Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III

Diploma Thesis

Formalizing Semantic Bidirectionalization in Agda

Helmut Grohne
helmut@subdivi.de

September 30, 2013

Supervisor: Jun.-Prof. Dr. rer. nat. habil. Janis Voigtländer

I declare that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from these sources are clearly marked as such.

Helmut Grohne

Contents

1 Introduction

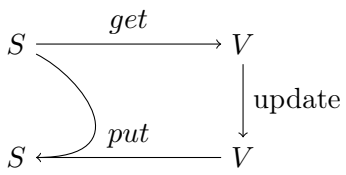
Voigtländer (2009) presented a semantic approach to bidirectionalization encompassing an implementation in Haskell and paper proofs of its correctness. We want study the process of formally expressing both the implementation and the correctness proofs in a dependently-typed programming language.

To understand bidirectionalization, we look at the following definition and provide an intuition afterwards.

Definition (lens laws) Let S, V be sets. A pair of partial functions $get\colon S \to V$ and $put\colon S \times V \to S$ is called *bidirectional transformation* or *lens* if the following two properties hold whenever the applications are defined.

$$\forall s \in S.\quad put(s, get(s)) = s \tag{GetPut}$$
$$\forall s \in S, v \in V.\quad get(put(s, v)) = v \tag{PutGet}$$

In this context, the bidirectionalization task is to find a useful function *put* for a given function *get* such that the lens laws are satisfied. The problem originates in a database setting and was formulated by Bancilhon and Spyratos (1981). There S describes the possible states of a database and *get* is a query to compute a view or partial representation of the database relevant to a particular application. The view is externally changed and *put* describes the process to record the update in the database. This process can be visualized using the following diagram.



To illustrate the task, let us consider an example outside the database context where S contains triples of numbers. An example *get* function maps (a, b, c) to $a + b$. Two ways to to turn it to a bidirectional transformation are the following functions.

$$put_1((a, b, c), d) = (d - b, b, c)$$
$$put_2((a, b, c), d) = (a, d - a, c)$$

It is not clear which of these is better and in general choosing the right bidirectional transformation is application specific. On the other hand, the following two functions violate in general the GetPut law.

$$put_3((a, b, c), d) = (d - b, b, 0)$$
$$put_4((a, b, c), d) = (d, 0, c)$$

The law requires that if no change is applied to a view before handing it to *put*, the original element of S should be retained, but put_3 changes the third element of a triple and put_4 changes the other elements when $d = a + b$. Similarly, the next function violates in general the PutGet law.

$$put_5((a, b, c), d) = (a, b, c)$$

The function fails to record the update by completely ignoring the view element. A valid but useless bidirectional transformation is given by the last example for *put*.

$$put_6((a, b, c), d) = \begin{cases} (a, b, c) & \text{if } d = a + b, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

It satisfies both lens laws by not being able to retain any update. Our attempt at a bidirectionalization of our example *get* has been manual, but in general it should be automatic. Ideally the need to write down a *put* function should be avoided altogether. Instead, it should be computable from *get*.

A syntactic approach to bidirectionalization was presented by Foster et al. (2007). The idea is to constrain the language used to express *get* functions and to combine bidirectionalizations which are explicitly given for individual language primitives. Then *put* can be computed from the definition of *get*.

The semantic approach we are going to examine was presented by Voigtländer. Here the behavior of *get* is restricted using the type system of the functional programming language Haskell. The type chosen to start with is `[a] -> [a]`. It describes polymorphic *get* functions that transform homogeneous lists. In particular, such functions cannot inspect or create elements of the content type `a`. Their behavior can be exhaustively captured by positional descriptions. So the implementation given for *put* observes the behavior of a *get* function on example inputs, such as lists of integers.

Being able to predict the behavior of functions based on past invocations is possible, because Haskell functions behave like mathematical functions. They are side effect free. This aspect facilitated proving the lens laws for the given implementation of *put* in Haskell. The implementation will be reproduced in Section 1.1. Starting with that section, Haskell knowledge is assumed.

Even though Haskell was useful for expressing *put*, it cannot be used to formalize the proofs of the lens laws. For this purpose, we want to use Agda which is a descendant of Haskell. It is implemented in and syntactically similar to Haskell. Both languages are based on typed λ-calculi. The one in Agda is extended to allow values to occur as parameters to types. We say that it supports dependent types. This mixing of types and values enables us to encode properties into types. Thus the type checker is able to verify the correctness of proofs. An introduction to the language is given in Section 2.

The main part of this thesis is to investigate how Agda can be applied to a real-world scientific result. We will translate the implementation of *put* given by Voigtländer to Agda and redevelop the proofs of the lens laws in parallel. Much of the work will go into proving assertions that were previously assumed obvious, but need to be explained to

Agda. Thus the result will gain in precision. In the process, we will explain how proofs are developed and what assistance is provided by Agda.

Note that Voigtländer shows a bit more than the lens laws given above in terms of definedness. Assuming that *get* is fully defined, the application of *put* used in GetPut is shown to be defined. Since arbitrary applications of *put* as used in PutGet are not necessarily defined, we give a sufficient precondition for successfully invoking *put* in Section 6.

1.1 Bidirectionalization in Haskell

In this section, we will briefly examine the bidirectionalization method presented in Voigtländer (2009). Apart from basic Haskell constructs, it uses the `Data.IntMap` library in order to associate numbers with values.

```
fromAscList :: [(Int, a)] -> IntMap a
empty       :: IntMap a
insert      :: Int -> a -> IntMap a -> IntMap a
union       :: IntMap a -> IntMap a -> IntMap a
lookup      :: Int -> IntMap a -> Maybe a
```

`IntMap.fromAscList` turns a sequence of pairs into an `IntMap`. The integer keys are expected to ascend. `IntMap.insert` extends an existing `IntMap` with a new key value pair and possibly overwrites an existing association. `IntMap.union` is left-biased.

With this container in place, we can define a more careful variant of `IntMap.insert`. It fails if the pair being inserted violates an existing association.

```
checkInsert :: Eq a => Int -> a -> IntMap a -> Maybe (IntMap a)
checkInsert i b m = case IntMap.lookup i m of
                      Nothing -> Just (IntMap.insert i b m)
                      Just c  -> if b == c then Just m
                                           else Nothing
```

The definition deviates from the original in using `Maybe` in place of `Either`. The `Left` constructor was used in place of `Nothing` to transport an error message. We are dropping error messages here, because we are only interested in verifying the lens laws.

With `checkInsert`, we can construct an `IntMap` from two lists by associating a number from the first list with the value at the same index in the second list. The construction fails when the lists have different lengths or when there are conflicting associations.

```
assoc :: Eq a => [Int] -> [a] -> Maybe (IntMap a)
assoc []       []       = Just IntMap.empty
assoc (i : is) (b : bs) = assoc is bs >>= checkInsert i b
assoc _        _        = Nothing
```

Again we changed the definition to use `Maybe` instead of `Either` with the same reasoning.

With these tools in place, we can proceed to examine the actual bidirectionalization method. It is called `bff` which is a short form of "bidirectionalization for free".

```
bff :: (forall a. [a] -> [a])
       -> (forall a. Eq a => [a] -> [a] -> [a])
```

The first parameter is a `get` function. In order to require it to be polymorphic, we need the `RankNTypes`[1] language extension.

```
bff get = \s v ->
  let s' = [0..length s - 1]
      g  = IntMap.fromAscList (zip s' s)
      h  = fromJust (assoc (get s') v)
      h' = IntMap.union h g
  in seq h (map (fromJust . flip IntMap.lookup h') s')
```

The symbol `s'` is bound to a template for the result of a `bff` application. Specifically, it determines the length of the resulting list to be equal to the length of the input `s`. Note that there are correct bidirectionalizations that violate this property[2], but this implementation does cover them. Since we need to be able to distinguish elements of the template, using numbers is a natural choice. A mapping `g` is constructed that maps elements of `s'` to the elements of `s` at corresponding indices. The `get` function is evaluated on `s'` and a mapping `h` is constructed to associate elements of `get s'` with corresponding elements of `v`. Note that if `v` and `get s'` have different lengths, the `bff` function errors out in the first `fromJust`. The left-biased union used to construct `h'` ensures that, whenever an element of `s'` shows up in `get s'`, the resulting association is preferred over the association of `s'` with `s`. To produce the result, the elements of `s'` are looked up in `h'`. Contrary to the first use of `fromJust`, the second one cannot fail, because all elements of `s'` are in the domain of `g` and thus of `h'`. Finally `seq` is used to propagate an error from the first `fromJust`, that otherwise may go unnoticed if no elements of the result are examined.

This version deviates from the original definition in using a different `assoc` function, that returns a `Maybe` when the original returned `Either`. Thus the error container is now eliminated with `fromJust` instead of `either`.

2 Agda

We will introduce the Agda language closely following Norell (2008). The most significant differences to Haskell are the introduction of dependent types and the requirement for all functions to be total. In order to show the benefits of machine verification, this document is written as a literate Agda file, so its source can be verified by Agda.

In a dependently typed language, the result type of a function may embody the concrete parameters passed to the function. This blurs the line between types and values, that one may be used to in Haskell. As a first example, let us have a look at the identity function. We will use a slightly simplified version of the definition from the standard library[3].

[1] Voigtländer used the now deprecated `Rank2Types` extension.

[2] Consider a `get` that maps `[a, b, c]` to `[a, a]` and behaves like the identity otherwise. Then the only correct result for `put [1, 2, 3] [1, 2]` is `[1, 2]`.

[3] The `id` function is available in the `Function` module. Further footnotes just mention the module name.

```
id : {α : Set} → α → α
id x = x
```

While the definition looks the same as in Haskell, the type declaration has changed. The availability of dependent types changes the way to express polymorphism. Instead of treating all lowercase variables as type variables, we say that α shall be an element of Set. The type Set contains all types that we will use, except for itself. Agda knows about a type that contains Set, but we are not interested in it and further types outside Set. Therefore, all citations from the standard library have their support for types beyond Set removed. Eliding those types allows us to give shorter type signatures.

The next notable difference in the type signature of id is the use of curly parentheses and the fact that it has two parameters instead of one. A parameter enclosed in curly parentheses is called *implicit*. When the function is defined or used, implicit parameters are not named or given. Instead, the type system is supposed to figure out the values of these parameters. In the case of the identity function, the type of the explicit parameter will be the value of the implicit parameter. It is possible to define functions for which the type system cannot determine the values of implicit parameters. A type error will be caused in the application of such a function.

For brevity, we can declare multiple consecutive parameters of the same type without repeating the type, as can be seen in the constant function as given in the standard library[4].

```
const : {α β : Set} → α → β → α
const x _ = x
```

As in Haskell, the underscore serves as a placeholder for parameters we do not care about.

Even though the identity and constant functions already use dependent types, these examples do not illustrate the benefits of this language feature. To that end, we will have a look at the types Fin and Vec soon.

The totality requirement might seem like a small change, but it has a noticeable impact. Omitting cases in function definitions or producing runtime errors is not allowed. For example, there is no literal translation of the `fromJust : Maybe a -> a` function known from Haskell, because there is no way to implement the `Nothing` case.

Furthermore, functions are required to terminate. Since checking this property is not possible in general, only a subset of terminating functions is accepted. A literal translation of the `repeat : a -> [a]` Haskell function would fail the termination checker, because a recursive call must be given structurally smaller parameters. Here, the recursive call takes the same parameter as the original call. We will not go into further detail, because we will not encounter a function that is rejected by the termination checker.

The advantage of the totality requirement is that it turns Agda into a proof verification system. Statements are represented by types and a proof is represented by a term that has the desired type. If the language permitted non-terminating functions, we could write terms of any type, breaking consistency of the embedded logic.

[4] Function

Note that the logic used by Agda is intuitionistic. The main impact for us is that a proof of an existential quantifier has to give one of the elements that are supposed to exist. Other than that, we will seldom notice a difference to classic logic. Let us defer examples of statements and proofs until we have some data types to reason about.

2.1 Data types

Most of the types that we will use are data types. These are similar in notation to the syntax introduced by the `GADTs` Haskell extension.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

This definition introduces the type of natural numbers as given in the standard library[5]. This type is named ℕ, is an element of Set and takes no arguments. It has two constructors, named zero and suc, of which the latter takes a natural number as a constructor parameter. To write down elements of this type, we use constructors like functions and apply them to the required parameters. So examples for elements of ℕ are zero and suc zero.

Let us have a look at a data type with arguments. The type of finite numbers, as given in the standard library[6], takes an argument of type ℕ and contains all numbers that are smaller than the argument.

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

We can see that declarations of the type and of constructors have the same syntax as function declarations. The names of the constructors are shared with the ℕ type. Overloading of names is allowed for constructors, because their types can often be inferred from the context. No other names may be overloaded. Therefore, the constructors of Fin use the suc constructor of ℕ in their types. Also note that the type Fin zero has no elements. To express Fin in Haskell, one has to use type level naturals and activate the `GADTs` extension.

The type of homogeneous sequences is also given in the standard library[7].

```
data List (α : Set) : Set where
  []   : List α
  _::_ : α → List α → List α
```

Before we delve into the type of List, we will have a closer look at the names of the constructors. Since these constructors are not alphanumeric, the question of what makes

[5] Data.Nat
[6] Data.Fin
[7] Data.List

a symbol arises. The language considers most non-whitespace characters valid members of symbols. Notable exceptions are round and curly parentheses and dots. In the print-out, white spaces may be thin, so the reader is asked to infer them from the context.

Underscores have a special meaning when used in symbols. They denote the places where arguments shall be given in an application. For example, the list containing just the number zero can be written as $\mathsf{zero}_{\mathbb{N}} :: [\,]$. Here we already have to disambiguate which zero we are referring to. In a partial function application, this syntax cannot be used though, so the function prepending a $\mathsf{zero}_{\mathbb{N}}$ to a given list would be written as $\_::\_\ \mathsf{zero}_{\mathbb{N}}$.

Like the Fin type, the List type takes one argument. However, this argument is given before the colon. We need to distinguish the places of arguments, because they serve different needs.

An argument given after the colon is called *data index*. Indices are noted like function types. Symbols bound there are not visible in constructors. The actual values given for indices can vary among constructors, as can be seen in the definition of Fin. Whenever one would use the `GADTs` extension in Haskell, an index is called for in Agda.

Arguments given before the colon are called *data parameters*. They are written as a space-separated sequence. All parameters must be given a name. Symbols bound as parameters can be used both in the type of indices and constructor type signatures. No differentiation on parameters is allowed. When declaring a constructor, parameters must appear unchanged in the result type of the signature. Parameters of a data type are not turned into implicit arguments of the constructors, as one might expect. So functions cannot branch on them when evaluating an element of a data type.

It is also possible to combine indices and parameters. An example for this is the type of fixed-length homogeneous sequences as given in the standard library[8].

```
data Vec (α : Set) : ℕ → Set where
  []    : Vec α zero
  _::_  : {n : ℕ} → α → Vec α n → Vec α (suc n)
```

This definition has similarity to Fin and List and employs both a parameter and an index. Unlike Fin, [] is only constructible for a zero index instead of a suc n index. So for each index value there is precisely one constructor with matching type.

2.2 Pattern matching

When defining functions on **data** types, we want to branch on the constructors. In Agda, as in Haskell, branching is expressed by *pattern matching*. A simple example that uses this technique is the length function from the standard library[9].

```
length : {α : Set} → List α → ℕ
length []       = zero
length (_ :: xs) = suc (length xs)
```

[8] Data.Vec
[9] Data.List

A pattern match allows us to replace a symbol in a parameter list of a function definition with a constructor of the corresponding type. In this case, the parameter matched has the type List α, which provides constructors [] and _::_. A clause is given for each constructor. The _::_ constructor introduces a new symbol xs for its second parameter. Symbols introduced in this way can be subject to pattern matches on their own.

Unlike in Haskell, clauses must not overlap. For instance, the following definition will be rejected for covering the case zero zero twice.

```
invalid-pattern-match : ℕ → ℕ → ℕ
invalid-pattern-match zero _    = zero
invalid-pattern-match _    zero = suc zero
```

It will also be rejected for not covering the case (suc i) (suc j). All constructor combinations must be covered to meet the totality requirement.

2.3 A dependently typed function

A common task to perform on sequences is to retrieve an element from a given position. In Haskell, this can be done using the `(!!) :: [a] -> Int -> a` function. When given a negative number or a number that exceeds the length of the list, this function fails at runtime. Such behaviour is prohibited by Agda, so a literal translation of this function is not possible. Ideally, the bound check should happen at compile time. Such a check requires some knowledge of the length of the sequence. The Vec type is accompanied with a corresponding index retrieval function in the standard library[10], as follows.

```
lookup : {α : Set} {n : ℕ} → Fin n → Vec α n → α
lookup zero    (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs
```

In this declaration, the implicit parameter n is used as a type parameter in the remaining function parameters. This appearance blends the type level and value level that are clearly separated in Haskell.

As a notational remark, the arrows between parameters in a type signature can be omitted if the parameters are parenthesized. The declaration above therefore lacks the arrow separating the implicit parameters.

With the totality requirement in mind the definition of lookup may seem incomplete, because we omitted the case of an empty Vec. A closer look reveals that this case cannot happen. The type of [] is Vec α zero, so it can only occur when n is zero. There is no constructor for Fin zero however. The type checker is able to infer this reasoning and recognizes that our definition covers all type-correct cases.

2.4 Semantic equality

In order to use Agda as a proof assistant, we need to encode desired statements as types. Many of the statements we are going to see assert equality of two expressions. We will

[10] Data.Vec

use a type from the standard library[11] that captures semantic equality.

```
data _≡_ {α : Set} (x : α) : α → Set where
  refl : x ≡ x
```

This type corresponds to statements of semantic equality of its parameters. Its elements correspond to proofs of such statements. The only available constructor expresses reflexivity. Let us look at a simple example.

```
length-one : {α : Set} → (x : α) → length (x :: []) ≡ suc zero
length-one _ = refl
```

Here, we prove that a list containing exactly one given element has length one. Agda is able to reduce the left hand side of the equation using each clause of the length definition once. The reduced term for the left hand side arrives at suc zero, which allows us to use the refl constructor to prove our statement. Precise semantics of the term reduction system backing Agda are given by Setzer (2008).

It can also happen that a type constructed with _≡_ has no elements, as is the case with Fin zero. An example of such a type is $\mathsf{zero}_{\mathbb{N}} \equiv \mathsf{suc}\ \mathsf{zero}_{\mathbb{N}}$. A type is called *inhabited* if it contains elements and *uninhabited* otherwise. Since proofs are elements, we need to look a bit further to prove that a type is uninhabited. The standard library[12] provides a prototype of an uninhabited type.

```
data ⊥ : Set where
```

Since ⊥ has no constructors, any type correct function that maps into ⊥ can never be called with type correct parameters due to the totality requirement. To disprove a statement, we can therefore map it into ⊥ as is done by the negation defined in the standard library[13].

```
¬_ : Set → Set
¬ P = P → ⊥
```

Before continuing our attempt to disprove the recent example statement, we observe that the definition of the negation constitutes a type alias. There is no special syntax, as is needed in Haskell. Instead, a type alias is written as a function into Set, because there is no distinction between type level and value level in a dependently typed language.

For convenience, we will use another type alias from the standard library[11] combining equality and negation.

```
_≢_ : {α : Set} → α → α → Set
x ≢ y = ¬ (x ≡ y)
```

[11] Relation.Binary.Core
[12] Data.Empty
[13] Relation.Nullary

We can now formulate the wrongness of the recent example in a positive way.

```
zero≢one : zero_ℕ ≢ suc zero_ℕ
```

Essentially, zero≢one shall be a function that maps proofs of $\mathsf{zero}_{\mathbb{N}} \equiv \mathsf{suc}\ \mathsf{zero}_{\mathbb{N}}$ into $\bot$, so it takes one parameter. Just there is no proof of $\mathsf{zero}_{\mathbb{N}} \equiv \mathsf{suc}\ \mathsf{zero}_{\mathbb{N}}$. In Agda, we need to express this insight using the *absurd pattern* written as (). It is a special case of a pattern match and expresses that none of the available constructors is applicable in a type correct way. In a function definition clause it replaces the parameter that is claimed to have no available constructors. When doing so, the right hand side is simply omitted.

```
zero≢one ()
```

Agda verifies our claim and here it agrees, because the head constructors of the sides of the equation differ already.

As a notable difference to classic logic, we cannot conclude a statement p from its double negation ¬ (¬ p) in the intuitionistic logic used by Agda. To get an idea why this conclusion is not available, we can look at the types. Assuming p has a type α, the type of ¬ (¬ p) is $\alpha \to \bot \to \bot$. A function of the latter type provides no means for constructing an element of α.

Let us also verify that _≡_ describes an equivalence relation. Since reflexivity holds by construction, we might wonder whether additional requirements are needed to establish symmetry and transitivity. Both properties are asserted in the standard library[14].

```
sym : {α : Set} {a b : α} → a ≡ b → b ≡ a
sym refl = refl

trans : {α : Set} → {a b c : α} → a ≡ b → b ≡ c → a ≡ c
trans refl eq = eq
```

As we can see, these assertions are provable. The way they are proven exhibits an aspect of dependently typed pattern matching. When we use a constructor in a pattern match, Agda unifies the type of the constructor with the type of the parameter. For types such as ℕ and List, all constructors yield the data type exactly, so the unification is trivial. On the other hand, the pattern match in sym causes a ≡ b to be unified with x ≡ x. Here the parameter b is replaced by a, so the return type is reduced to a ≡ a. A similar unification happens in the proof of trans. It changes the type of the second parameter to a ≡ c.

3 Implementing `assoc` in Agda

After this basic introduction, we want to move on the application to bidirectionalization. We keep introducing new aspects of Agda and functions from the standard library as needed. Our first goal is to phrase the definition of `assoc` in Agda, because it is all that is needed to formulate and prove the first lemma from Voigtländer (2009).

[14] Relation.Binary.PropositionalEquality

3.1 `IntMap`

In Haskell, an `IntMap` is a partial mapping defined on integers. It can be thought of as a table and therefore can only have a finite domain. This type is not predefined in the standard library of Agda, so we have to come up with an own definition. A naive implementation would be to construct an association list of pairs of inputs and mapped values.

We want to use a different implementation to ease the formulation of the first lemma, which will express an equality about such mappings. We opt not to define an equivalence relation on representations of these mappings. So the representations need to be unique. The proposed association lists do not have this property, because reordering a list does not change mapping it represents.

When looking at the use case of our `IntMap`s, we will observe a bound on the contained numbers. So instead of using a mapping from $\mathbb{N}$, we will map from Fin m for some yet unknown bound m. Since the domain can now be assumed to be finite, we can represent a mapping as a list of length m with the indices being the possible inputs.

```
FinMap : ℕ → Set → Set
FinMap m α = Vec α m
```

This type alias allows us to reuse the lookup function for retrieving elements. Unfortunately, it does not fulfill our need, because we have to represent mappings from arbitrary finite subsets of $\mathbb{N}$. A mapping with the domain $\{5\}$ cannot be represented as a FinMap. To achieve this, we permit each element to be individually absent using the Maybe type defined in the standard library[15].

```
data Maybe (α : Set) : Set where
  nothing : Maybe α
  just    : α → Maybe α
```

Now we wrap the content type of our FinMap definition in a Maybe.

```
FinMapMaybe : ℕ → Set → Set
FinMapMaybe m α = Vec (Maybe α) m
```

Applying the lookup function to a FinMapMaybe now results in a Maybe α, as does the `lookup` function used in Voigtländer (2009). We remember that, apart from `lookup`, we will have to implement `empty`, `fromAscList`, `insert` and `union`. Even though the standard library does have functions[16] almost matching our need for `empty` and `insert`, we give direct implementations here for brevity.

```
empty : {m : ℕ} → {α : Set} → FinMapMaybe m α
empty {zero}  = []
empty {suc m} = nothing :: empty
```

[15] Data.Maybe

[16] The functions replicate and _[_]:=_ are available in Data.Vec.

Since the length parameter m is the first implicit parameter of empty, we can access it by surrounding it with curly braces. Then we can pattern match on it as usual. We do not have to mention it in the recursive invocation of empty, because it can be inferred from the expected return type.

```
insert : {m : ℕ} → {α : Set} → Fin m → α → FinMapMaybe m α →
         FinMapMaybe m α
insert zero    a (x :: xs) = just a :: xs
insert (suc i) a (x :: xs) = x :: insert i a xs
```

In order to express `fromAscList`, we need a notion of tuples. We will defer this definition to a later time and just assume that _×_ is the type of pairs and that _,_ is the corresponding constructor for now.

```
fromAscList : {m : ℕ} → {α : Set} → List (Fin m × α) → FinMapMaybe m α
fromAscList []             = empty
fromAscList ((i , a) :: xs) = insert i a (fromAscList xs)
```

Note that our definition of fromAscList does not require an ascending list. The corresponding Haskell function was using this property for efficiency. We will later use it in this more general form for defining a function restrict, but keep the name, because it is also an abbreviation of "from association list".

With the `union` function, we depart a bit from the original definition. The left-biasedness required in Voigtländer (2009) is kept, but the types change. Again peeking into the use case, we will see that the second parameter of union will always be fully defined. In that case, the result is always fully defined and this property can be represented in the types. Even though Voigtländer et al. (2013) presented a variant where a disjoint union is used instead, we rely on these properties here. To define it, we use the tabulate function from the standard library[17], which turns a function into a Vec α m by recording the results on all the possible inputs. The definition of tabulate needs the function composition _∘_ also available from the standard library[18].

```
tabulate : {n : ℕ} {α : Set} → (Fin n → α) → Vec α n
tabulate {zero}  f = []
tabulate {suc n} f = f zero :: tabulate (f ∘ suc)
```

Furthermore, we need the independently typed version of the maybe function from the standard library[19]. It works the same way as Haskell's `Maybe` eliminator.

```
maybe : {α β : Set} → (α → β) → β → Maybe α → β
```

Then, we can define the left-biased union of two functions using the λ notation for anonymous functions.

[17] Data.Vec
[18] Function
[19] Data.Maybe

```
union : {m : ℕ} → {α : Set} → FinMapMaybe m α → FinMap m α →
        FinMap m α
union h₁ h₂ = tabulate (λ i → maybe id (lookup i h₂) (lookup i h₁))
```

The anonymous function passed to tabulate takes an index i of type Fin m and tries to look it up in h_1. Since h_1 is a FinMapMaybe, the result of the lookup can be nothing, in which case the other mapping h_2 is tried. Otherwise the wrapping just constructor is removed by maybe.

3.2 `checkInsert`

The `checkInsert` function given above is supposed to extend an existing mapping with a new pair ensuring that already existing inputs are mapped to the same elements as before. If this update cannot be achieved, the function shall fail. We remember its Haskell type to be `Eq a => Int -> a -> IntMap a -> Maybe (IntMap a)`. Translating this function to Agda requires more effort. The first issue is the use of the `Eq` type class. As of this writing, there is no feature in Agda that directly corresponds to Haskell's type classes. To avoid using the `Eq` type class, we can simply pass a comparison function as an additional parameter. Using the Bool type with the constructors true and false from the standard library[20], we can type a comparison function as $\alpha \to \alpha \to$ Bool. Then, we can draft a first version of checkInsert.

```
checkInsert : {m : ℕ} → {α : Set} → (α → α → Bool) → Fin m → α →
              FinMapMaybe m α → Maybe (FinMapMaybe m α)
```

The original function uses `case` to branch on the result of `lookup`. In Agda, this language construct has been generalized and therefore gained a new syntax. The **with** keyword can be used in a function clause to introduce an additional parameter and to give an expression for it at the same time. New parameters introduced in this way can be used for pattern matching, except that they need to be separated with | signs. To avoid repetition, the head of a function clause can be replaced with three dots if the omitted parts match the preceding clause. We will defer the generalization to a later time and use it in this basic form for now.

```
checkInsert eq? i b h with lookup i h
...                   | nothing = just (insert i b h)
...                   | just c with eq? b c
...                             | true  = just h
...                             | false = nothing
```

This version of checkInsert still needs improvement. We cannot expect the passed eq? function to represent an equivalence relation, but this property is an essential assumption. Therefore, we have to choose a more precise type for eq?. Even though the standard library has a notion of equivalence relations, we actually want to presume eq? to test semantic

[20] Data.Bool

equality. Using the definition of equivalence relations, as given in the standard library, would require the use of multiple levels of Set, a concept that shall remain unexplained and has been removed from all citations of the standard library. This choice keeps the complexity of our statements manageable. It also matches the usage of the `Eq` type class, because many types from the Haskell `Prelude` instantiate `Eq` as semantic equality.

We already saw the type _≡_ representing statements of semantic equality. Our task is to decide semantic equality though, so we need another definition from the standard library[21] to capture decidability of statements.

```
data Dec (P : Set) : Set where
  yes : P → Dec P
  no  : ¬ P → Dec P
```

An element of Dec P tells us whether the passed type P is inhabited by either giving us an inhabitant or a proof that there is no inhabitant. So the eq? function should have the type (a b : α) → Dec (a ≡ b).

Another issue with the eq? parameter of checkInsert is that it is tedious to pass around. Every function that uses the previous definition of checkInsert has to carry an eq? parameter as well. In Haskell, this passing of eq? has been avoided by the usage of the `Eq` type class. In Agda, the usual approach is to push such symbols into module parameters. Such parameters can be used like any other function, but the module will not contain definitions for them. When using a parameterized module, a user has to provide definitions for the parameters in order to obtain contained functions. We will rename α to Carrier to avoid confusion with local bindings and rename eq? to deq as a short hand for "decidable equality". Without delving into the syntax of module parameters, we will assume that the following symbols are defined.

```
Carrier : Set
deq : (b c : Carrier) → Dec (b ≡ c)
```

Such a deq does not exist for all types bound as Carrier, just like there are Haskell types that are not instances of `Eq`. For example, when Carrier is a function type, defining deq is impossible. Types such as ℕ and Fin usually support equality tests in the standard library[22] though.

```
_≟_ : {n : ℕ} → (x y : Fin n) → Dec (x ≡ y)
```

Now we can apply this machinery to the definition and implementation of checkInsert.

```
checkInsert : {m : ℕ} → Fin m → Carrier → FinMapMaybe m Carrier →
              Maybe (FinMapMaybe m Carrier)
checkInsert i b h with lookup i h
...                 | nothing = just (insert i b h)
```

[21] Relation.Nullary.Core
[22] Data.Fin.Props

```
...                 | just c with deq b c
...                         | yes b≡c  =  just h
...                         | no b≢c   =  nothing
```

3.3 `assoc`

The Haskell type of `assoc` is `Eq a => [Int] -> [a] -> Maybe (IntMap a)`. To phrase it in Agda, we need to replace the use of `IntMap` with FinMapMaybe and use Carrier and deq in place of `Eq a`. These changes are necessary to use the previously defined checkInsert. Furthermore, we need the monad bind operator $\_ \gg\!= \_$ from the standard library[23].

```
_>>=_ : {α β : Set} → Maybe α → (α → Maybe β) → Maybe β
nothing >>= _  =  nothing
just a  >>= f  =  f a
```

This version of the bind operator was rewritten to avoid using other parts of the standard library. We arrive at an almost literal translation of the `assoc` function given earlier.

```
assoc : {m : ℕ} → List (Fin m) → List Carrier →
        Maybe (FinMapMaybe m Carrier)
assoc []       []        =  just empty
assoc (i :: is) []       =  nothing
assoc []       (b :: bs) =  nothing
assoc (i :: is) (b :: bs) =  assoc is bs >>= checkInsert i b
```

The cases where the lists have different lengths consume two lines. What if assoc could require the lists to have the same length? This could be enforced by requiring an explicit proof object.

```
assoc : {m : ℕ} → (is : List (Fin m)) → (bs : List Carrier) →
        length is ≡ length bs → Maybe (FinMapMaybe m Carrier)
```

The cases where the lengths differ would no longer have to be implemented. However, Agda would need to be told explicitly that they cannot occur. So instead of going this route, we will restrict the type of the lists.

The length is going to be part of the type and carried as a parameter given to Vec. An implicit length parameter n is introduced ensuring that the lengths are the same.

```
assoc : {n m : ℕ} → Vec (Fin m) n → Vec Carrier n →
        Maybe (FinMapMaybe m Carrier)
assoc {zero}  []       []        =  just empty
assoc {suc n} (i :: is) (b :: bs) =  assoc is bs >>= checkInsert i b
```

[23] Data.Maybe

Here we pattern match on the length of the fixed length lists. For each length, there is only one possible Vec constructor available. Indeed the pattern match on the length parameter can be dropped from the implementation as it can be inferred from the pattern match on the vectors.

4 Transforming statements to Agda

We will now try to formulate and prove the first lemma given in Voigtländer (2009). The only difference here is the change of the return type of `assoc` from `Either` to `Maybe`.

Lemma 1 (Voigtländer, 2009) For every `is :: [Int]`, type τ that is an instance of `Eq`, and `f :: Int -> τ`, we have

```
assoc is (map f is) ≡ Just h
```

for some `h :: IntMap τ` with

```
lookup i h ≡ if elem i is then Just (f i) else Nothing
```

for every `i :: Int`.

The use of $\equiv$ in the above citation refers to semantic equality and therefore matches nicely with the type _≡_ defined above. Translating this lemma to Agda still requires adapting a few aspects. Where `IntMap` was used, we now use FinMapMaybe. Since our assoc operates on Vec and expects Fin m instead of `Int`, we need to mention these parameters at least implicitly. The use of `map` in a parameter of `assoc` means that we need a version of `map` that works on Vec. It is readily available from the standard library[24] and has the following type.

$$\mathsf{map}_{\mathsf{Vec}} : \{\mathsf{n} : \mathbb{N}\}\ \{\alpha\ \beta : \mathsf{Set}\} \to (\alpha \to \beta) \to \mathsf{Vec}\ \alpha\ \mathsf{n} \to \mathsf{Vec}\ \beta\ \mathsf{n}$$

Observe that it retains the length of the passed Vec. Our implementation of `Eq` assumes the global symbols Carrier and deq, so these are not introduced anymore.

Lemma 1′ Let $\mathsf{m}\ \mathsf{n} : \mathbb{N}$. For every $\mathsf{is} : \mathsf{Vec}\ (\mathsf{Fin}\ \mathsf{m})\ \mathsf{n}$, and every $\mathsf{f} : \mathsf{Fin}\ \mathsf{m} \to \mathsf{Carrier}$, we have

$$\mathsf{assoc}\ \mathsf{is}\ (\mathsf{map}_{\mathsf{Vec}}\ \mathsf{f}\ \mathsf{is}) \equiv \mathsf{just}\ \mathsf{h}$$

for some $\mathsf{h} : \mathsf{FinMapMaybe}\ \mathsf{m}\ \mathsf{Carrier}$, such that for every $\mathsf{i} : \mathsf{Fin}\ \mathsf{m}$ we have

$\mathsf{lookup}\ \mathsf{i}\ \mathsf{h} \equiv \mathsf{just}\ (\mathsf{f}\ \mathsf{i})$ if i is an element of is and nothing otherwise.

Compared to the original version, the most significant difference is the use of semantic equality instead of an arbitrary equivalence relation captured by the type class `Eq`. This resembles the original intention and cuts down the length of proofs considerably.

[24] Data.Vec

Still the claimed existence of the element h is not very handy and we have not yet considered how to explain its property to Agda. The intuitionistic nature would require a proof to give an explicit element for h. Instead, we will consider being explicit about what h is in the lemma itself. Observe that the claimed property on h uniquely defines it. So we define a function that turns a given function and a list representing the desired domain into a FinMapMaybe. To ease the definition, we use the functions zip and $\mathsf{map}_{\mathsf{List}}$ from the standard library[25] that closely match their Haskell counterparts.

```
restrict : {α : Set} → {m : ℕ} → (Fin m → α) → List (Fin m) →
           FinMapMaybe m α
restrict f is = fromAscList (zip is (map_List f is))
```

Note that this definition does not necessarily provide an ascending list to fromAscList. Given this function, we try to replace h with restrict f is. Except that, restrict wants a List and we give a Vec. The standard library[26] has a function that facilitates the conversion process.

```
toList : {n : ℕ} {α : Set} → Vec α n → List α
```

Wrapping is in toList the application to restrict is type correct allowing us to proceed.

The property given on h holds by construction. We will therefore remove it from our version of the lemma. Even though this yields a somewhat different assertion, the resulting lemma will suffice to build on.

```
lemma-1 : {m n : ℕ} → (is : Vec (Fin m) n) → (f : Fin m → Carrier) →
          assoc is (map_Vec f is) ≡ just (restrict f (toList is))
```

4.1 A proof

This version of lemma-1 is very convenient, because both assoc and restrict have the same structure of successively inserting elements into empty. A paper proof would induct on the list is and leave the details as an exercise to the reader like Voigtländer (2009). Instead, we will use this induction as an example to see how Agda source is developed. Most Agda source is written using the Emacs editor, due to its supportive `agda2-mode`. During development, our code can contain holes. They are written as ? or {! !} and behave like `undefined` in Haskell in that they can have any type. We use them to denote places that we wish to defer writing down. The `agda2-mode` for Emacs provides a number of ways working with them. For example, we can ask for the type of a particular hole. Details on the usage of the editor can be found in the Agda wiki (WIKI). So here is our first attempt to the lemma, writing a hole in place of a proof.

```
lemma-1 is f = {! !}
```

[25] Data.List
[26] Data.Vec

The idea was to induct on is. An induction on a list nicely translates to branching on the constructors, which is an operation provided by the editor environment and is called case split. So we ask the editor to case split on the parameter is.

```
lemma-1 []       f = {! !}
lemma-1 (i :: is) f = {! !}
```

When we ask for the type of the first hole, we receive an equality type with reduced terms. For instance, $\mathsf{map}_{\mathsf{Vec}}$ f [], as found on the left hand side, is reduced to [] using the base case of the definition of $\mathsf{map}_{\mathsf{Vec}}$. In fact, the type of the hole reduces to just empty ≡ just empty, so we can use the refl constructor to prove the base case. Note that empty is not a fully reduced form, because its first definition clause can be applied. When giving reductions, we will use terms that are brief and bear some insight, but not necessarily fully reduced. The reduction of the type of the second hole arrives at the following type.

```
assoc is (map_Vec f is) >>= checkInsert i (f i) ≡ just (insert i (f i) (restrict f (toList is)))
```

To prove this statement, we will have to combine multiple results. One part is the induction hypothesis and another part is some result about checkInsert behaving like insert in this context. We already learned about the proof combinator trans, that expresses transitivity of semantic equality. Instead of using it directly, we will use a few sugar functions from the standard library[27]. Their aim is to improve readability of proofs for humans. The technique is called *equality reasoning* and provides the functions begin_, _≡⟨_⟩_ and _□. Instead of giving definitions, an example shall explain the usage.

```
begin term1
  ≡⟨ proof1 ⟩
term2
  ≡⟨ proof2 ⟩
term3 □
```

The above expression is a verbose form of trans proof_1 proof_2. Additionally, it tells the reader about the types of the involved proofs. In the expression above, proof_1 has the type $\mathsf{term}_1 \equiv \mathsf{term}_2$, proof_2 has the type $\mathsf{term}_2 \equiv \mathsf{term}_3$, and the whole expression has the type $\mathsf{term}_1 \equiv \mathsf{term}_3$. Using this technique, we can extend our proof.

```
lemma-1 []       f = refl
lemma-1 (i :: is) f = begin
  (assoc is (map_Vec f is) >>= checkInsert i (f i))
    ≡⟨ {! !} ⟩
  (just (restrict f (toList is)) >>= checkInsert i (f i))
    ≡⟨ refl ⟩
  checkInsert i (f i) (restrict f (toList is))
    ≡⟨ {! !} ⟩
  just (insert i (f i) (restrict f (toList is))) □
```

[27] Relation.Binary.PropositionalEquality.≡-Reasoning

For clarity, we added a reduction step with the proof refl. This step adds no value from a verification point of view, but it can make the proof easier to read. When looking at the first hole, we observe that its type is similar to the induction hypothesis. We can fill the hole by applying $\lambda\ \mathsf{h} \to \mathsf{h} \gg\!= \mathsf{checkInsert\ i\ (f\ i)}$ to both sides of the hypothesis. This operation is facilitated by the standard library[28].

```
cong : {α β : Set} → {a b : α} → (f : α → β) → a ≡ b → f a ≡ f b
cong f refl = refl
```

We use an editor operation called refine to improve said hole. To do that, we enter the expression cong (λ h → h >>= checkInsert i (f i)) in the hole and ask the editor to refine. The agda2-mode then looks at the type to discover that the expression we gave does not fully match the required type. It is a function whereas an equality proof was required. Missing parameters are turned into new holes, so we turned one hole into another. The new hole, we are told, has the shorter type $\mathsf{assoc\ is\ (map_{Vec}\ f\ is)} \equiv \mathsf{just\ (restrict\ f\ (toList\ is))}$, so we can directly insert our induction hypothesis lemma-1 f is. We defer the proof of the other hole by introducing a new lemma.

```
lemma-checkInsert-restrict : {m : ℕ} →
  (f : Fin m → Carrier) → (i : Fin m) → (is : List (Fin m)) →
  checkInsert i (f i) (restrict f is) ≡ just (restrict f (i :: is))
```

Assuming this lemma, we can complete the proof.

```
lemma-1 []       f = refl
lemma-1 (i :: is) f = begin
  (assoc is (map_Vec f is) >>= checkInsert i (f i))
    ≡⟨ cong (λ h → h >>= checkInsert i (f i)) (lemma-1 is f) ⟩
  (just (restrict f (toList is)) >>= checkInsert i (f i))
    ≡⟨ refl ⟩
  checkInsert i (f i) (restrict f (toList is))
    ≡⟨ lemma-checkInsert-restrict f i (toList is) ⟩
  just (insert i (f i) (restrict f (toList is))) □
```

4.2 Insight into checkInsert

In order to prove lemma-checkInsert-restrict, we introduce a technique presented by Norell (2008). It is called *view* and shall not be confused with a view in a database or bidirectionalization setting. In Agda, a view consists of a parameterized data type whose sole purpose is to encode some insight about its parameters, and a function returning elements of the view type. Similarly to `GADTs`, the constructors give elements of specific subtypes. So by pattern matching on the constructors, we can learn something about the parameters

[28] Relation.Binary.PropositionalEquality

of the data type. In fact, we already saw a slightly degenerated example of a view. It was, the _≡_ type, which did not come with a corresponding function. Nevertheless, matching on the refl constructor causes the type checker to unify the constructor type with the parameter type. We used this unification in the proofs of cong, sym and trans.

The insight we want to gain here is a more abstract representation of the behaviour of checkInsert. Looking at the definition, we can see that there are three possible outcomes. If the element being inserted already is in the domain of the given mapping, it either returns just the unchanged mapping or nothing. Otherwise it will insert the missing element and return just an updated mapping. Some functions using the view we are about to construct will need intermediate steps besides the case distinction. We therefore pass additional insights gained in the process as constructor parameters.

```
data InsertionResult {m : ℕ} (i : Fin m) (b : Carrier)
  (h : FinMapMaybe m Carrier) : Maybe (FinMapMaybe m Carrier) → Set where
  same  : lookup i h ≡ just b → InsertionResult i b h (just h)
  new   : lookup i h ≡ nothing → InsertionResult i b h (just (insert i b h))
  wrong : (c : Carrier) → b ≢ c → lookup i h ≡ just c →
          InsertionResult i b h nothing
```

The outcomes of checkInsert correspond to the constructors of InsertionResult. For each constructor, the actual outcome can be found in the last type parameter of the type. The simplest constructor is new. It comes with a proof of the failure of lookup. In the same constructor, we not only learn that lookup is successful. The result of lookup is precisely the value we were trying to insert. In the wrong constructor, we learn about an element different from b to be the result of lookup.

We specialize the type to InsertionResult i b h (checkInsert i b h) when constructing elements of the view. Pattern matching a constructor will then cause the checkInsert term to be unified with the expected outcome of the respective constructor. This is similar to matching the refl constructor of _≡_. We still need the function returning elements of InsertionResult.

```
insertionresult : {m : ℕ} → (i : Fin m) → (b : Carrier) →
                  (h : FinMapMaybe m Carrier) →
                  InsertionResult i b h (checkInsert i b h)
```

We need to examine the same expressions as in the checkInsert definition in order to reduce the application of checkInsert used in the result type of insertionresult. So we add **with** constructs for the contained lookup and deq calls. The created holes can be immediately refined to the corresponding constructors, since the checkInsert call reduces to the terms given in the constructor types. Using a different constructor than the intended one in any of the holes causes a type error.

```
insertionresult i b h with lookup i h
...                   | nothing = new {!!}
...                   | just c with deq b c
```

```
...                                  | yes b≡c  =  same {! !}
...                                  | no b≢c  =  wrong c b≢c {! !}
```

Looking at the new case, we still need a proof of lookup i h ≡ nothing. It seems to be immediate as we just discovered it by case splitting on the result of lookup i h. Unfortunately, the connection between the reduced term nothing and the unreduced term lookup i h is lost.

The standard library[29] comes to the rescue with its inspect function. It takes a function f and a parameter x to f and gives an object that remembers the connection between a deferred application f x and an immediate one. Its result can be pattern matched using the sole constructor [_], whose parameter has the type f x ≡ y if y is the immediate application f x with term reduction applied. A simplified but sufficient definition and explanation can be found in Norell (2008). Trusting this sketch, we add an inspect (lookup i) h to the first **with**.

```
insertionresult i b h with lookup i h  | inspect (lookup i) h
insertionresult i b h  | nothing       | [p]  =  new p
insertionresult i b h  | just c        | [p] with deq b c
insertionresult i b h  | just c        | [p]  | yes b≡c  =  same {! !}
insertionresult i b h  | just c        | [p]  | no b≢c  =  wrong c b≢c p
```

Note that, when introducing multiple expressions into a **with**, those expressions are separated by | signs. The parameter p of the [_] constructor of the inspect result immediately fills the holes in the parameters of new and wrong, but not of same. Observe how the type of p depends on the reduction of the lookup by looking at the definition of InsertionResult. In the same constructor, we claimed that the lookup would result in precisely the b passed. So far, we only know that it results in some c and we have a proof of b ≡ c.

To solve this issue, we case split on b≡c in the parameter of yes. In this process, the text editor replaces b≡c with refl and the unification of the constructor type with the parameter type causes c to be replaced with b. Since c occurs as an explicit parameter, it is replaced with .b, which is an example of a dot pattern. By prefixing a term with a dot, we tell the compiler that this term shall be inferred by type checking, instead of being a free parameter or pattern match. Mentioning a variable twice on the left hand side without any dot simply is an error. Most of the time, dot patterns are not written down explicitly, but generated as part of a case split operation.

```
insertionresult i b h  | just .b  | [p]  | yes refl  =  same p
```

4.3 Missing parts for lemma-1

We will now use the machinery created in the previous section to prove the missing lemma-checkInsert-restrict. First of all, let us recall what it asserts.

[29] Relation.Binary.PropositionalEquality

```
lemma-checkInsert-restrict : {m : ℕ} →
  (f : Fin m → Carrier) → (i : Fin m) → (is : List (Fin m)) →
  checkInsert i (f i) (restrict f is) ≡ just (restrict f (i :: is))
```

We are mainly reasoning about the result of a checkInsert here, so we can immediately employ the previously constructed view.

```
lemma-checkInsert-restrict f i is with insertionresult i (f i) (restrict f is)
...                                | r = {! !}
```

Unfortunately, the attempt to case split r fails, because some types do not unify. When inserting a constructor here, its type should unify with the result type of the insertionresult call. Both types are equal, except for their last parameter of InsertionResult. That last parameter is a checkInsert call in the return type of insertionresult and a particular outcome in the type of each InsertionResult constructor. None of these terms can be reduced without introducing them in a new **with**. So we put the checkInsert term in a **with**, without actually being interested in seeing its reductions.

```
lemma-checkInsert-restrict f i is with checkInsert i (f i) (restrict f is)
                                  | insertionresult i (f i) (restrict f is)
lemma-checkInsert-restrict f i is | ._ | same p = {! !}
lemma-checkInsert-restrict f i is | ._ | new _ = refl
lemma-checkInsert-restrict f i is | ._ | wrong c fi≢c p = {! !}
```

We replaced the boring reductions with underscores. They still carry the dots that tell the compiler to infer those terms. We observe that the right hand side of the goal reduces to just (insert i (f i) (restrict f is)). The new case turned out to be trivial, since checkInsert produces the very same insert call in this case. We need more lemmata for the other holes. For the same case, the left hand side does not contain an insert call that made the new case trivial. We need to prove that it can be dropped from the right hand side. This insight can be formulated as follows.

```
lemma-insert-same : {τ : Set} {m : ℕ} → (h : FinMapMaybe m τ) →
  (i : Fin m) → (a : τ) → lookup i h ≡ just a → h ≡ insert i a h
```

Note that this function takes a proof of the success of lookup as a parameter. We can see that preconditions are turned into universal quantifiers on proof objects in Agda.

To prove this lemma, we will have to follow the definition of insert. It inducts simultaneously on h and i. h is a Vec, so if it is [], then m is zero. Notice that there is no constructor for an element i of Fin zero, so we can use the absurd pattern to cover this case.

```
lemma-insert-same []          ()      a p
lemma-insert-same (.(just a) :: xs) zero    a refl = refl
lemma-insert-same (x :: xs)   (suc i) a p    =
                  cong (_::_ x) (lemma-insert-same xs i a p)
```

Accessing the proof p is done by case splitting as is done in the proof of cong. The split is successful in the base case, because the lookup call can be reduced to the head of h when i is zero and we have a free variable x for that head. So the dot pattern replaces just x and not the whole map h. The recursive case follows from the induction hypothesis.

Back to lemma-checkInsert-restrict, we want to see that the wrong case actually cannot occur. We will therefore obtain a proof of f i ≡ c, which contradicts the fi≢c parameter of the constructor wrong.

```
lemma-lookup-restrict : {α : Set} {m : ℕ} → (i : Fin m) → (f : Fin m → α) →
  (is : List (Fin m)) → (a : α) → lookup i (restrict f is) ≡ just a → f i ≡ a
```

Actually proving this lemma takes a few more steps and further lemmata. The general idea is to induct on the passed list is. The base case is absurd, because lookup is required to succeed. In the recursive case, we branch on whether the element i being looked up is the same as the head element of the passed list is. If we discover that those elements are equal, we can finish the proof. Otherwise we drop the head of the list and continue. For expressing the proof to Agda, it is helpful to add a few smaller lemmata. They can all be proven using inductive arguments.

```
lemma-lookup-empty : {α : Set} {m : ℕ} → (i : Fin m) →
  lookup {α = Maybe α} i empty ≡ nothing
lemma-lookup-insert : {α : Set} {m : ℕ} → (i : Fin m) → (a : α) →
  (h : FinMapMaybe m α) → lookup i (insert i a h) ≡ just a
lemma-lookup-insert-other : {α : Set} {m : ℕ} → (i j : Fin m) → (a : α) →
  (h : FinMapMaybe m α) → i ≢ j → lookup i h ≡ lookup i (insert j a h)
```

Note that lemma-lookup-empty explicitly gives a value for the implicit parameter named α of the lookup function. Otherwise Agda would be unable to figure the content type of the Maybe returned by lookup. Accessing implicit parameters by name is also allowed in function clauses.

With all of these lemmata in place, we could pass a proof of f i ≡ c to fi≢c, which results in an element of ⊥. This application could be introduced using another **with**. Instead, we will use another helper function from the standard library[30] for convenience.

```
contradiction : {P : Set} {Whatever : Set} → P → ¬ P → Whatever
contradiction p ¬p with ¬p p
...                   | ()
```

Assuming lemma-insert-same and lemma-lookup-restrict, we can complete the proof of lemmata-checkInsert-restrict.

```
lemma-checkInsert-restrict f i is with checkInsert i (f i) (restrict f is)
                                     | insertionresult i (f i) (restrict f is)
lemma-checkInsert-restrict f i is | ._ | same p = cong just
```

[30] Relation.Nullary.Negation

```
                                  (lemma-insert-same _ i (f i) p)
lemma-checkInsert-restrict f i is | ._ | new _  =  refl
lemma-checkInsert-restrict f i is | ._ | wrong c fi≢c p  =  contradiction
                                  (lemma-lookup-restrict i f is c p) fi≢c
```

Note that we did not spell out the map that is passed to lemma-insert-same and used the placeholder _ instead. Doing so is valid, because in the map can be deduced from the proof p passed later. So we turned an explicit parameter into an implicit one.

5 Main proof

After having seen how to formalize lemma-1, we want to continue with the second lemma exploring utilities as needed. In this process, we will characterize the domain of a FinMapMaybe in a positive way. Only then, we will give the Agda implementation of bff, because developing its type is elaborate. Formulating and proving the lens laws then mostly is a matter of fitting the pieces together.

5.1 Lemma 2

For formulating the second lemma from Voigtländer (2009), we already have most of the building blocks. After renaming of basic constructs, importing the flip function matching its Haskell counterpart from the standard library[31] and using the module parameters Carrier and deq, it can be written as follows.

Lemma 2 (Voigtländer, 2009) Let m n : $\mathbb{N}$, is : Vec (Fin m) n, v : Vec Carrier n, and h : FinMapMaybe m Carrier. We have that if

assoc is v ≡ just h,

then

$\mathsf{map}_{\mathsf{Vec}}$ (flip lookup h) is ≡ $\mathsf{map}_{\mathsf{Vec}}$ just v.

Unlike the original version, we explicitly require that is and v are of the same length. assoc needs this property to succeed. So the length requirement is not an additional restriction. We turn the success requirement into a another parameter like we did with lemma-insert-same. Thus we can strip the prose.

```
lemma-2 : {m n : ℕ} → (is : Vec (Fin m) n) → (v : Vec Carrier n) →
  (h : FinMapMaybe m Carrier) → assoc is v ≡ just h →
  map_Vec (flip lookup h) is ≡ map_Vec just v
```

Following the hint in Voigtländer (2009), we induct on is. Consequently we have to induct on n and therefore on v as well. Due to term reduction the base case is immediate.

[31] Function

```
lemma-2 []       []       h ph = refl
lemma-2 (i :: is) (b :: bs) h ph = {! !}
```

Naturally we want to employ the induction hypothesis. To invoke it, we need an element of assoc is bs ≡ just h′ for some h′. One way to obtain h′ is to evaluate assoc is bs and hope that its result is just h′. We use the **with** keyword to introduce a new parameter to evaluate.

```
lemma-2 (i :: is) (b :: bs) h ph with assoc is bs
...                               | nothing = {! !}
...                               | just h′  = {! !}
```

At this point, we cannot immediately claim that the result is not nothing. Looking back at our definition of assoc, we can see that if the recursive call fails, then so does the whole call. To explain this insight to Agda, let us revisit the type of ph. Originally it was assoc is bs ≡ just h. The assoc call in that type is subject to term reduction after the case distinction. So in the nothing case the type of ph has changed to nothing ≡ just h. There is no constructor for that type since both parameters to _≡_ have their outermost calls fully reduced. When we case split ph, the editor mode replaces it with the absurd pattern.

```
lemma-2 (i :: is) (b :: bs) h ph with assoc is bs
lemma-2 (i :: is) (b :: bs) h ()  | nothing
...                               | just h′ = {! !}
```

Now we have the resulting h′, but we still need a proof of assoc is bs ≡ just h′. This connection can be established using inspect again.

```
lemma-2 (i :: is) (b :: bs) h ph with assoc is bs | inspect (assoc is) bs
lemma-2 (i :: is) (b :: bs) h ()  | nothing       | [ph′]
...                               | just h′       | [ph′] = {! !}
```

The parameter ph′ now has the intended type, so we can use the induction hypothesis lemma-2 is bs h′ ph′. As we did in lemma-1, we try to prove the head and the tail independently.

```
lemma-2 (i :: is) (b :: bs) h ph | just h′ | [ph′] = begin
  lookup i h :: map_Vec (flip lookup h) is
    ≡⟨ cong (flip _::_ (map_Vec (flip lookup h) is)) {! !} ⟩
  just b :: map_Vec (flip lookup h) is
    ≡⟨ cong (_::_ (just b)) {! !} ⟩
  just b :: map_Vec just bs □
```

Instead of solving the first hole here, we move it to a separate lemma.

```
lemma-lookup-assoc : {m n : ℕ} → (i : Fin m) → (is : Vec (Fin m) n) →
  (b : Carrier) → (bs : Vec Carrier n) → (h : FinMapMaybe m Carrier) →
  assoc (i :: is) (b :: bs) ≡ just h → lookup i h ≡ just b
```

Since assoc boils down to repeatedly calling checkInsert, it can be proven by employing the insertionresult view in a similar way as we used for lemma-checkInsert-restrict. The only non-trivial case is new, where an inductive argument is needed.

At first, the second hole might look like the induction hypothesis. Unfortunately, the map that we use to lookup is h, whereas the induction hypothesis would expect h′ there. The difference between those maps is an additional checkInsert call, so we should get the same result using either map for the proof to succeed. Inserting an additional step, we can finally apply the induction hypothesis.

```
lemma-2 (i :: is) (b :: bs) h ph | just h′ | [ph′] = begin
  lookup i h :: map_Vec (flip lookup h) is
    ≡⟨ cong (flip _::_ (map_Vec (flip lookup h) is))
            (lemma-lookup-assoc i is b bs h {! !}) ⟩
  just b :: map_Vec (flip lookup h) is
    ≡⟨ cong (_::_ (just b)) {! !} ⟩
  just b :: map_Vec (flip lookup h′) is
    ≡⟨ cong (_::_ (just b)) (lemma-2 is bs h′ ph′) ⟩
  just b :: map_Vec just bs □
```

The first hole looks as if ph should fit in there, but it does not. By evaluating assoc is bs, we learned a bit about ph. This allowed us to refute its existence in the nothing case. The just case is also affected, so here ph has the type checkInsert i b h′ ≡ just h. To revert the term reduction, we combine it with ph′, which still carries a part of the original type on the left hand side.

We will defer the second hole to another lemma again. It is rather specific and we will spend the next section on its proof.

```
lemma-2-change-mapping : {m n : ℕ} → (i : Fin m) → (is : Vec (Fin m) n) →
  (b : Carrier) → (bs : Vec Carrier n) → (h : FinMapMaybe m Carrier) →
  (h′ : FinMapMaybe m Carrier) → assoc is bs ≡ just h′ →
  checkInsert i b h′ ≡ just h →
  map_Vec (flip lookup h) is ≡ map_Vec (flip lookup h′) is
```

Assuming lemma-lookup-assoc and lemma-2-change-mapping, we can almost complete the proof. The first attempt fails to type check, because some types cannot be uniquely inferred we are told. Indeed, a little ambiguity resides in the use of _::_. It is both a constructor for List and Vec. Disambiguating the one in the first cong usage is enough to solve this case.

```
lemma-2 (i :: is) (b :: bs) h ph | just h′ | [ph′] = begin
  lookup i h :: map_Vec (flip lookup h) is
    ≡⟨ cong (flip _::_Vec_ (map_Vec (flip lookup h) is))
            (lemma-lookup-assoc i is b bs h origph) ⟩
  just b :: map_Vec (flip lookup h) is
    ≡⟨ cong (_::_ (just b)) (lemma-2-change-mapping i is b bs h h′ ph′ ph) ⟩
```

```
just b :: map_Vec (flip lookup h′) is
  ≡⟨ cong (_::_ (just b)) (lemma-2 is bs h′ ph′) ⟩
just b :: map_Vec just bs □
  where origph = trans (cong (flip _>>=_ (checkInsert i b)) ph′) ph
```

5.2 Domain of a FinMapMaybe

The last missing piece for lemma-2 is lemma-2-change-mapping. In it, we are using the list is in two different contexts. It is used to construct the mapping h′ and is used as a list to map over. Why are we not using different lists here? The choice made guarantees that all lookups succeed. Without this property, the additional checkInsert used to construct h from h′ could change a failing lookup from the right hand side to succeed on the left hand side. Simply using different lists leads us to a wrong statement. In fact, this double usage of is makes an inductive proof on is impossible. The induction hypothesis would talk about different mappings h and h′ that would not be suitable for the tail of the map.

So we have to disentangle is into two lists is and js where is shall be used to construct h′ and js shall be used in the map calls. We need to require that all lookups of elements of js in h′ succeed. Formulating this property requires another look into the standard library[32].

```
data All {α : Set} (P : α → Set) : List α → Set where
  []    : All P []
  _::_ : {x : α} {xs : List α} → P x → All P xs → All P (x :: xs)
```

This type allows us to require a parameterized property for each element of a List. Like for List and Vec, there also is a map_All function in the standard library[32]. For a given element i, the desired property is that lookup i h′ results in a just constructor. We do not want to claim anything about the parameter of that just constructor besides its existence.

We need another tool to express the existential quantifier. This can be addressed using the dependent pair, as defined in the standard library[33], by using a special **record** syntax.

```
record Σ (α : Set) (β : α → Set) : Set where
  constructor _,_
  field
    proj1 : α
    proj2 : β proj1
```

Without the **record** syntax, we would have to express this type using the **data** keyword and define the projections as functions. The equivalent written using **data** is rather longer than the previous definition.

[32] Data.List.All
[33] Data.Product

```
data Σ (α : Set) (β : α → Set) : Set where
  _,_ : (a : α) → β a → Σ α β
proj₁ : {α : Set} {β : α → Set} → Σ α β → α
proj₁ (a , _) = a
proj₂ : {α : Set} {β : α → Set} → (s : Σ α β) → β (proj₁ s)
proj₂ (_ , b) = b
```

In contrast, the independent pair would make the parameter β not to depend on α. So the definition of _×_, as used to define fromAscList, is a special case of the dependent pair.

```
_×_ : (α β : Set) → Set
α × β = Σ α (const β)
```

The standard library[34] also contains a sugar function for turning the first parameter of Σ into an implicit one.

```
∃ : {α : Set} → (α → Set) → Set
∃ = Σ _
```

With $\exists$ we can write a type that depends on an unexplained value that shall exist. In the spirit of intuitionistic logic, an inhabitant then consists of an actual element that is claimed to exist and a proof of the desired property about this element. Now we can express what it means for an element m of Maybe Carrier to have a just constructor using $\exists\, \lambda\, x \to m \equiv just\ x$. Then we can replace m with our lookup call and plug it into All.

```
_in-domain-of_ : {m : ℕ} → (is : List (Fin m)) → (FinMapMaybe m Carrier) →
                 Set
_in-domain-of_ is h = All (λ i → ∃ λ x → lookup i h ≡ just x) is
```

Before moving on to use this property, let us prove that it is actually satisfiable. To that end, we remember that it is satisfied when the mapping is constructed using assoc.

```
lemma-assoc-domain : {m n : ℕ} → (is : Vec (Fin m) n) →
  (bs : Vec Carrier n) → (h : FinMapMaybe m Carrier) → assoc is bs ≡ just h →
  (toList is) in-domain-of h
```

To ease the proof, we introduce another lemma saying that an additional checkInsert call does not change a succeeding lookup.

```
lemma-lookup-checkInsert : {m : ℕ} → (i j : Fin m) → (b c : Carrier) →
  (h h′ : FinMapMaybe m Carrier) → lookup i h ≡ just b →
  checkInsert j c h ≡ just h′ → lookup i h′ ≡ just b
```

It can be proven using the insertionresult view. The same case is immediate, the wrong case is absurd and the interesting case is new. There we need to determine that i and j are different to be able to use lemma-lookup-insert-other.

[34] Data.Product

Back to the proof of lemma-assoc-domain, we will induct on n. The base case is immediate. In the induction step, we put the assoc call in a **with** and refute the nothing result, like we did in lemma-2. We also put the assoc call in an inspect, because we will need it later.

```
lemma-assoc-domain []       []       h ph  =  []
lemma-assoc-domain (i :: is) (b :: bs) h ph with assoc is bs  |  inspect (assoc is) bs
lemma-assoc-domain (i :: is) (b :: bs) h ()  |  nothing       |  [ph′]
lemma-assoc-domain (i :: is) (b :: bs) h ph  |  just h′       |  [ph′]  =  {! !}
```

When asked for the type of ph, Agda will answer checkInsert i b h′ ≡ just h. The checkInsert call comes from the recursive clause of assoc. Now we dissect the checkInsert call using the insertionresult view. So we put both the checkInsert call and the view in a **with**. In addition we also inspect the checkInsert call, because we will need it later as well. The wrong case from the view can be easily refuted using ph and the same case works out fine. Just for new there is more work to do.

```
lemma-assoc-domain (i :: is) (b :: bs) h   ph   |  just h′  |  [ph′] with checkInsert i b h′
   |  inspect (checkInsert i b) h′  |  insertionresult i b h′
lemma-assoc-domain (i :: is) (b :: bs) h   ()   |  just h′  |  [ph′]  |  ._
   |  _     |  wrong _ _ _
lemma-assoc-domain (i :: is) (b :: bs) .h′ refl  |  just h′  |  [ph′]  |  ._
   |  _     |  same pl  =  (b , pl) :: (lemma-assoc-domain is bs h′ ph′)
lemma-assoc-domain (i :: is) (b :: bs) ._ refl  |  just h′  |  [ph′]  |  ._
   |  [pc]  |  new _  =  {! !}
```

In the head of the _in-domain-of_ to be constructed we are lucky. The element being looked up is the one we just inserted and we can reuse lemma-lookup-insert. The tail is where things get interesting, because we need to replace h′ from the induction hypothesis by h. We already know that they differ by an actual insertion due to the case distinction of insertionresult. Here we choose not use that information, but revert it using pc. That way, we can use lemma-lookup-checkInsert and avoid having to prove i ≢ j if j is the element being looked up.

```
lemma-assoc-domain (i :: is) (b :: bs) ._ refl  |  just h′  |  [ph′]  |  ._  |  [pc]
   |  new _  =  (b , lemma-lookup-insert i b h′) ::
                (map_All
                  (λ {j} p → proj_1 p ,
                              lemma-lookup-checkInsert j i (proj_1 p) b h′
                                                           (insert i b h′) (proj_2 p) pc)
                  (lemma-assoc-domain is bs h′ ph′))
```

After completing lemma-assoc-domain, we remember that the goal was to find a better formulation of lemma-2-change-mapping that permits us to do an induction. Our idea was to use two lists is for the assoc invocation and js to map over. But we no longer need

to know that the given mapping h′ is constructed using assoc. All we need to know is (toList js) in-domain-of h′.

```
lemma-map-lookup-assoc : {m n : ℕ} → (i : Fin m) → (b : Carrier) →
  (h h′ : FinMapMaybe m Carrier) → checkInsert i b h′ ≡ just h →
  (js : Vec (Fin m) n) → (toList js) in-domain-of h′ →
  map_Vec (flip lookup h) js ≡ map_Vec (flip lookup h′) js
```

With this lemma and lemma-assoc-domain, we can easily prove the missing piece of lemma-2.

```
lemma-2-change-mapping i is b bs h h′ ph′ ph =
  lemma-map-lookup-assoc i b h h′ ph is (lemma-assoc-domain is bs h′ ph′)
```

Before moving on to prove the now missing piece, we look at a sibling of cong in the standard library[35]. It will allow us to treat the head and tail of the pending Vec equality independently.

```
cong₂ : {α β γ : Set} {a a′ : α} {b b′ : β} →
        (f : α → β → γ) → a ≡ a′ → b ≡ b′ → f a b ≡ f a′ b′
cong₂ f refl refl = refl
```

We remember that the idea was to use induction on js to prove lemma-map-lookup-assoc. As usual, the base case is immediate. Since most of the complexity went into the parameter that can be constructed using lemma-assoc-domain, the only interesting part is to prove the equation lookup j h ≡ lookup j h′ for the head of js. The tail equality is the induction hypothesis.

```
lemma-map-lookup-assoc i b h h′ ph []       pj = refl
lemma-map-lookup-assoc i b h h′ ph (j :: js) ((c , pl) :: pj) = cong₂ _::_
  (trans (lemma-lookup-checkInsert j i c b h′ h pl ph) (sym pl))
  (lemma-map-lookup-assoc i b h h′ ph js pj)
```

5.3 Polymorphism and Vec

Let us look at a first attempt to the Agda type of the `bff` function.

```
bff : ({α : Set} → List α → List α) →
      ({α : Set} → {(x y : α) → Dec (x ≡ y)} → List α → List α → List α)
```

Unlike the Haskell version, we can directly express it without the need for the `RankNTypes` compiler extension. Taking a look back at our definition of assoc, we remember that we are using Vec instead of List. For easier reasoning, we should use Vec for bff, too. A first

[35] Relation.Binary.PropositionalEquality

attempt gets stuck in the return type though. Since this issue will require more thought, we will abandon bff for now and just concentrate on the type of its first parameter.

```
get : {α : Set} → {n : ℕ} → Vec α n → Vec α {! !}
```

We need a way to specify the length of the returned Vec. To fill this hole, we need to look a bit further and invoke the corresponding free theorem, as given in Wadler (1989). We will not prove this result and therefore just tell Agda to assume it.

```
postulate
  free-theorem_List : (get : {α : Set} → List α → List α) → {β γ : Set} →
    (f : β → γ) → (l : List β) → get (map_List f l) ≡ map_List f (get l)
```

One motivation given for the introduction of free theorems is the intuition that the length of the list returned from get should be independent of the contents of the passed list and only depend on the length. Using this intuition, we can give a type for get.

```
get : {getlen : ℕ → ℕ} → {α : Set} → {n : ℕ} → Vec α n → Vec α (getlen n)
```

So we need a way to obtain such a getlen function. Our attempt will use the replicate$_{\text{List}}$ function, matching its Haskell counterpart from the standard library[36], to construct a template list of a given length. Since we do not care about the contents of the template, we use the sole element tt of the unit type from the standard library[37].

```
get_List-to-getlen : ({α : Set} → List α → List α) → ℕ → ℕ
get_List-to-getlen get = length ∘ get ∘ flip replicate_List tt
```

Then we pass the template through get and compute its length. This definition gives us a function of the desired type, but we still want some justification that it works for types other than the unit type as well.

```
get_List-length : (get : {α : Set} → List α → List α) →
                  {β : Set} → (l : List β) →
                  length (get l) ≡ get_List-to-getlen get (length l)
```

To prove this result, we introduce a call to map$_{\text{List}}$ outside the application of get and use the free-theorem$_{\text{List}}$ to swap it with get. In the parameter of get, we can then turn it into the desired replicate$_{\text{List}}$ call.

```
get_List-length get l = begin
  length (get l)
    ≡⟨ {! !} ⟩
  length (map_List (const tt) (get l))
    ≡⟨ cong length {! !} ⟩
```

[36] Data.List
[37] Data.Unit

```
    length (get (map_List (const tt) l))
      ≡⟨ cong (length ∘ get) {! !} ⟩
    length (get (replicate_List (length l) tt)) □
```

Fortunately, the first hole is filled by standard library[38].

```
length-map : {α β Set} → (f : α → β) → (xs : List α) →
             length (map_List f xs) ≡ length xs
```

Well it does not fit exactly. The sides of the equation are flipped. We can adjust it using the symmetry property sym. The second hole is our free theorem, again flipping the sides of the equation. Lacking a suitable function in the standard library, we will leave the last hole as an exercise to the reader and just give its type.

```
replicate-length : {α : Set} → (l : List α) →
                   map_List (const tt) l ≡ replicate_List (length l) tt
```

Now we can complete get$_{\text{List}}$-length.

```
get_List-length get l = begin
  length (get l)
    ≡⟨ sym (length-map (const tt) (get l)) ⟩
  length (map_List (const tt) (get l))
    ≡⟨ cong length (sym (free-theorem_List get (const tt) l)) ⟩
  length (get (map_List (const tt) l))
    ≡⟨ cong (length ∘ get) (replicate-length l) ⟩
  length (get (replicate_List (length l) tt)) □
```

Given this result, we have an idea of how any List based get function can be transformed into a Vec based one. We will not further this exercise and instead just use the Vec based type from now on.

5.4 bff

Equipped with a type for get, we can improve the type of bff. While doing so, we also use the module parameters Carrier and deq.

```
bff : ({getlen : ℕ → ℕ} → {α : Set} → {n : ℕ} →
       Vec α n → Vec α (getlen n)) →
      Vec Carrier {! !} → Vec Carrier {! !} → Vec Carrier {! !}
```

Before tackling the holes, we look back at the Haskell definition of `bff`. It uses the `fromJust` function, that produces a runtime error when given the value `Nothing`. Since partial functions are not allowed in Agda, we need to encapsulate the return type of bff

[38] Data.List.Properties

in a Maybe. Voigtländer et al. (2010) revised the type of `bff` to allow arbitrary monads in place of Maybe.

To determine the lengths, we look back at the lens definition for $put\colon S \times V \to S$. So the first and third hole denote the length of Vecs from the source domain and the second hole denotes the length of a Vec from view domain. Since get maps source to view, the length relation getlen should apply here, too. Just getlen is not in scope, so we have to move it out of the first parameter.

```
bff : {getlen : ℕ → ℕ} →
      ({α : Set} → {n : ℕ} → Vec α n → Vec α (getlen n)) →
      {n : ℕ} → Vec Carrier n → Vec Carrier (getlen n) → Maybe (Vec Carrier n)
```

Before transferring the implementation, we need a some utilities again. Since we have no syntactic sugar for lists, we also have no ellipse notation like `[0..n]`. Observing that we only use this construct in the form `[0 .. length l - 1]`, we define a helper for constructing such ranges. At this point, we obtain a trivial bound on the numbers we work with.

```
enumerate : {n : ℕ} → Vec Carrier n → Vec (Fin n) n
enumerate _ = tabulate id
```

Another part of the Haskell `bff` function is `fromAscList (zip s' s)` where `s'` and `s` are lists as follows. `s` is the second parameter of `bff` and we note its length as `n`. Then `[0 .. n - 1]` is bound to `s'`. So the expression considered really gives a fully defined mapping from Fin n into the element type of `s`, which is Carrier here. We defined the type FinMap m Carrier for such mappings earlier. To obtain a FinMap, we first compute a function Fin n → Carrier and then apply tabulate. Our next helper will compute said function from `s`.

```
denumerate : {n : ℕ} → Vec Carrier n → Fin n → Carrier
denumerate = flip lookup
```

The names suggest some variant of an inversion property. It exists and is easy to prove by induction using map-∘ and tabulate-∘ from the standard library[39].

```
lemma-map-denumerate-enumerate : {n : ℕ} → (bs : Vec Carrier n) →
                                 map_Vec (denumerate bs) (enumerate bs) ≡ bs
```

As a last piece we need to use Maybe as a functor to operate on the wrapped values. Like its Haskell counterpart, it is named _<$>_ and defined in the standard library[40].

```
_<$>_ : {α β : Set} → (α → β) → Maybe α → Maybe β
f <$> nothing = nothing
f <$> (just a) = just (f a)
```

[39] Data.Vec.Properties
[40] Data.Maybe

Like the monad bind operator, this version was rewritten to not rely on other parts of the standard library.

```
bff get s v = let s′ = enumerate s
                  g  = tabulate (denumerate s)
                  h  = assoc (get s′) v
                  h′ = (flip union g) <$> h
              in (flip map_Vec s′ ∘ flip lookup) <$> h′
```

Note that, due to g being a FinMap, the application to union is correct. It validates our claim that the second parameter would be fully defined. Furthermore, the result of union is fully defined. This eliminates the need for `error`, which cannot be defined in Agda. It was needed to drop the `Either` container from the result of `assoc` in the Haskell version. Without the distinction into partially and fully defined maps, we would have a hard time proving that the lookup succeeds in any case.

Another remark is that whenever assoc succeeds so does bbf. This was not as easy to see in the original definition since Haskell does not force the programmer to exhibit undefinedness using Maybe. Let us phrase this remark in Agda. Proving it then is a matter of applying the remaining steps of bff to the result of assoc.

```
lemma-assoc-enough : {getlen : ℕ → ℕ} →
  (get : {α : Set} → {n : ℕ} → Vec α n → Vec α (getlen n)) →
  {n : ℕ} → (s : Vec Carrier n) → (v : Vec Carrier (getlen n)) →
  ∃ (λ h → assoc (get (enumerate s)) v ≡ just h) → ∃ λ u → bff get s v ≡ just u
lemma-assoc-enough get s v (h , p) =
  u , cong (_<$>_ (flip map_Vec s′ ∘ flip lookup) ∘ _<$>_ (flip union g)) p
  where s′ = enumerate s
        g  = tabulate (denumerate s)
        u  = map_Vec (flip lookup (union h g)) s′
```

Our choice to explicitly propagate failures from assoc also removes the need for `seq`, as used in the original version. This failure mode is now encapsulated in the usage of `_<$>_`.

5.5 GetPut

Formalizing the first theorem from Voigtländer (2009) is now straight forward. As usual, we have to replace the `Eq` type class and use the type of get defined for usage with our bff. Moreover, the statement that bff is actually defined on the particular input is explicitly spelled out in the usage of just.

```
theorem-1 : {getlen : ℕ → ℕ} →
            (get : {α : Set} → {n : ℕ} → Vec α n → Vec α (getlen n)) →
            {n : ℕ} → (s : Vec Carrier n) → bff get s (get s) ≡ just s
```

We will now develop the proof to be found in Listing 1. The original proof given in Voigtländer (2009) first uses a property that we established as lemma-map-denumerate-enumerate.

It is used to expand the parameter s of get to a form using $\mathsf{map}_{\mathsf{Vec}}$. The resulting term contains a composition of get and $\mathsf{map}_{\mathsf{Vec}}$ in order to apply the free theorem. Previously, we defined the free theorem to work on List, but we need a version on Vec now. So we are claiming that it exists for Vec as well.

```
postulate
  free-theorem_Vec : {getlen : ℕ → ℕ} →
    (get : {α : Set} → {n : ℕ} → Vec α n → Vec α (getlen n)) →
    {β γ : Set} → (f : β → γ) → {n : ℕ} → (l : Vec β n) →
    get (map_Vec f l) ≡ map_Vec f (get l)
```

After the application of the $\mathsf{free\text{-}theorem}_{\mathsf{Vec}}$, we are supposed to use lemma-1. To do that, we need to expand the usage of bff. We explicitly write down this step and similar steps using refl to improve readability. Since the expansion is very long, we move common subexpressions to the local bindings h↦h′ and h′↦r. The remainder of the proof, we are told in Voigtländer (2009), is done by looking at the specifications of utility functions. A mere look is not enough for Agda though, so we first expand h↦h′ to see the usage of union. We can see that both parameters to union are constructed using denumerate s. The second mapping passed to union actually extends the first. Its domain is a superset of the domain of the restricted mapping. In this case, the result of union is the second mapping.

```
lemma-union-restrict : {m : ℕ} → {α : Set} →
                       (f : Fin m → α) → (is : List (Fin m)) →
                       union (restrict f is) (tabulate f) ≡ tabulate f
```

We skip the proof for now. After expanding h′↦r, we can spot a composition of lookup and tabulate. Those functions are somewhat inverse, so we might expect that their composition is the identity. However, this property does not hold, because Agda has no extensionality on functions. We can only prove that the composition is pointwise equal to the identity. To get an idea what that means, we have a look at a simplified version of pointwise equality defined in the standard library[41].

```
_≗_ : {α β : Set} → (f g : α → β) → Set
_≗_ {α} f g = (x : α) → f x ≡ g x
```

Using a different notion of equality means that we cannot use cong to apply such a property, as it can only deal with semantic equality. The standard library[42] provides the statement about the composition called lookup∘tabulate and map-cong, which is able to apply it to $\mathsf{map}_{\mathsf{Vec}}$. Finally we can apply lemma-map-denumerate-enumerate again to reduce our initial expansion and receive the desired result.

To sketch the proof of the skipped lemma-union-restrict, we remember that union was defined using tabulate. Since both sides of the equation are wrapped in tabulate calls, we introduce a new lemma to be proven by the reader.

[41] Relation.Binary.PropositionalEquality
[42] Data.Vec.Properties

```
theorem-1 get s = let h↦h′ = _<$>_ (flip union (tabulate (denumerate s)))
                      h′↦r = _<$>_ (flip map_Vec (enumerate s) ∘ flip lookup)
                  in begin
  bff get s (get s)
    ≡⟨ cong (bff get s ∘ get) (sym (lemma-map-denumerate-enumerate s)) ⟩
  bff get s (get (map_Vec (denumerate s) (enumerate s)))
    ≡⟨ cong (bff get s) (free-theorem_Vec get (denumerate s) (enumerate s)) ⟩
  bff get s (map_Vec (denumerate s) (get (enumerate s)))
    ≡⟨ refl ⟩
  (h′↦r ∘ h↦h′) (assoc (get (enumerate s))
                       (map_Vec (denumerate s) (get (enumerate s))))
    ≡⟨ cong (h′↦r ∘ h↦h′) (lemma-1 (get (enumerate s)) (denumerate s)) ⟩
  (h′↦r ∘ h↦h′ ∘ just) (restrict (denumerate s) (toList (get (enumerate s))))
    ≡⟨ refl ⟩
  (h′↦r ∘ just) (union (restrict (denumerate s) (toList (get (enumerate s))))
                      (tabulate (denumerate s)))
    ≡⟨ cong (h′↦r ∘ just) (lemma-union-restrict (denumerate s)
                                                (toList (get (enumerate s)))) ⟩
  (h′↦r ∘ just) (tabulate (denumerate s))
    ≡⟨ refl ⟩
  just (map_Vec (flip lookup (tabulate (denumerate s))) (enumerate s))
    ≡⟨ cong just (map-cong (lookup∘tabulate (denumerate s)) (enumerate s)) ⟩
  just (map_Vec (denumerate s) (enumerate s))
    ≡⟨ cong just (lemma-map-denumerate-enumerate s) ⟩
  just s □
```

Listing 1: Proof for theorem-1

```
lemma-tabulate-∘ : {m : ℕ} {α : Set} {f g : Fin m → α} →
  f ≗ g → tabulate f ≡ tabulate g
```

Then we need to prove that the function f passed to lemma-union-restrict is pointwise equal to the following function.

```
λ j → maybe id (lookup j (tabulate f)) (lookup j (restrict f is))
```

To do so, we can induct on the list is. For the empty list, we can use lemma-lookup-empty to see that the second lookup results in nothing. The other lookup is eliminated using lookup∘tabulate.

In the induction step, we branch on $j \stackrel{?}{=} i$. An equality here means that we are done with lemma-lookup-insert. The insert call comes from an expansion of fromAscList, which is called by restrict. An inequality means that we can use the induction hypothesis after applying lemma-lookup-insert-other.

5.6 PutGet

For the second theorem from Voigtländer (2009), we need to formulate a precondition on the definedness of bff. Definedness means that the result of bff is just something. That argument to just should exist and due to the intuitionistic nature we require it to be explicitly given.

```
theorem-2 : {getlen : ℕ → ℕ} →
            (get : {α : Set} → {n : ℕ} → Vec α n → Vec α (getlen n)) →
            {n : ℕ} → (s : Vec Carrier n) → (v : Vec Carrier (getlen n)) →
            (u : Vec Carrier n) → bff get s v ≡ just u → get u ≡ v
```

The proof can be found in Listing 2 and is explained in the following. We have established that if a particular assoc call succeeds, then so does bff as lemma-assoc-enough. Here we need the converse. More precisely, we need to drop the _<$>_ calls on the result of assoc. Another lemma will facilitate that task.

```
lemma-<$>-just : {α β : Set} {f : α → β} {b : β} {ma : Maybe α} →
                 f <$> ma ≡ just b → ∃ λ a → ma ≡ just a
lemma-<$>-just {ma = just x}    _ = x , refl
lemma-<$>-just {ma = nothing} ()
```

Since we have two uses of _<$>_ in bff, this lemma is applied twice to arrive at the return value of the assoc call and the corresponding proof.

The proof then starts using propositional equality. Note that the left hand side indirectly talks about bff via u. So our first task is to get the u replaced by something containing bff to be able to reason about it. To that end, we have to temporarily introduce a just on both sides of the equation. We can get rid of it again, because constructors work like injective functions. Since the standard library overloads just as well, we need to disambiguate it.

```
just-injective : {α : Set} → {x y : α} → (just_Maybe x) ≡ (just_Maybe y) → x ≡ y
just-injective refl = refl
```

With just-injective applied, we can reason about just (get u). Our first step is to replace u with the bff call using p. We are not that interested in the embedded assoc call, except for two properties. One of those properties is that assoc succeeds. We are using ph to turn the assoc call into just h. Since we now have a just constructor again, we conclude our just-injective subproof. The next step is to swap the invocations of get and the $\mathsf{map}_{\mathsf{Vec}}$ call from bff using $\mathsf{free\text{-}theorem}_{\mathsf{Vec}}$.

We arrive at a term that looks up all elements from get s′ using the mapping union h g. This is almost the situation needed for lemma-2, except that it expects the mapping to be h. We remember that union is left biased, so we want to establish that it can indeed be dropped. This is successful if all elements being looked up are in the domain of h. Those elements are the ones in get s′, which is the first list passed to assoc, so we can use lemma-assoc-domain. We phrase this observation as a new lemma.

```
lemma-union-not-used : {m n : ℕ} (h : FinMapMaybe m Carrier) →
  (g : FinMap m Carrier)
  (is : Vec (Fin m) n) → (toList is) in-domain-of h →
  map_Vec just (map_Vec (flip lookup (union h g)) is) ≡ map_Vec (flip lookup h) is
```

It can be proven by induction on is and therefore by simultaneous reduction of the domain property. Note that the lookup calls on the left hand side give elements of Carrier whereas the lookup calls on the right hand side give elements of Maybe Carrier. That is why the left hand side has to be wrapped in a $\mathsf{map}_{\mathsf{Vec}}$ just. As in the use of just-injective, we have to temporarily introduce these constructors. This time they do not appear outside the Vec, but within its member type, so we need a different lemma here.

```
map-just-injective : {α : Set} {n : ℕ} {xs ys : Vec α n} →
                     map_Vec just_Maybe xs ≡ map_Vec just_Maybe ys → xs ≡ ys
```

Again, we need to disambiguate just. It can be proven by induction on the implicit parameters using the fact that _::_ is injective, too. When creating a new subproof using map-just-injective, the pieces lemma-union-not-used and lemma-2 nicely fit together to complete the proof of theorem-2.

6 A precondition for bff

The bff we use is only partially defined. As a consequence, the PutGet theorem only holds when bff is defined. We already know that bff get s v succeeds if assoc (get (enumerate s)) v succeeds from lemma-assoc-enough. The only way for assoc to go wrong is to have conflicting associations, which implies that an element in the first parameter is duplicate. Conversely, assoc will succeed if all elements of the first parameter are pairwise different. Note that this condition is not inevitable but sufficient.

To phrase this observation in Agda, we first need a notion of membership. Like our _in-domain-of_ property, membership is not expressed on sets, but on Lists. The standard library[43] defines the Any type to implement it.

```
data Any {α : Set} (P : α → Set) : List α → Set where
  here  : ∀ {x xs} (px  : P x)       → Any P (x :: xs)
  there : ∀ {x xs} (pxs : Any P xs) → Any P (x :: xs)
```

It gives the notion of a parameterized property to be true for any element of a list. The standard library[44] provides type aliases for equality and inequality, which we are interested in.

```
_∈_ : {α : Set} → α → List α → Set
x ∈ xs = Any (_≡_ x) xs
```

[43] Data.List.Any
[44] Data.List.Any.Membership-≡

```
theorem-2 get s v u p with lemma-<$>-just (proj₂ (lemma-<$>-just p))
theorem-2 get s v u p | h , ph = let s′      = enumerate s
                                    g       = tabulate (denumerate s)
                                    h↦h′    = flip union g
                                    h′↦r    = flip map_Vec s′ ∘ flip lookup
                                in begin
  get u
    ≡⟨ just-injective (begin
      get <$> (just u)
        ≡⟨ cong (_<$>_ get) (sym p) ⟩
      get <$> (bff get s v)
        ≡⟨ cong (_<$>_ get ∘ _<$>_ h′↦r ∘ _<$>_ h↦h′) ph ⟩
      get <$> (h′↦r <$> (h↦h′ <$> just h)) □) ⟩
  get (map_Vec (flip lookup (h↦h′ h)) s′)
    ≡⟨ free-theorem_Vec get (flip lookup (h↦h′ h)) s′ ⟩
  map_Vec (flip lookup (h↦h′ h)) (get s′)
    ≡⟨ map-just-injective (begin
      map_Vec just (map_Vec (flip lookup (union h g)) (get s′))
        ≡⟨ lemma-union-not-used h g (get s′)
                                    (lemma-assoc-domain (get s′) v h ph) ⟩
      map_Vec (flip lookup h) (get s′)
        ≡⟨ lemma-2 (get s′) v h ph ⟩
      map_Vec just v
        □) ⟩
  v □
```

Listing 2: Proof for theorem-2

```
_∉_ : {α : Set} → α → List α → Set
x ∉ xs = ¬ (x ∈ xs)
```

A way to phrase that the elements of a list are pairwise different is to say that no element is contained in the list that follows it.

```
data All-different {α : Set} : List α → Set where
  different-[] : All-different []
  different-:: : {x : α} {xs : List α} →
                 x ∉ xs → All-different xs → All-different (x :: xs)
```

Note that we cannot phrase this type using All, because the desired property depends on the tail. We can use it to claim that assoc u v succeeds if All-different (toList u).

```
different-assoc : {m n : ℕ} → (u : Vec (Fin m) n) → (v : Vec Carrier n) →
                  All-different (toList u) → ∃ λ h → assoc u v ≡ just h
```

We want to prove that all individual checkInsert calls made during assoc are turned into insert calls. This assertion holds whenever the lookup of the element being inserted results in nothing. We capture this insight in a trivial lemma which is proven by case splitting the passed proof.

```
lemma-checkInsert-new : {m : ℕ} → (i : Fin m) → (b : Carrier) →
  (h : FinMapMaybe m Carrier) → lookup i h ≡ nothing →
  checkInsert i b h ≡ just (insert i b h)
```

To use it, we need a tool to turn the x ∉ xs proof from different-:: into a lookup i h ≡ nothing. Unfortunately, we cannot use _in-domain-of_ here since that statement only makes positive assertions on the domain. Here we need to know that certain elements are not in the domain of a FinMapMaybe.

```
lemma-∉-lookup-assoc : {m n : ℕ} → (i : Fin m) → (is : Vec (Fin m) n) →
  (bs : Vec Carrier n) → (h : FinMapMaybe m Carrier) →
  assoc is bs ≡ just h → (i ∉ toList is) → lookup i h ≡ nothing
```

Assuming it, we can complete the proof of different-assoc by inducing on the passed Vecs. In the induction step, we put the hypothesis in a **with**, which directly tells Agda that the recursive assoc call succeeds. The rest is a composition of the previous lemmata.

```
different-assoc []       []       _  =  empty , refl
different-assoc (u :: us) (v :: vs) (different-:: u∉us diff-us)
                with different-assoc us vs diff-us
different-assoc (u :: us) (v :: vs) (different-:: u∉us diff-us)
                | h , p  =  insert u v h , (begin
  (assoc us vs >>= checkInsert u v)
    ≡⟨ cong (flip _>>=_ (checkInsert u v)) p ⟩
  checkInsert u v h
    ≡⟨ lemma-checkInsert-new u v h (lemma-∉-lookup-assoc u us vs h p u∉us) ⟩
  just (insert u v h) □)
```

Proving the now missing lemma is mostly a matter of inducing on the Vecs passed to assoc and using previously established techniques. Besides lemma-lookup-empty we need a variant of lemma-lookup-insert-other.

```
lemma-lookup-checkInsert-other : {m : ℕ} → (i j : Fin m) → i ≢ j →
  (b : Carrier) → (h h′ : FinMapMaybe m Carrier) → checkInsert j b h ≡ just h′ →
  lookup i h ≡ lookup i h′
```

It can be proven using lemma-lookup-insert-other by putting the involved terms in **with**. Then we can actually complete the precondition.

```
lemma-∉-lookup-assoc i []       []       .empty refl i∉is
                     =  lemma-lookup-empty i
```

```
lemma-∉-lookup-assoc i (i′ :: is′) (b′ :: bs′) h        ph  i∉is
                       with assoc is′ bs′  |  inspect (assoc is′) bs′
lemma-∉-lookup-assoc i (i′ :: is′) (b′ :: bs′) h        ()  i∉is
                        |  nothing          |  [ph′]
lemma-∉-lookup-assoc i (i′ :: is′) (b′ :: bs′) h        ph  i∉is
                        |  just h′          |  [ph′]  =  begin
  lookup i h
    ≡⟨ sym (lemma-lookup-checkInsert-other i i′ (i∉is ∘ here) b′ h′ h ph) ⟩
  lookup i h′
    ≡⟨ lemma-∉-lookup-assoc i is′ bs′ h′ ph′ (i∉is ∘ there) ⟩
  nothing □
```

By combining lemma-assoc-enough and different-assoc, we can see that there are substantial cases where bff succeeds and theorem-2 is applicable.

7 Conclusion

We succeeded in formalizing the bidirectionalization method and the accompanying proofs presented by Voigtländer (2009) in Agda. Like the original proof, we assumed the $\mathsf{free\text{-}theorem}_{\mathsf{Vec}}$. Most of the work went into proving a fair number of auxiliary assertions. A similar number of assertions could be used from the standard library. Some of our assertions, such as ::-injective, are general and could be added to the standard library. The majority is specific to our application though.

Our choice to implement the `Eq` type class as decidable semantic equality restricts the applicability of our implementation. On the other hand, this choice cuts down the length of proofs and was necessary to avoid dealing with types beyond Set. Future work may investigate the usage of equivalence relations, as provided by the standard library.

An essential part of the proof construction has been to find suitable formulations of the desired assertions and preconditions. Earlier versions of the whole formalization that used List everywhere were discarded, because switching to Vec made a few arguments redundant. For example, the definition of assoc dropped two clauses. Our version of the union function is another example where the choice of the type matters. Its type requires that the second parameter is always fully defined and it transfers this property to the return type. We exploited this assertion in the definition of bff and proofs about it.

The usage of Vec bears a benefit when using bff. Applications of the Haskell `bff` can fail when an update changes the length of the list. In contrast, bff rejects such updates on the type level.

Besides the implementation of bff we reasoned about, Voigtländer (2009) presented further variants. The second method considers *get* functions of type `Eq a => [a] -> [a]`. It uses an adapted `checkInsert` function that avoids inserting duplicate values. Our auxiliary assertions about checkInsert cannot be carried over without modification.

Another variant considers data structures other than lists for *get* to operate on, for example trees. It may be possible to extend our formalization to this variant retaining the assertions about checkInsert including the InsertionResult view.

References

François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981. doi: 10.1145/319628.319634.

Nils Anders Danielsson et al. The Agda standard library version 0.6, 2011. URL `http://www.cse.chalmers.se/~nad/software/lib-0.6.tar.gz`.

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), 2007. doi: 10.1145/1232420.1232424.

Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. doi: 10.1007/978-3-642-04652-0_5.

Anton Setzer. Interactive theorem proving for Agda users. Lecture notes at Swansea University, 2008. URL `http://www.cs.swan.ac.uk/~csetzer/lectures/intertheo/07/interactiveTheoremProvingForAgdaUsers.html`.

Janis Voigtländer. Bidirectionalization for free! (pearl). In *Proceedings of Principles of Programming Languages*, pages 165–176. ACM, 2009. doi: 10.1145/1480881.1480904.

Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Combining syntactic and semantic bidirectionalization. In *Proceedings of International Conference on Functional Programming*, pages 181–192. ACM, 2010. doi: 10.1145/1863543.1863571.

Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. To be published in the Journal of Functional Programming, 2013.

Philip Wadler. Theorems for free! In *Proceedings of Functional Programming languages and Computer Architecture*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404.

WIKI. The Agda wiki, 2013. URL `http://wiki.portal.chalmers.se/agda/`.

Acknowledgements

I would like to thank my supervisor Janis Voigtländer for his constructive guidance. My thanks extends to Klaus Aehlig, Joachim Breitner, Christine Grohne, Hendrik Hoeth, Andres Löh, Stefan Mehner, Marc Schäfer, Daniel Seidel, Wouter Swierstra and Matthias Warkentin for the fruitful discussions.

Index