
  Lit_Int i
markAll (Lit_Bool b) = Ann P.False $
  Lit_Bool b
markAll (Not      e) = Ann P.False $
  Not (markAll e)
markAll (Add    e1 e2) = Ann P.False $
  Add (markAll e1) (markAll e2)
markAll (Sub    e1 e2) = Ann P.False $
  Sub (markAll e1) (markAll e2)
markAll (Mul    e1 e2) = Ann P.False $
  Mul (markAll e1) (markAll e2)
markAll (Eq_Int e1 e2) = Ann P.False $
  Eq_Int (markAll e1) (markAll e2)
markAll (LT_Int e1 e2) = Ann P.False $
  LT_Int (markAll e1) (markAll e2)
markAll (And    e1 e2) = Ann P.False $
  And (markAll e1) (markAll e2)
markAll (If e1 e2 e3) = Ann P.False $
  If (markAll e1) (markAll e2) (markAll e3)
markAll (Ann _ e) =
  markAll e
```

```
  -- helper function
annCond :: ∀ k.Inj k ⇒
  Maybe (Ann k) → k → k
annCond (Just ann) e = inj ann e
annCond Nothing e = e
```

```
  -- pushing down the annotation, so the unannotated
  -- nodes inherit the parent's annotation
class PushDown t where
  pushDown :: (Maybe (Ann t)) →
    t → t
```

```
  -- pushing down the annotations for functions
instance PushDown r ⇒
  PushDown (Data a ann → r) where
  pushDown ann f = pushDown ann.f
```

```
  -- pushing down the annotation for terms of
  -- type Data a ann
instance PushDown (Data a ann) where
  pushDown ann (Var      x) = annCond ann $
    Var     x
  pushDown ann (Lit_Int  i) = annCond ann $
    Lit_Int i
  pushDown ann (Lit_Bool b) = annCond ann $
    Lit_Bool b
  pushDown ann (Not      e) = annCond ann $
    Not (pushDown ann e)
  pushDown ann (Add    e1 e2) = annCond ann $
    Add (pushDown ann e1) (pushDown ann e2)
  pushDown ann (Sub    e1 e2) = annCond ann $
    Sub (pushDown ann e1) (pushDown ann e2)
  pushDown ann (Mul    e1 e2) = annCond ann $
    Mul (pushDown ann e1) (pushDown ann e2)
  pushDown ann (Eq_Int e1 e2) = annCond ann $
    Eq_Int (pushDown ann e1) (pushDown ann e2)
  pushDown ann (LT_Int e1 e2) = annCond ann $
    LT_Int (pushDown ann e1) (pushDown ann e2)
  pushDown ann (And    e1 e2) = annCond ann $
    And (pushDown ann e1) (pushDown ann e2)
  pushDown ann (If e1 e2 e3) = annCond ann $
    If (pushDown ann e1) (pushDown ann e2)
      (pushDown ann e3)
  pushDown _ (Ann ann e) =
    pushDown (Just ann) e
  -- pushing down the annotation for terms of
  -- type Exp_C ann
instance PushDown (Exp_C ann) where
  pushDown ann (Var_C x)      = annCond ann $
    Var_C x
  pushDown ann (Num i)        = annCond ann $
    Num i
  pushDown ann (Infix e1 x e2) = annCond ann $
    Infix (pushDown ann e1) x (pushDown ann e2)
  pushDown ann (Unary x e) = annCond ann $
    Unary x (pushDown ann e)
  pushDown _ (Ann_ExpC ann e) =
    pushDown (Just ann) e
```

```
  -- pushing down the annotation for terms of
  -- type Stmt ann
instance PushDown (Stmt ann) where
  pushDown ann (If_C e stmts1 stmts2) = annCond ann $
    If_C (pushDown ann e) (pushDown ann `map` stmts1)
      (pushDown ann `map` stmts2)
  pushDown ann (Assign x e)        = annCond ann $
    Assign x (pushDown ann e)
  pushDown ann (Declare t x)       = annCond ann $
    Declare t x
  pushDown _ (Ann_Stmt ann stmt) =
    pushDown (Just ann) stmt
```

```
  -- pushing down the annotation for terms of
  -- type Func ann
instance PushDown (Func ann) where
  pushDown ann (Func x vs stmts) = Func x vs $
    pushDown ann `map` stmts
```

Module Feldspar.BX

This module provides the necessary functions to bidirectionalize the transformation from EDSL to C code by composing the bidirectionalization of each smaller transformations in between.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
module Feldspar.BX where
import qualified Prelude as P
import Prelude (String,Either (..),Maybe (..),Eq (..)
  ,Read (..),Monad (..),map,zip,(.),($))
import Data.Foldable (toList)
```

```
import Feldspar.Types
import Feldspar.FrontEnd.AST
import Feldspar.Compiler.BXCompiler (BXable (..))
import Feldspar.BackEnd.BXPretty (putPretty)
import Feldspar.Compiler.Compiler (toFunc,compile)
import Feldspar.BackEnd.Pretty (Pretty (..))
import Feldspar.AnnotationUtils (PushDown (..))
import Annotations (Inj (..))
```

```
  -- zipping similiar AST with different Annotations
class ZipData t t' where
  zipData :: t → t' → [(Ann t,Ann t')]
instance (SingT a,ZipData r r'
  ,Ann r~ann,Ann r'~ann') ⇒
  ZipData (Data a ann → r)
                    (Data a ann' → r') where
  zipData f g = zipData
    (f $ Var $ VarT "_x" sing)
    (g $ Var $ VarT "_x" sing)
instance ZipData (Data a ann) (Data a ann') where
  zipData d d' = zip (toList d) (toList d')
  -- putting back changes up to the src-loc
putAnn :: ∀ t t'.
  (PushDown t',BXable t,ZipData t t'
  ,Ann t~P.Bool) ⇒
  P.Bool → (t' → t) → t' → String →
  Either String [Ann t']
putAnn cn markA d src = do
  let dS = pushDown Nothing d
  let dM = markA d
  dU ← put cn dM src
  return [s | (b,s) ← zipData dU dS,b]
```

```
  -- putting back changes up to the high-level AST
put :: ∀ b.
     (Eq (Ann b),Read (Ann b),
       Pretty (Ann b),BXable b) ⇒
        P.Bool → b → String →
        Either String b
put b s v' = do
  let s' = (toFunc.compile 0) s
  let ps' = if b
    then pushDown Nothing s'
    else s'
  v ← putPretty ps' v'
  putCompile 0 s v
```

Module Feldspar.FrontEnd.AST

This module, provides the type-safe representation (via GADTs) of the high-level language.

```
 {-# LANGUAGE GADTs #-}
 {-# LANGUAGE DataKinds #-}
 {-# LANGUAGE KindSignatures #-}
module Feldspar.FrontEnd.AST where
import qualified Prelude as P
import Feldspar.Types
```

```
  -- AST of the EDSL (high-level)
data Data (a :: Types) ann where
  Var :: VarT a → Data a ann
  Lit_Int :: P.Int → Data Int32 ann
  Add :: Data Int32 ann → Data Int32 ann → Data Int32 ann
  Sub :: Data Int32 ann → Data Int32 ann → Data Int32 ann
  Mul :: Data Int32 ann → Data Int32 ann → Data Int32 ann
  Eq_Int :: Data Int32 ann → Data Int32 ann → Data Bool ann
  LT_Int :: Data Int32 ann → Data Int32 ann → Data Bool ann
  Lit_Bool :: P.Bool → Data Bool ann
  Not :: Data Bool ann → Data Bool ann
  And :: Data Bool ann → Data Bool ann → Data Bool ann
  If :: Data Bool ann → Data a ann → Data a ann → Data a ann

  Ann :: ann → Data a ann → Data a ann

  -- Variables
data VarT t = VarT P.String (SingTypes t)
```

Module Feldspar.FrontEnd.Interface

This module, provides some utility functions to program in the high-level language.

```
  {-# LANGUAGE FlexibleInstances #-}
  {-# LANGUAGE DataKinds #-}
module Feldspar.FrontEnd.Interface where
import qualified Prelude
import Prelude (Num (..),Int,($),Show,String)
import Feldspar.FrontEnd.AST
import Feldspar.Types

instance Num (Data Int32 ann) where
  fromInteger i = Lit_Int $ fromInteger i
  (+) = Add
  (−) = Sub
  (∗) = Mul
  signum x = condition (x < 0)
    (−1)
    (condition (x == 0) 0 1)
  abs x = (signum x) ∗ x
```

$(==) :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow$
$\quad \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$(==) = \mathit{Eq}_{\mathit{Int}}$

$(<) :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow$
$\qquad \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$(<) = \mathit{LT}_{\mathit{Int}}$

$(>) :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow$
$\qquad \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$e_1 > e_2 = \neg\ \$\ e_1 < e_2$

$(\leqslant) :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow$
$\quad \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$e_1 \leqslant e_2 = (e_1 < e_2) \wedge (e_1 == e_2)$

$(\geqslant) :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow \mathit{Data}\ \mathit{Int32}\ \text{ann} \rightarrow$
$\quad \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$e_1 \geqslant e_2 = (e_1 > e_2) \wedge (e_1 == e_2)$

$\mathit{true} :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$\mathit{true} = \mathit{Lit}_{\mathit{Bool}}\ \mathit{Prelude}.\mathit{True}$

$\mathit{false} :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$\mathit{false} = \mathit{Lit}_{\mathit{Bool}}\ \mathit{Prelude}.\mathit{False}$

$\neg :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Bool}\ \text{ann} \rightarrow \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$\neg = \mathit{Not}$

$(\wedge) :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Bool}\ \text{ann} \rightarrow \mathit{Data}\ \mathit{Bool}\ \text{ann} \rightarrow$
$\quad \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$(\wedge) = \mathit{And}$

$(\vee) :: \forall\ \text{ann}. \mathit{Data}\ \mathit{Bool}\ \text{ann} \rightarrow \mathit{Data}\ \mathit{Bool}\ \text{ann} \rightarrow$
$\quad \mathit{Data}\ \mathit{Bool}\ \text{ann}$
$x \vee y = \neg\ ((\neg\ x) \wedge (\neg\ y))$

$\mathit{condition} :: \forall\ a\ \text{ann}. \mathit{Data}\ \mathit{Bool}\ \text{ann} \rightarrow \mathit{Data}\ a\ \text{ann} \rightarrow$
$\quad \mathit{Data}\ a\ \text{ann} \rightarrow \mathit{Data}\ a\ \text{ann}$
$\mathit{condition} = \mathit{If}$

Module Feldspar.FrontEnd.Derivings

In this module, the type classes *Functor*, *Foldable* and *Traversable* are derived for the high-level AST.

```
  -- the code is omitted
```

Module Feldspar.Compiler.Compiler

This module, contains the main code for compiling the high-level AST to C code.

```
 {-# LANGUAGE GADTs #-}
 {-# LANGUAGE TypeSynonymInstances #-}
 {-# LANGUAGE FlexibleInstances #-}
 {-# LANGUAGE FlexibleContexts #-}
module Feldspar.Compiler.Compiler where
import qualified Prelude as P
import Prelude ((.),Show (..),putStrLn,IO
  ,Int,String,(++),(+),Monad (..))
import Control.Monad.State (State,put,get
  ,evalState)


import Feldspar.Types
import Feldspar.FrontEnd.AST
import Feldspar.BackEnd.AST
import Feldspar.BackEnd.Pretty


import Feldspar.Annotations


  -- the monadic function to compile the
  -- the high-level AST to a pair containing
  -- an expression containing the returned
  -- value and a list of statements; the
  -- state contains a counter to generate
  -- fresh variables
```

```
compileM :: SingT a ⇒ Data a ann →
  State Int (ExpC ann,[Stmt ann])
compileM (Var (VarT v _)) =
  return (VarC v,[ ])

compileM (LitInt x) =
  return (Num x,[ ])

compileM (LitBool P.True) =
  return (VarC "true",[ ])

compileM (LitBool P.False) =
  return (VarC "false",[ ])

compileM (Add e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "+" eC2
    ,st1 ++ st2)

compileM (Sub e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "-" eC2
    ,st1 ++ st2)

compileM (Mul e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "*" eC2
    ,st1 ++ st2)

compileM (EqInt e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "==" eC2
    ,st1 ++ st2)
```

```
compileM (LT_Int e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "<" eC2
    ,st1 ++ st2)
```

```
compileM (And e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "&&" eC2
    ,st1 ++ st2)
```

```
compileM (Not e1) = do
  (eC1,st1) ← compileM e1
  return (Unary "!" eC1
    ,st1)
```

```
compileM e@(If e1 e2 e3) = do
  i ← get
  put (i + 1)
  let v = "v" ++ (show i)
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  (eC3,st3) ← compileM e3
  return
    (Var_C v
    ,st1 ++
      [Declare (getType e) v
      ,If_C eC1
        (st2 ++ [Assign v eC2])
        (st3 ++ [Assign v eC3])])
```

```
compileM e = preserve e compileM
```

```
  -- overloaded function to compile
  -- regardless of AST being parametric
class Inj t ⇒
  Compilable t where
  compileF :: ([Var],t) →
    State Int
    ([Var],Types
    ,Exp_C (Ann t)
    ,[Stmt (Ann t)])



  -- a parametric AST is first applied to
  -- a fresh variable with the right type
  -- and then it is compiled
instance (SingT a,Compilable r) ⇒
      Compilable (Data a ann' → r) where
  compileF (ps,f) = do
    i ← get
    put (i + 1)
    let v = "v" ++ (show i)
      a = Var (VarT v (getTypeF f))
      r = f a
    compileF ((ps ++ [(v,getType a)]),r)



  -- a non-parametric AST is compiled in
  -- the normal way defined in compileM
instance SingT a ⇒
  Compilable (Data a ann) where
  compileF (ps,d) = do
    (e,sts) ← compileM d
    return (ps,getType d,e,sts)



  -- coversion to Func
toFunc :: ([Var],Types,Exp_C ann,[Stmt ann]) →
  Func ann
toFunc (ps,ty,exp_C,stmts) =
  Func "test" (ps ++ [("*out",ty)])
  (stmts ++ [Assign "*out" exp_C])
  -- running the state monad with a seed
compile :: Compilable a ⇒ Int →
  a → ([Var],Types,Exp_C (Ann a)
    ,[Stmt (Ann a)])
compile seed d = evalState (compileF ([],d)) seed
```

```
  -- an interface to the compiler
scompile :: (Compilable a,Pretty (Ann a)) ⇒
  a → String
scompile = show.pretty.toFunc.(compile 0)
```

```
  -- an interface to the compiler
icompile :: (Compilable a,Pretty (Ann a)) ⇒
  a → IO ()
icompile = putStrLn.scompile
```

Module Feldspar.Compiler.Compiler

This module, contains the main code for compiling the high-level AST to C code.

```
 {-# LANGUAGE GADTs #-}
 {-# LANGUAGE TypeSynonymInstances #-}
 {-# LANGUAGE FlexibleInstances #-}
 {-# LANGUAGE FlexibleContexts #-}
module Feldspar.Compiler.Compiler where
import qualified Prelude as P
import Prelude ((.),Show (..),putStrLn,IO
  ,Int,String,(++),(+),Monad (..))
import Control.Monad.State (State,put,get
  ,evalState)
```

```
import Feldspar.Types
import Feldspar.FrontEnd.AST
import Feldspar.BackEnd.AST
import Feldspar.BackEnd.Pretty
```

```
import Feldspar.Annotations
```

```
  -- the monadic function to compile the
  -- the high-level AST to a pair containing
  -- an expression containing the returned
  -- value and a list of statements; the
  -- state contains a counter to generate
  -- fresh variables

compileM :: SingT a ⇒ Data a ann →
  State Int (ExpC ann,[Stmt ann])
compileM (Var (VarT v _)) =
  return (VarC v,[])

compileM (LitInt x) =
  return (Num x,[])

compileM (LitBool P.True) =
  return (VarC "true",[])

compileM (LitBool P.False) =
  return (VarC "false",[])

compileM (Add e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "+" eC2
    ,st1 ++ st2)

compileM (Sub e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "-" eC2
    ,st1 ++ st2)

compileM (Mul e1 e2) = do
  (eC1,st1) ← compileM e1
  (eC2,st2) ← compileM e2
  return (Infix eC1 "*" eC2
    ,st1 ++ st2)
```

```
compileM ($Eq_{Int}$ $e_1$ $e_2$) = do
  ($e_{C1}$,$st_1$) ← compileM $e_1$
  ($e_{C2}$,$st_2$) ← compileM $e_2$
  return (Infix $e_{C1}$ "==" $e_{C2}$
    ,$st_1$ ++ $st_2$)


compileM ($LT_{Int}$ $e_1$ $e_2$) = do
  ($e_{C1}$,$st_1$) ← compileM $e_1$
  ($e_{C2}$,$st_2$) ← compileM $e_2$
  return (Infix $e_{C1}$ "<" $e_{C2}$
    ,$st_1$ ++ $st_2$)


compileM (And $e_1$ $e_2$) = do
  ($e_{C1}$,$st_1$) ← compileM $e_1$
  ($e_{C2}$,$st_2$) ← compileM $e_2$
  return (Infix $e_{C1}$ "&&" $e_{C2}$
    ,$st_1$ ++ $st_2$)


compileM (Not $e_1$) = do
  ($e_{C1}$,$st_1$) ← compileM $e_1$
  return (Unary "!" $e_{C1}$
    ,$st_1$)


compileM e@(If $e_1$ $e_2$ $e_3$) = do
  i ← get
  put (i + 1)
  let v = "v" ++ (show i)
  ($e_{C1}$,$st_1$) ← compileM $e_1$
  ($e_{C2}$,$st_2$) ← compileM $e_2$
  ($e_{C3}$,$st_3$) ← compileM $e_3$
  return
    ($Var_C$ v
    ,$st_1$ ++
      [Declare (getType e) v
      ,$If_C$ $e_{C1}$
        ($st_2$ ++ [Assign v $e_{C2}$])
        ($st_3$ ++ [Assign v $e_{C3}$])])


compileM e = preserve e compileM
```

```
  -- overloaded function to compile
  -- regardless of AST being parametric
class Inj t ⇒
  Compilable t where
  compileF :: ([Var],t) →
    State Int
    ([Var],Types
    ,Exp_C (Ann t)
    ,[Stmt (Ann t)])

  -- a parametric AST is first applied to
  -- a fresh variable with the right type
  -- and then it is compiled
instance (SingT a,Compilable r) ⇒
      Compilable (Data a ann' → r) where
  compileF (ps,f) = do
    i ← get
    put (i + 1)
    let v = "v" ++ (show i)
      a = Var (VarT v (getTypeF f))
      r = f a
    compileF ((ps ++ [(v,getType a)]),r)

  -- a non-parametric AST is compiled in
  -- the normal way defined in compileM
instance SingT a ⇒
  Compilable (Data a ann) where
  compileF (ps,d) = do
    (e,sts) ← compileM d
    return (ps,getType d,e,sts)

  -- coversion to Func
toFunc :: ([Var],Types,Exp_C ann,[Stmt ann]) →
  Func ann
toFunc (ps,ty,exp_C,stmts) =
  Func "test" (ps ++ [("*out",ty)])
  (stmts ++ [Assign "*out" exp_C])
  -- running the state monad with a seed
compile :: Compilable a ⇒ Int →
  a → ([Var], Types, Exp_C (Ann a)
    , [Stmt (Ann a)])
compile seed d = evalState (compileF ([], d)) seed
```

```
  -- an interface to the compiler
scompile :: (Compilable a, Pretty (Ann a)) ⇒
  a → String
scompile = show.pretty.toFunc.(compile 0)
```

```
  -- an interface to the compiler
icompile :: (Compilable a, Pretty (Ann a)) ⇒
  a → IO ()
icompile = putStrLn.scompile
```

Module Feldspar.Compiler.BXCompiler

This module contains the code to bidirectionalize the compile functions.

```
 {-# LANGUAGE FlexibleContexts #-}
 {-# LANGUAGE FlexibleInstances #-}
 {-# LANGUAGE GADTs #-}
module Feldspar.Compiler.BXCompiler where
import qualified Prelude as P
import Prelude (String, Eq (..), Either (..), Int, const
  , Monad (..), (.), tail, Show (..), (+)
  , (++), ($))
```

```
import BX
import Annotations
import Feldspar.Types
import Feldspar.BackEnd.AST
import Feldspar.FrontEnd.AST
import Feldspar.Compiler.Compiler
import Feldspar.FrontEnd.Derivings ()
import Feldspar.BackEnd.Derivings ()
```

```
  -- overloaded function to bidirectionalize
  -- instances of Compilable
class Compilable t ⇒ BXable t where
  putCompile :: Eq (Ann t) ⇒ Int →
    t → Func (Ann t) →
    Either String t
```

```
  -- Bidirectionalization done by bff_GUS_G_Gen
instance SingT a ⇒ BXable (Data a ann) where
  putCompile i = $bff^{a/d-*}_{GUS}$
    (const (toFunc.compile i))
    (const () :: ∀ ann.()$^1$ ann → ())
```

```
  -- Bidirectionalization done manually
instance (SingT a,BXable r
           ,Ann r~ann,Abstract r) ⇒
           BXable (Data a ann → r) where
  putCompile i f (Func x ps stmts) = do
    let n = "v" ++ (show i)
    let vt = VarT n (getTypeF f)
    let v = (Var vt)
    let r = f v
    r′ ← putCompile (i + 1) r (Func x (tail ps) stmts)
    return $ λvv → abstract vt vv r′
```

```
  -- overloaded function to abstract over
  -- a variable and generate the parametric AST
class Abstract t where
  abstract :: ∀ a. VarT a →
    Data a (Ann t) → t → t
```

```
instance Abstract r ⇒
  Abstract (Data a ann → r) where
  abstract vt d f = abstract vt d.f
```

```
instance Abstract (Data a ann) where
  abstract (VarT v SBool) d
    e@(Var (VarT x SBool))
     | v == x = d
     | P.True = e
  abstract (VarT v SInt32) d
    e@(Var (VarT x SInt32))
     | v == x = d
     | P.True = e
  abstract _ _ e@(Var _) = e
```

```
abstract _ _ (Lit_Int i) =
  Lit_Int i
```

```
abstract _ _ (Lit_Bool b) =
  Lit_Bool b
```

```
abstract v d (Not e) =
  Not (abstract v d e)
```

```
abstract v d (Ann a e) =
  Ann a (abstract v d e)
```

```
abstract v d (Add e1 e2) =
  Add (abstract v d e1)
      (abstract v d e2)
```

```
abstract v d (Sub e1 e2) =
  Sub (abstract v d e1)
      (abstract v d e2)
```

```
abstract v d (Mul e1 e2) =
  Mul (abstract v d e1)
      (abstract v d e2)
```

```
abstract v d (Eq_Int e1 e2) =
  Eq_Int (abstract v d e1)
         (abstract v d e2)
```

```
abstract v d (LT_Int e1 e2) =
  LT_Int (abstract v d e1)
         (abstract v d e2)
```

```
abstract v d (And e1 e2) =
  And (abstract v d e1)
      (abstract v d e2)
```

```
abstract v d (If e1 e2 e3) =
  If (abstract v d e1)
     (abstract v d e2)
     (abstract v d e3)
```

Module Feldspar.BackEnd.AST

This module contains the declaration of the AST of the low-level language (*C*).

```
module Feldspar.BackEnd.AST where
import qualified Prelude
import Prelude (Int,String)
import Feldspar.Types


  -- variables
type Var = (String,Types)


  -- C function
data Func ann =
  Func String [Var] [Stmt ann]


  -- C statement
data Stmt ann =
  If_C (Exp_C ann) [Stmt ann] [Stmt ann]
  | Assign String (Exp_C ann)
  | Declare Types String
  | Ann_Stmt ann (Stmt ann)
```

```
  -- C expressions
data Exp_C ann =
    Var_C String
    | Num Int
    | Infix (Exp_C ann) String (Exp_C ann)
    | Unary String (Exp_C ann)
```

```
  | Ann_ExpC ann (Exp_C ann)
```

Module Feldspar.BackEnd.Pretty

This module contains the code for pretty-printing the low-level AST. It uses John Hughes's and Simon Peyton Jones's Pretty Printer Combinators [Hug95].

```
 {-# LANGUAGE FlexibleInstances #-}
module Feldspar.BackEnd.Pretty where
import qualified Prelude
import Prelude (($),map,foldl1)
import Text.PrettyPrint (Doc,text,int,parens,semi,space
   ,comma,lbrace,rbrace,vcat,nest
   ,($+$),($$),(<>),(<+>))
import qualified Data.List
import Feldspar.BackEnd.AST
import Feldspar.Types

class Pretty a where
   pretty :: a → Doc

instance Pretty ann ⇒
   Pretty (Exp_C ann) where
   pretty (Var_C x) = text x
   pretty (Num i) = int i
   pretty (Infix e1 op e2) = parens (pretty e1
                              <+> text op
                              <+> pretty e2)
   pretty (Unary op e) = parens (text op
                              <+> pretty e)
```

```
pretty (Ann_ExpC ann e) = text "/*"
                           <+> (pretty ann) <+>
                          text "*/"
                           <+> pretty e

instance Pretty ann ⇒
  Pretty (Stmt ann) where

pretty (If_C e1 e2 e3) = text "if"
  <+> parens (pretty e1)
  $+$ lbrace
  $+$ nest 2 (vcat (map pretty e2))
  $+$ rbrace
  $+$ text "else"
  $+$ lbrace
  $+$ nest 2 (vcat (map pretty e3))
  $+$ rbrace

pretty (Assign v e) = text v <+> text "="
                      <+> pretty e <> semi

pretty (Declare t v) = pretty t <+> text v <> semi

pretty (Ann_Stmt ann st) = text "/*"
                            <+> (pretty ann) <+>
                           text "*/"
                            $$ pretty st

instance Pretty ann ⇒
  Pretty (Func ann) where

pretty (Func name vs body) =
  text "#include \"feldspar.h\""
  $+$ text "void" <+> text name
  <+> parens (commaCat (map pretty vs))
  $+$ lbrace
  $+$ nest 2 (vcat (map pretty body))
  $+$ rbrace
```

```
instance Pretty Var where
  pretty (v,t) = pretty t <+> text v

instance Pretty Types where
  pretty Int32 = text "int32_t"
  pretty Bool = text "uint32_t"

commaCat :: [Doc] → Doc
commaCat ds = foldl1 (<>) $
  Data.List.intersperse (comma <> space) ds
```

Module Feldspar.BackEnd.Derivings

In this module, the type classes *Eq*,*Functor*,*Foldable* and *Traversable* is derived for the AST of the low-level language.

```
-- the code is omitted
```

Module Feldspar.BackEnd.BXPretty

This module contains the code needed to bidirectionalize the pretty-printing transformation.

```
{-# LANGUAGE Rank2Types #-}
module Feldspar.BackEnd.BXPretty where
import qualified Prelude
import Prelude (Eq (..),Show (..),(.),Int,id,String
  ,Bool (..),Functor (..),Read (..),Monad (..),Maybe (..)
  ,Either (..),map,filter,($),fst,¬,splitAt,read
  ,tail,(+),length,(∧))
```

```
import Text.PrettyPrint (Doc,int,text)
import Control.Monad (unless)
import Data.List (isPrefixOf,stripPrefix)
import Data.Foldable (toList)
import Data.Traversable (Traversable)
import Data.Function (on)
import BX (fromJust,fromList,index,assoc,validAssoc
  ,union,lookupAll)
import Feldspar.BackEnd.Pretty (Pretty (..))
```

```
  -- lexical tokens
data Token =
      Ann String
         -- the annotations (comments)
    | Etc String
         -- anything except annotations
  deriving Show
```

```
  -- tokens are compared ignoring space
  -- and new-line characters
instance Eq Token where
  (Ann s) == (Ann s') = ((==) `on`
     (filter (λx → (x/ = '\n') ∧
        (x/ = ' ')))) s s'
  (Etc s) == (Etc s') = ((==) `on`
     (filter (λx → (x/ = '\n') ∧
        (x/ = ' ')))) s s'
  _ == _ = False
```

```
  -- checking if a token is an annotation
isAnn :: Token → Bool
isAnn (Ann _) = True
isAnn _ = False
```

```
  -- lexical tokenizer
tokenize :: String → Maybe [Token]
tokenize [] = Just []
tokenize ('/' : '*' : ' ' : xs) = do
  (before,after) ← splitBy " */" xs
  ts             ← tokenize after
  return $ (Ann before) : ts
tokenize (x : xs) = do
  ts ← tokenize xs
  return $ case ts of
     []              → Etc [x]     : ts
     (Ann _) : _     → Etc [x]     : ts
     (Etc y) : ts'   → Etc (x : y) : ts'
  -- finding index of a string inside another string
infixAt :: Eq a ⇒ [a] → [a] → Maybe Int
infixAt needle haystack = infixAt′ 0 needle haystack
  where
    infixAt′ _ _ []          = Nothing
    infixAt′ i n hs
       | n ‘isPrefixOf‘ hs = Just i
       | True              = infixAt′ (i + 1) n (tail hs)
```

```
  -- spliting a string by the given key string
splitBy :: Eq a ⇒ [a] → [a] → Maybe ([a],[a])
splitBy infx s = do
  i ← infixAt infx s
  let (before,wafter) = splitAt i s
  after ← stripPrefix infx wafter
  return (before,after)
```

```
  -- The format of the output string of
  -- pretty printing us to extract the annotations
  -- by 1.tokenizing the string 2.extracting the
  -- the comments 3.parsing the strings to the
  -- actual annotation values, hence the Read
  -- constraint
toList_Doc :: ∀ a.Read a ⇒ String → [a]
toList_Doc d = [read s | Ann s ← fromJust $ tokenize d]
```

```
  -- the shape of two output strings are
  -- compared by ignoring the values in the
  -- comments
eqShape_Doc :: String → String → Bool
eqShape_Doc = (==) ‘on‘
  (fmap (filter (¬.isAnn))
  .tokenize)
```

```
  -- since pretty printing uses type classes for
  -- overloading, we are no longer able to use
  -- our generic function (bff); we have to change
  -- the code slightly (as highlighted)
```

```
$bx_{PP} :: \forall\ k.(Traversable\ k, Pretty\ (k\ Doc)) \Rightarrow$
  $(\forall\ t.Pretty\ t \Rightarrow$
    $k\ t \rightarrow String) \rightarrow$
  $(\forall\ a.(Read\ a, Eq\ a, Pretty\ a) \Rightarrow$
    $k\ a \rightarrow String \rightarrow$
    $Either\ String\ (k\ a))$
$bx_{PP}\ get\ s\ v = \mathbf{do}$
  $\mathbf{let}\ s^{list}\ = toList\ s$
  $\mathbf{let}\ v^{list}\ = toList_{Doc}\ v$
  $\mathbf{let}\ get^{list}_{By}\ :: \forall\ a.(Read\ a, Pretty\ a) \Rightarrow$
    $[a] \rightarrow [a]$
    $get^{list}_{By}\ x = toList_{Doc}\ \$$
      $get\ (fromList\ s\ x)$
  $unless\ eqShape\_Doc\ (get\ s)\ v$
    $\$\ Left$ "Modified view of wrong shape!"
  $\acute{s}^{list} \leftarrow bff\_Pretty\ get^{list}_{By}\ s^{list}\ v^{list}$
  $return\ \$\ fromList\ s\ \acute{s}^{list}$
```

```
  -- the version of bff working with lists and
  -- pretty printing constraint; it does not
  -- check for validity of the mappings since
  -- the type Doc is abstract and the exposed
  -- observer functions by the module, namely
  -- the functions show and render are not
  -- used in our pretty printer
$bff\_Pretty :: (\forall\ a.(Read\ a, Pretty\ a) \Rightarrow$
    $[a] \rightarrow [a]) \rightarrow$
  $(\forall\ a.(Eq\ a, Pretty\ a) \Rightarrow$
    $[a] \rightarrow [a] \rightarrow Either\ String\ [a])$
$bff\_Pretty\ get\ s\ v = \mathbf{do}$
    -- Step 1
  $\mathbf{let}\ ms = index\ s$
    -- Step 2
  $\mathbf{let}\ is\ \ = fst\ `map`\ ms$
  $\mathbf{let}\ iv\ \ = get\ is$
    -- Step 3
  $unless\ (length\ v == length\ iv)$
    $\$\ Left$ "Modified view of wrong length!"
  $\mathbf{let}\ mv = assoc\ iv\ v$
  -- Step 4
unless (validAssoc mv)
   $ Left "Inconsistent duplicated values!"
  -- Step 5
let ms′ = union mv ms
  -- Step 5.1
  -- check is removed
  -- Step 6
return $ lookupAll is ms′
```

```
  -- the put (backward) function that
  -- bidirectionalizes the pretty printer
putPretty :: ∀ k a.
  (Eq a,Read a,Traversable k
  ,Pretty (k Doc),Pretty a) ⇒
  k a → String → Either String (k a)
putPretty = bx_PP (show.pretty.(fmap pretty))
```

```
instance Pretty Doc where
  pretty = id
instance Pretty Int  where
  pretty = int
instance Pretty Bool where
  pretty = text.show
```

Module Examples.TestFeldspar

This module contains an example program written in the high-level language.

```
{-#OPTIONS_GHC -F -pgmF qapp#-}
```

```
module Examples.TestFeldspar where
import qualified Prelude
import Feldspar
import Feldspar.Compiler
```

```
inc :: Data Int32 → Data Int32
inc x = x + 1
```

```
dec :: Data Int32 → Data Int32
dec x = x − 1

incAbs :: Data Int32 → Data Int32
incAbs a = condition (a < 0) (dec a) (inc a)

cCode :: IO ()
cCode = icompile incAbs
```

C Code Examples.TestFeldspar

Listing 1: Pico-Feldspar/Examples/TestFeldspar.c

```
#include "feldspar.h"
void test (int32_t v0, int32_t *out)
{
  /* False */
  int32_t v1;
  /* False */
  if (/* False */ (/* False */ v0 < /* False */ 0))
  {
    v1 = /* False */ (/* False */ v0 - /* False */ 1);
  }
  else
  {
    v1 = /* True */ (/* False */ v0 + /* False */ 1);
  }
  *out = /* False */ v1;
}
```

Module Examples.BXTestFeldspar

```
module Example.BXTestFeldspar where
import Feldspar.AnnotationUtils (markAllF,markAll)
import Feldspar.BX              (putAnn)
import Examples.TestFeldspar (incAbs)
import Feldspar.Compiler.Compiler (scompile)

  -- the location of the generated C code
c :: String
c = "Examples/TestFeldspar.c"
```

```
  -- forward transformation from EDSL to C
forward :: IO ()
forward = writeFile c
  (scompile ((markAllF markAll)
    incAbs))


  -- backward transformation from C to src-loc
backward :: IO ()
backward = do
  cSrc ← readFile c
  let r = putAnn False (markAllF markAll) incAbs cSrc
  case r of
    Right locs → putStrLn $ show locs
    Left er → putStrLn er
```

Appendix C

Prelude Polymorphic Functions – Accepted by Our Algorithm (Part I)

$\mathit{Just} :: a \rightarrow \mathit{Maybe}\ a$

$\mathit{Left} :: a \rightarrow \mathit{Either}\ a\ b$

$\mathit{Right} :: b \rightarrow \mathit{Either}\ a\ b$

$\mathit{fst} :: (a,b) \rightarrow a$

$\mathit{snd} :: (a,b) \rightarrow b$

$\mathit{id} :: a \rightarrow a$

$\mathit{const} :: a \rightarrow b \rightarrow a$

$\mathit{asTypeOf} :: a \rightarrow a \rightarrow a$

$\mathit{seq} :: a \rightarrow b \rightarrow b$

$(+\!\!+) :: [a] \rightarrow [a] \rightarrow [a]$

$\mathit{head} :: [a] \rightarrow a$

$\mathit{last} :: [a] \rightarrow a$

$\mathit{tail} :: [a] \rightarrow [a]$

$\mathit{init} :: [a] \rightarrow [a]$

$(!!) :: [a] \rightarrow \mathit{Int} \rightarrow a$

$\mathit{reverse} :: [a] \rightarrow [a]$

$\mathit{concat} :: [[a]] \rightarrow [a]$

$\mathit{replicate} :: \mathit{Int} \rightarrow a \rightarrow [a]$

$\mathit{cycle} :: [a] \rightarrow [a]$

$\mathit{take} :: \mathit{Int} \rightarrow [a] \rightarrow [a]$

Prelude Polymorphic Functions – Accepted by Our Algorithm (Part II)

$drop :: Int \rightarrow [a] \rightarrow [a]$

$splitAt :: Int \rightarrow [a] \rightarrow ([a],[a])$

$repeat :: a \rightarrow [a]$

$lookup :: Eq\ a \Rightarrow a \rightarrow [(a,b)] \rightarrow Maybe\ b$

$maximum :: Ord\ a \Rightarrow [a] \rightarrow a$

$minimum :: Ord\ a \Rightarrow [a] \rightarrow a$

$filter :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$

$takeWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$

$dropWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$

$dropWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$

$break :: (a \rightarrow Bool) \rightarrow [a] \rightarrow ([a],[a])$

$span :: (a \rightarrow Bool) \rightarrow [a] \rightarrow ([a],[a])$

$zip :: [a] \rightarrow [b] \rightarrow [(a,b)]$

$zip3 :: [a] \rightarrow [b] \rightarrow [c] \rightarrow [(a,b,c)]$

$unzip :: [(a,b)] \rightarrow ([a],[b])$

$unzip3 :: [(a,b,c)] \rightarrow ([a],[b],[c])$

Prelude Polymorphic Functions – Polymorphic Output

$error :: [Char] \rightarrow a$

$ioError :: IOError \rightarrow IO\ a$

Prelude Polymorphic Functions – Polymorphic Input

$print :: Show\ a \Rightarrow a \rightarrow IO\ ()$

$even :: Integral\ a \Rightarrow a \rightarrow Bool$

$odd :: Integral\ a \Rightarrow a \rightarrow Bool$

$any :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$

$all :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$

$elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$

$notElem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$

$shows :: Show\ a \Rightarrow a \rightarrow ShowS$

$print :: Show\ a \Rightarrow a \rightarrow IO\ ()$

$null :: [a] \rightarrow Bool$

$length :: [a] \rightarrow Int$

Prelude Polymorphic Functions – Higher Kinded

$mapM :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b]$

$mapM\_ :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ ()$

$sequence :: Monad\ m \Rightarrow [m\ a] \rightarrow m\ [a]$

$sequence\_ :: Monad\ m \Rightarrow [m\ a] \rightarrow m\ ()$

$(=\!\!\ll) :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow m\ a \rightarrow m\ b$

Prelude Polymorphic Functions – Not Accepted by Our Algorithm

$\mathit{maybe} :: b \rightarrow (a \rightarrow b) \rightarrow \mathit{Maybe}\ a \rightarrow b$

$\mathit{either} :: (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow \mathit{Either}\ a\ b \rightarrow c$

$\mathit{curry} :: ((a,b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$

$\mathit{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a,b) \rightarrow c$

$\mathit{subtract} :: \mathit{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$

$\mathit{gcd} :: \mathit{Integral}\ a \Rightarrow a \rightarrow a \rightarrow a$

$\mathit{lcm} :: \mathit{Integral}\ a \Rightarrow a \rightarrow a \rightarrow a$

$(\uparrow) :: (\mathit{Num}\ a, \mathit{Integral}\ b) \Rightarrow a \rightarrow b \rightarrow a$

$(\uparrow\uparrow) :: (\mathit{Fractional}\ a, \mathit{Integral}\ b) \Rightarrow a \rightarrow b \rightarrow a$

$\mathit{fromIntegral} :: (\mathit{Integral}\ a, \mathit{Num}\ b) \Rightarrow a \rightarrow b$

$\mathit{realToFrac} :: (\mathit{Real}\ a, \mathit{Fractional}\ b) \Rightarrow a \rightarrow b$

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

$\mathit{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$

$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$

$\mathit{until} :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a$

$(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$

$\mathit{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

$\mathit{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

$\mathit{foldl1} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$

$\mathit{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

$\mathit{foldr1} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$

$\mathit{sum} :: \mathit{Num}\ a \Rightarrow [a] \rightarrow a$

$\mathit{product} :: \mathit{Num}\ a \Rightarrow [a] \rightarrow a$

$\mathit{concatMap} :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b]$

$\mathit{scanl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow [a]$

$\mathit{scanl1} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]$

$\mathit{scanr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow [b]$

$\mathit{scanr1} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]$

$\mathit{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a]$

$\mathit{zipWith} :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$

$\mathit{zipWith3} :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow [a] \rightarrow [b] \rightarrow [c] \rightarrow [d]$

$\mathit{reads} :: \mathit{Read}\ a \Rightarrow \mathit{ReadS}\ a$

$\mathit{readParen} :: \mathit{Bool} \rightarrow \mathit{ReadS}\ a \rightarrow \mathit{ReadS}\ a$

$\mathit{read} :: \mathit{Read}\ a \Rightarrow \mathit{String} \rightarrow a$

$\mathit{readIO} :: \mathit{Read}\ a \Rightarrow \mathit{String} \rightarrow \mathit{IO}\ a$

$\mathit{readLn} :: \mathit{Read}\ a \Rightarrow \mathit{IO}\ a$

Prelude Monomorphic Functions

$putChar :: Char \to IO\ ()$

$putStr :: String \to IO\ ()$

$putStrLn :: String \to IO\ ()$

$interact :: (String \to String) \to IO\ ()$

$readFile :: FilePath \to IO\ String$

$writeFile :: FilePath \to String \to IO\ ()$

$appendFile :: FilePath \to String \to IO\ ()$

$(\wedge) :: Bool \to Bool \to Bool$

$(|) :: Bool \to Bool \to Bool$

$\neg :: Bool \to Bool$

$and :: [Bool] \to Bool$

$or :: [Bool] \to Bool$

$lines :: String \to [String]$

$words :: String \to [String]$

$unlines :: [String] \to String$

$unwords :: [String] \to String$

$showChar :: Char \to ShowS$

$showString :: String \to ShowS$

$showParen :: Bool \to ShowS \to ShowS$

$lex :: ReadS\ String$

$userError :: String \to IOError$

Prelude Constant Functions

$False :: Bool$

$True :: Bool$

$otherwise :: Bool$

$LT :: Ordering$

$EQ :: Ordering$

$GT :: Ordering$

$getChar :: IO\ Char$

$getLine :: IO\ String$

$getContents :: IO\ String$

$\perp :: a$

$Nothing :: Maybe\ a$

Type Classes Accepted by Our Algorithm

Eq

Ord

Show

Type Classes Not Accepted by Our Algorithm

Enum

Bounded

Num

Real

Integral

Fractional

Floating

RealFrac

RealFloat

Read

Monad

Functor

Appendix D

fromColoredtoBW.c

```
 #include "fromColoredtoBW.h"
 #include "feldspar_c99.h"
 #include "feldspar_array.h"
 #include "feldspar_future.h"
 #include "ivar.h"
 #include "taskpool.h"
 #include <stdint.h>
 #include <string.h>
 #include <math.h>
 #include <stdbool.h>
 #include <complex.h>


void fromColoredtoBW(struct array * v0, struct array * out)
{
    struct array v12 = {0};
    uint32_t len0;
    struct array v2 = {0};
    uint32_t len2;

    /* SrcLoc { srcFilename = "IP.hs"
             , srcLine = 36, srcColumn = 1}
               (vector length) */
    len0 = (getLength(v0) / 3);
    initArray(&v12, sizeof(int32_t), 0);
    for(uint32_t v1 = 0; v1 < len0; v1 += 1)
    {
```

```
    int32_t v11;
    uint32_t v10;
    struct array e1 = {0};

    /* SrcLoc {srcFilename = "IP.hs",
               srcLine = 29, srcColumn = 1} */
    v10 = (v1 * 3);
    /* SrcLoc {srcFilename = "IP.hs",
               srcLine = 16, srcColumn = 1} */
    /* SrcLoc {srcFilename = "IP.hs",
               srcLine = 20, srcColumn = 1} */
    /* SrcLoc {srcFilename = "IP.hs",
               srcLine = 24, srcColumn = 1} */
    v11 = ((int32_t)(truncf((((
              ((float)(at(int32_t,v0,v10))) *
               0.30000001192092896f) +
              (((float)(at(int32_t,v0,(v10 + 1)))) *
               0.5899999737739563f)) +
              (((float)(at(int32_t,v0,(v10 + 2)))) *
               0.10999999940395355f)))));
    initArray(&e1, sizeof(int32_t), 1);
    for(uint32_t v4 = 0; v4 < 1; v4 += 1)
    {
        /* SrcLoc {srcFilename = "IP.hs",
                   srcLine = 29, srcColumn = 1} */
        at(int32_t,&e1,v4) = v11;
    }
    initArray(&v2, sizeof(int32_t),
              (getLength(&v12) + 1));
    copyArray(&v2, &v12);
    copyArrayPos(&v2, getLength(&v12), &e1);
    initArray(&v12, sizeof(int32_t),
               getLength(&v2));
    copyArray(&v12, &v2);
}
/* SrcLoc {srcFilename = "IP.hs",
           srcLine = 36, srcColumn = 1}
           (vector length) */
len2 = getLength(&v12);
initArray(out, sizeof(int32_t), len2);
for(uint32_t v5 = 0; v5 < len2; v5 += 1)
{
    /* SrcLoc {srcFilename = "IP.hs",
                  srcLine = 11, srcColumn = 1}
                  (vector element) */
        /* SrcLoc {srcFilename = "IP.hs",
                  srcLine = 36, srcColumn = 1}
                  (vector element) */
        /* SrcLoc {srcFilename = "IP.hs",
                  srcLine = 36, srcColumn = 1}
                 (vector length) */
        if((at(int32_t,&v12,v5) < 127))
        {
            at(int32_t,out,v5) = 1;
        }
        else
        {
            at(int32_t,out,v5) = 0;
        }
    }
    freeArray(&v12);
    freeArray(&v2);
}
```