Formalizing Semantic Bidirectionalization with Dependent Types

Helmut Grohne
University of Bonn
grohne@cs.uni-bonn.de

Andres Löh
Well-Typed LLP
andres@well-typed.com

Janis Voigtländer
University of Bonn
jv@iai.uni-bonn.de

ABSTRACT

Bidirectionalization is the task of automatically inferring one of two transformations that as a pair realize the forward and backward relationship between two domains, subject to certain consistency conditions. A specific technique, semantic bidirectionalization, has been developed that takes a *get*-function (mapping forwards from sources to views) as input — but does not inspect its syntactic definition — and constructs a *put*-function (mapping an original source and an updated view back to an updated source), guaranteeing standard well-behavedness conditions. Proofs of the latter have been done by hand in the original paper, and recently published extensions of the technique have also come with more or less rigorous proofs or sketches thereof.

In this paper we report on a formalization of the original technique in a dependently typed programming language (turned proof assistant). This yields a complete correctness proof, with no details left out. Besides demonstrating the viability of such a completely formal approach to bidirectionalization, we see further benefits:

1. Exploration of variations of the original technique could use our formalization as a base line, providing assurance about preservation of the well-behavedness properties as one makes adjustments.

2. Thanks to being presented in a very expressive type theory, the formalization itself already provides more information about the base technique than the original work. Specifically, while the original by-hand proofs established only a partial correctness result, useful preconditions for total correctness come out of the mechanized formalization.

3. Finally, also thanks to the very precise types, there is potential for generally improving the bidirectionalization technique itself. Particularly, shape-changing updates are known to be problematic for semantic bidirectionalization, but a refined technique could leverage the information about the relationship between the shapes of sources and views now being expressed at the type level, in a way we sketch and plan to explore further.

1. INTRODUCTION

We are interested here in well-behaved, state-based, asymmetric lenses, in which both transformation parts of the BX are total functions. Formally, let S, V be sets. A lens in the above sense is a pair of total functions $get\colon S \to V$ and $put\colon S \times V \to S$ for which the following two properties hold:

$$\forall s \in S. \quad put(s, get(s)) = s \qquad \text{(GetPut)}$$

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

Specifically, we are interested in the case when *get* is a program in a pure functional programming language and *put* is another program in the same language that is automatically obtained from *get* somehow.

Voigtländer (2009) presented a concrete technique, semantic bidirectionalization, that lets the programmer write *get* in Haskell and delivers a suitable *put* for it. The technique is both general and restricted: general in that it works independently of the syntactic definition of *get*, and restricted in that it requires *get* to have a certain (parametrically polymorphic) type. Also, it comes at the price of partiality: even when *get* is indeed a total function, the delivered *put* is in general partial; and while GetPut indeed holds as given above, PutGet becomes conditioned by $put(s, v)$ actually being defined. Recent works have extended semantic bidirectionalization in various ways (Matsuda and Wang, 2013, Voigtländer et al., 2013, Wang and Najd, 2014), both to make it applicable to more *get*-functions (lifting restrictions on *get*'s type, thus allowing more varied behavior) and to make *put* (for a given *get*) defined on more inputs.

The original paper by Voigtländer (2009) gives proofs of the base technique, and papers about extensions of the technique also come with formal statements about correctness (i.e., about satisfying GetPut and PutGet) and proofs or proof sketches thereof. As is typical for by-hand proofs, details are left out and the reader is asked to believe that certain lemmas that are not explicitly proved do indeed hold and could in principle be proved by standard but tedious means. In the programming languages community there is a movement towards working more rigorously by using mechanized proof assistants to establish properties of programs (and of programming languages) in a fully formal way, see for example the PoplMark challenge (Aydemir et al., 2005). We report here on applying this way of thinking to the semantic bidirectionalization technique, which has led to a complete formalization (Grohne, 2013) that moreover provides more precision concerning definedness of *put* than the previous proofs. The proof assistant we use is Agda, which at the same time is a pure functional programming language with an even more expressive type system than Haskell, and we take off from there to discuss further potential such expressivity has in making semantic bidirectionalization itself more useful.

2. LANGUAGE

Agda is what is called a dependently typed programming language. It is a descendant of Haskell, and it is implemented in and syntactically similar to Haskell. Based, like Haskell, on a typed λ-calculus, Agda additionally allows values to occur as parameters to types. This mixing of types and values enables us to encode properties into types, and thus the type checker is able to verify the correctness of proofs: statements are represented by types and a proof is represented by a term that has the desired type. For this to work out, a strong discipline is required so that the type checker's logic remains consistent; in particular, all functions must be total — runtime errors as well as non-termination of programs are ruled out by a combination of syntactic means and type checking rules. We give a brief introduction to the language; a more comprehensive account is given by Norell (2008).

As mentioned, the line between types and values is blurred in a dependently typed language. As a first example, let us have a look at the identity function. We use a slightly simplified version of the definition from the standard library[1].

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

While the definition itself looks much the same as in any functional language, the type declaration is different from what one would have in Haskell, for example. That is because the availability of dependent types changes the way to express polymorphism. Instead of some convention treating certain names in a type (say, all lowercase identifiers) as type variables, we explicitly say here that α shall be an element of Set. The type Set contains all types that we will use, except for itself.[2]

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[3].

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

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 functions on the data types Fin and Vec soon. Data types are introduced by notation as follows.

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

This definition introduces the type of natural numbers as given in the standard library[4]. 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 zero and suc zero are examples for elements of ℕ.

Let us have a look at a data type with arguments. The type of finite numbers, as given in the standard library[5], 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. Therefore, the constructors of Fin use the suc constructor of ℕ in their types. Also note that the type Fin zero has no elements.

The type of homogeneous sequences is also given in the standard library[6].

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

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.

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. Arguments given before the colon are called *data parameters*. They are written as

[1]The id function is available in the Function module. Further footnotes about the origin of functions just mention the module name.

[2]Actually, 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.

[3]Function

[4]Data.Nat

[5]Data.Fin

[6]Data.List

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[7].

```
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.

When defining functions on data types, we want to branch on the constructors by *pattern matching*. A simple example is the length function from the standard library[6].

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

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), since all constructor combinations must be covered to meet the totality requirement.

Let us look at a truly dependently typed function now. 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 behavior 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[7], 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.

[7]Data.Vec

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. Another example in a similar spirit is the definition of the head function from the standard library[7].

```
head : {α : Set} {n : ℕ} → Vec α (suc n) → α
head (x :: _) = x
```

The input type Vec α (suc n) effectively expresses that only non-empty lists can be passed — thus, no runtime error like for the corresponding Haskell function can occur.

For further familiarization, let us look at other polymorphic functions on Lists and/or Vecs. Our first example is to skip every other element of a sequence. When implemented using Lists, its type and implementation closely match what we would write in Haskell.

```
sieve_List : {α : Set} → List α → List α
sieve_List []             = []
sieve_List (x :: [])      = x :: []
sieve_List (x :: _ :: xs) = x :: sieve_List xs
```

Writing it using Vec requires us to give a length expression for the result type. More precisely, we need a function that relates input length to output length.

```
⌈_/2⌉ : ℕ → ℕ
⌈ zero        /2⌉ = zero
⌈ suc zero    /2⌉ = suc zero
⌈ suc (suc n) /2⌉ = suc ⌈ n /2⌉
```

It is available from the standard library[4] and computes the upwards rounded division by 2. Equipped with this function, we can update the type retaining the implementation.

```
sieve_Vec : {α : Set} {n : ℕ} → Vec α n
                              → Vec α ⌈ n /2⌉
```

As another example, we consider the function that reverses a size-indexed list. We can base our implementation on the dependently typed left fold as does the standard library[7].

```
reverse_Vec : {α : Set} {n : ℕ} → Vec α n → Vec α n
reverse_Vec {α} = foldl (Vec α) (λ rev x → x :: rev) []
```

3. SEMANTIC BIDIRECTIONALIZATION

The Haskell version of semantic bidirectionalization, in its easiest form, works for functions of type `[a] -> [a]`, i.e., polymorphic *get*-functions on homogeneous lists. We want to translate the Haskell implementation of "*put* from *get*" given by Voigtländer (2009) to Agda, and redevelop the proofs of the well-behavedness lens laws in parallel. So we should first look at the type of the forward function in Agda. We can think of something like sieve or reverse, so a reasonably general type expressing both the polymorphism and possible type-level information about lengths, would use vectors as follows:

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

where {! !} is a hole that still needs to be filled by some expression. For the sake of maximal generality, we can turn the dependence of the output length on the input length into an explicit function, thus arriving at the following type:

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

This is notation for a dependent pair as defined in the standard library[8], expressing here that there is one component that is a function from ℕ to ℕ and another component whose type depends on the former function (named getlen). Clearly, both $\mathsf{sieve}_{\mathsf{Vec}}$ and $\mathsf{reverse}_{\mathsf{Vec}}$ can be embedded thus, for suitable choices of the getlen function.

That indeed every polymorphic function on homogeneous lists has such an embedding depends on free theorems, as given by Wadler (1989). One free theorem in Haskell is that for every function of type `[a] -> [a]` the length of the list returned is independent of the contents of the passed list, instead only depending on its length. Correspondingly, for list-based get the correct getlen function can be constructively obtained, and then used to define the type of the vector-based variant of get. The details of this construction are given by Grohne (2013).

Now we are in a position to give the main construction from (Voigtländer, 2009). There, it is a Haskell function named `bff` (which is a short form of "bidirectionalization for free") with the following type:[9]

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

Apparently, a *get*-function is turned into a *put*-function, where the latter must be allowed to compare elements for equality. The most interesting bit in Agda of course is how the type plays out. It does become quite a bit more verbose, but that verbosity is useful since the additional pieces carry important information. Without further ado, here is the Agda type for `bff`:

```
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)
```

Let us discuss this type a bit. First of all note how the dependent pair from the above prototypical Agda type for get, which has to take the role of the `(forall a. [a] -> [a])` argument function in Haskell's `bff`, is turned into two arguments for bff by currying. For the produced *put*, instead of quantifying over an `Eq`-constrained type variable, we use a Carrier type that is a parameter of the Agda module in which bff is defined. That is solely done for convenience — since a client of the module can pass an arbitrary type for that parameter, as long as a decidable equality is defined for that type, there is no less flexibility when applying the outcome *put*-function of bff than there is in the Haskell case. Another notable difference is that the final outcome is wrapped in a Maybe. The reason for this is that in Agda all functions must be total. So while the Haskell implementation fails with a runtime error if no suitable result can be produced by *put*, in Agda we instead need to explicitly signal error cases as special values. Finally, the vector lengths in the type of the produced *put*-function tell us about shape constraints. In fact, mismatches between expected shape (from the original view obtained from the original source) and actual shape (from the updated view) are one reason for runtime errors in the Haskell version of `bff`. In Agda, trying to combine a source s that has type Vec Carrier n for some natural number n with a view v that has any other type than Vec Carrier (getlen n), in particular one that has any other length than the expected getlen n, will not even be type-correct — so a possible runtime error has been turned into a static check.

The actual definition of bff is not much different than in Haskell. Using some functions from Agda standard libraries and some auxiliary functions we do not repeat from (Grohne, 2013) in full here, we arrive at:

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

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

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

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′
```

We do not explain all syntax used here, in particular the generalized form of pattern matching via **with**. Beside the fact that apart from the more informative types these function definitions are rather close to those from (Voigtländer, 2009), the more interesting aspect is anyway what we can *prove* about them.

4. PROVING CORRECTNESS

Voigtländer (2009) proves two theorems about `bff`, corresponding to GetPut and PutGet. In Agda, a theorem is represented/encoded as a type and a proof is a term that has that type. The two theorems as expressed in Agda are:

```
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
```

and:

[8]Data.Product

[9]We do not here consider the versions of `bff` that work with input functions of the type `forall a. Eq a => [a] -> [a]` or `forall a. Ord a => [a] -> [a]`.

```
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
```

Note how both are first "quantified" — since an argument type means a piece that the user of the theorem can choose freely as long as being type-correct — over the ingredients (a getlen and a get) that are the main inputs to bff. Then, theorem-1 expresses that for every s and every put obtained as bff get holds: put s (get s) ≡ just s, i.e., the here appropriate version of the GetPut law $put(s, get(s)) = s$. Similarly, theorem-2 expresses that for every s, v, u, if bff get s v ≡ just u (note that a precondition simply becomes a function argument whose type is a statement, and thus whose every value witness will be a proof object for that statement), then get u ≡ v. In other words, again for put obtained as bff get: if there is some u such that put s v ≡ just u, then get of that u is v. That of course corresponds to the PutGet law, $get(put(s, v)) = v$, conditioned by $put(s, v)$ actually being defined.

Complete proof objects for theorem-1 and theorem-2 are given in the Agda file `http://www.iai.uni-bonn.de/~jv/bx-project/fsbxia.agda` accompanying (Grohne, 2013). We will not give those proofs/terms here; the important thing is that they exist. What is interesting to record, of course, is what assumptions they depend on. The only dependency that is *not* proved within said formalization itself is the Vec variant of the free theorem for polymorphic functions on homogeneous lists. Instead, it is only postulated:

```
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)
```

where $\mathsf{map}_{\mathsf{Vec}}$ is from the standard library[7]. That is the natural transfer of the free theorem statement for lists from Wadler (1989) to the setting of vectors. Actually proving it in Agda as well would require techniques that are orthogonal to our consideration of the lens laws (Bernardy et al., 2012), so we opt for keeping it as a postulation here, just as the list version of that free theorem for Haskell was an assumption (by all beliefs of the Haskell community a very well-founded one) in the proofs of Voigtländer (2009). The important thing is that the proofs of theorem-1 and theorem-2 from $\mathsf{free\text{-}theorem}_{\mathsf{Vec}}$ are now fully machine-checked!

Those proofs themselves proceed via a series of lemmas, similarly as one would do on paper, but of course Agda is uncompromising in requiring an explicit argument for each step. There is no "this is obvious" or "left as an exercise to the reader" as in (Voigtländer, 2009) and other papers on semantic bidirectionalization and extensions thereof. Just to give a taste, here are statements that we encounter which correspond to Lemmas 1 and 2 of Voigtländer (2009):

```
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))
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
```

as well as how an induction proof in Agda looks like, for the former:[10]

```
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))) ∎
```

farming out to another auxiliary 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))
```

which in turn requires further inductions, etc. Something we do not dwell on here is the actual *process* of arriving at the proofs, but Grohne (2013) describes in detail how interactive proof construction works and how Agda lends a helping hand, while also requiring familiarization with certain idioms for effective formalization. This guidance should be helpful when embarking on a similar endeavor for correctness proofs of other techniques, or when further developing the provided formalization, to cover extensions of semantic bidirectionalization already presented in the literature or still to be explored.

5. SO WHAT?

We have arrived at formal proofs of GetPut and PutGet for the bidirectionalization technique from (Voigtländer, 2009). But we already knew, or at least very strongly believed, that the technique was correct beforehand. After all, the original paper did contain lemmas, theorems, and proofs that seemed acceptable to the community. So what have we actually gained?

Beside the reassuring feeling that comes with a machine-checked proof, the dependent types and formalization work bring concrete additional benefits in terms of better understanding of the formalized technique and its properties. We have already remarked on the fact that the Haskell version of `bff` can fail with a runtime error, and that one reason for

[10]The refl steps correspond to reflexivity of propositional equality ≡. It can be used when Agda is able to prove an equality by its built-in rewriting strategy based on function definitions. Such rewriting also happens silently, but of course always with Agda's correctness guarantee, in some other steps.

such failure is shape mismatches, and that the constraints on vector lengths in the Agda types we use prevent those. Actually, it was already informally observed in previous work for the Haskell version that only when the shapes of get s and v are the same is there any hope that put s v is defined, but the dependent types in the Agda version are both explicit and more rigorous about this.

And there is more. Even when the shapes are in the correct relationship, the put obtained as bff get can fail. After all, that is why we have wrapped the ultimate return type of bff in a Maybe. Such failure occurs when get duplicates some list entry from the source and the two copies in the view are updated to different values. On the other hand, if no duplication takes place, then bff should not end up returning nothing (thus signaling failure). In Agda, we can formalize this intuition based on the following predicate:

```
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)
```

What this definition says is that, trivially, the elements of the empty list are pairwise different, and the elements of a non-empty list are pairwise different if the head element is not contained in the tail and if, moreover, the elements of the tail are pairwise different. Based on All-different, Grohne (2013) proves a sufficient condition for when an assoc-call succeeds (i.e., for when there exists some h such that the result of assoc is just h rather than nothing):

```
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)
```

Moreover, he proves that if a certain assoc-call succeeds, then the put obtained as bff get succeeds:

```
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)
```

Combining different-assoc and lemma-assoc-enough, we learn that bff get s v succeeds, and thus the precondition of PutGet/theorem-2 is fulfilled, if

```
All-different (toList (get (enumerate s)))
```

holds, where

```
enumerate : {n : ℕ} → Vec Carrier n → Vec (Fin n) n
```

is a function that given a vector of length n produces a vector that corresponds to the list [0 , 1 , . . . , n-1]. Thus, we have formally established that a sufficient condition on get to guarantee that the dependently typed bff get always succeeds is what is called *semantically affine* in (Voigtländer et al., 2013).

Further exploration of semantic bidirectionalization techniques should also profit from the availability of a formalization. Indeed, such availability would have benefited us in the past. For example, the original paper (Voigtländer, 2009) proved GetPut and PutGet, but only claimed that a third law, PutPut, also holds. Later work (Foster et al., 2012) refactored the definition of `bff`, essentially by formulating it in terms of the constant-complement approach (Bancilhon and Spyratos, 1981), to make more apparent that PutPut indeed holds. But this refactoring required extra care and consideration to make sure that no other properties were destroyed. In fact, new arguments were needed for correctness of the refactored version. Of course, the same would have been the case if an Agda formalization of the original correctness arguments had already been available, but the dependent types and proof assistant would have provided a safety net, just as standard type systems provide a safety net when refactoring ordinary programs instead of programs and proofs in one go. Similarly, other and further variations of semantic bidirectionalization may profit now. It would be useful to first extend the formalization to treat data structures other than lists for get to operate on, for example trees; we foresee no real problems in doing so.

Finally, let us mention a promising new direction for bidirectionalization that uses dependent types not only for verification but for doing a better job at the bidirectionalization task itself. The idea here is to turn dependent types into a 'plug-in' in the sense of (Voigtländer et al., 2013). In brief, the variation of semantic bidirectionalization presented by Voigtländer et al. (2013) overcomes the limitation of only being able to handle shape-preserving updates. It does so by requiring that each invocation of `bff` is enriched by a 'shape bidirectionalizer', a function that performs well-behaved updates on an abstraction of sources and views to the shape level, for example list lengths. Several possibilities are discussed for solving the shape-level problem, ranging from requesting programmer input, over search and syntactic transformations, to bootstrapping semantic bidirectionalization for abstracted problems. All this happens in Haskell, but in Agda we have another resource for such plug-in techniques. Namely, we can turn to shape information that comes from the types. Specifically, the getlen functions already express relationships between source and view list lengths. Since the propagation direction needed for shape bidirectionalizer plug-ins is from views to sources, we would actually need at least a partial inverse of getlen. But with the rich expressiveness available at the type level in Agda, we could even explore different abstractions, be they general relations between source and view shapes, or functions in one or the other direction. We can also prove connections between these abstractions, and potentially move between them, depending on what is most convenient for a given get-function. As a very simple example of what we have in mind, consider the tail function with its canonical type in Agda:

```
tail : {α : Set} {n : ℕ} → Vec α (suc n) → Vec α n
tail (_ :: xs) = xs
```

The type does not only express that tail is only well-defined on non-empty lists, it also tells us in no uncertain terms that its input is always exactly one entry longer than its output (so suc acts as getlen^{-1} here). Concerning bidirectionality that tells us that if tail is get and the view list is changed to some new length, we know exactly what the new source length should be — exactly the information that a shape bidirectionalizer plug-in needs to provide, but now actually available statically by virtue of the very definition of get in a dependently typed language. We plan to develop a general technique from this idea, of course with Agda implementation and formalization going hand in hand.

References

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Proceedings of Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005. doi: 10.1007/11541868_4.

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.

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — Parametricity for dependent types. *Journal of Functional Programming*, 22(2):107–152, 2012. doi: 10.1017/S0956796812000056.

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`.

Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three complementary approaches to bidirectional programming. In *Spring School on Generic and Indexed Programming (SSGIP 2010), Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer, 2012. doi: 10.1007/978-3-642-32202-0_1.

Helmut Grohne. Formalizing semantic bidirectionalization in Agda. Master's thesis, University of Bonn, 2013. URL `http://www.iai.uni-bonn.de/~jv/bx-project/fsbxia-final.pdf`. Agda formalization available at `http://www.iai.uni-bonn.de/~jv/bx-project/fsbxia.agda`.

Kazutaka Matsuda and Meng Wang. Bidirectionalization for free with runtime recording: Or, a light-weight approach to the view-update problem. In *Proceedings of Principles and Practice of Declarative Programming*, pages 297–308. ACM, 2013. doi: 10.1145/2505879.2505888.

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.

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. Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. *Journal of Functional Programming*, 23(5):515–551, 2013. doi: 10.1017/S0956796813000130.

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

Meng Wang and Shayan Najd. Semantic bidirectionalization revisited. In *Proceedings of Partial Evaluation and Program Manipulation*. ACM, 2014. To appear.