JFP, 15 pages, 2021. © Cambridge University Press 2021
doi:10.1017/xxxxx

FUNCTIONAL PEARL

A well-known representation of monoids and its application to the function "vector reverse"

WOUTER SWIERSTRA
Utrecht University
(e-mail: w.s.swierstra@uu.nl)

Abstract

Vectors—or length-indexed lists—are classic example of a dependent type. Yet most tutorials stay clear of any function on vectors whose definition requires non-trivial equalities between natural numbers to type check. This pearl shows how to write functions, such as vector reverse, that rely on monoidal equalities to be type correct without having to write any additional proofs. These techniques can be applied to many other functions over types indexed by a monoid, written using an accumulating parameter, and even be used decide arbitrary equalities over monoids 'for free.'

1 Introduction

Many tutorials on programming with dependent types define the type of length-indexed lists, also known as *vectors*. Using a language such as Agda (Norell, 2007), we can write:

```
data Vec (a : Set) : Nat → Set where
  Nil  : Vec a Zero
  Cons : a → Vec a n → Vec a (Succ n)
```

Many familiar functions on lists can be readily adapted to work on vectors, such as concatenation:

```
vappend : Vec a n → Vec a m → Vec a (n + m)
vappend Nil         ys = ys
vappend (Cons x xs) ys = Cons x (vappend xs ys)
```

Here the definitions of both addition and concatenation proceed by induction on the first argument; this coincidence allows concatenation to type check, without having to write explicit proofs involving natural numbers. Programming languages such as Agda will happily expand definitions while type checking—but any non-trivial equality between natural numbers may require further manual proofs.

However, not all functions on lists are quite so easy to adapt to vectors. How should we reverse a vector? There is an obvious—but inefficient—definition.

```
snoc : Vec a n → a → Vec a (Succ n)
snoc Nil y           = Cons y Nil
snoc (Cons x xs) y = Cons x (snoc xs y)

slowReverse : Vec a n → Vec a n
slowReverse Nil            = Nil
slowReverse (Cons x xs) = snoc (slowReverse xs) x
```

The snoc function traverses a vector, adding a new element at its end. Repeatedly traversing the intermediate results constructed during reversal yields a function that is quadratic in the input vector's length. Fortunately, there is a well-known solution using an accumulating parameter, often attributed to Hughes (1986). If we try to implement this version of the reverse function on vectors, we get stuck quickly:

```
revAcc : Vec a n → Vec a m → Vec a (n + m)
revAcc Nil          ys = ys
revAcc (Cons x xs) ys = {revAcc xs (Cons x ys)}_0
    Goal: Vec a (Succ (n + m))
    Have: Vec a (n + Succ m)
```

Here we have highlighted the unfinished part of the program, followed by the type of the value we are trying to produce and the type of the expression that we have written so far. Each of these goals that appear in the text will be numbered, starting from 0 here. In the case for non-empty lists, the recursive call revAcc xs (Cons x ys) returns a vector of length $n + \mathsf{Succ}\ m$, whereas the function's type signature requires a vector of length $(\mathsf{Succ}\ n) + m$. Addition is typically defined by induction over its first argument, immediately producing an outermost successor when possible; correspondingly, the definition of vappend type checks directly—but revAcc does not.

We can remedy this by defining a variation of addition that mimics the accumulating recursion of the revAcc function:

```
addAcc : Nat → Nat → Nat
addAcc Zero      m = m
addAcc (Succ n) m = addAcc n (Succ m)
```

Using this accumulating addition, we can define the accumulating vector reversal function directly:

```
revAcc : Vec a n → Vec a m → Vec a (addAcc n m)
revAcc Nil          ys = ys
revAcc (Cons x xs) ys = revAcc xs (Cons x ys)
```

When we try to use the revAcc function to define the top-level vreverse function, however, we run into a new problem:

```
vreverse : Vec a n → Vec a n
vreverse xs = {revAcc xs Nil}_1
    Goal: Vec a n
    Have: Vec a (addAcc n Zero)
```

Again, the desired definition does not type check: revAcc xs Nil produces a vector of length addAcc n Zero, whereas a vector of length n is required. We could try another variation of addition that pattern matches on its second argument, but this will break the first clause of the revAcc function. To complete the definition of vector reverse, we can use an explicit proof to coerce the right-hand side, revAcc xs Nil, to have the desired length. To do so, we define an auxiliary function that coerces a vector of length n into a vector of length m, provided that we can prove that n and m are equal:

```
coerce-length : n ≡ m → Vec a n → Vec a m
coerce-length refl xs = xs
```

Using this function, we can now complete the definition of vreverse as follows:

```
vreverse : (n : Nat) → Vec a n → Vec a n
vreverse n xs = coerce-length proof (revAcc xs Nil)
  where
  proof : addAcc n Zero ≡ n
```

We have omitted the definition of proof—but we will return to this point in the final section.

This definition of vreverse is certainly correct—but the additional coercion will clutter any subsequent lemmas that refer to this definition. To prove any property of vreverse will require pattern matching on the proof to reduce—rather than reasoning by induction on the vector directly.

Unfortunately, it is not at all obvious how to complete this definition without such proofs. We seem to have reached an impasse: how can we possibly define addition in such a way that Zero is both a left *and* a right identity?

2 Monoids and endofunctions

The solution can also be found in Hughes's article, that explores using an alternative representation of lists known as *difference lists*. These difference lists identify a list with the partial application of the append function. Rather than work with natural numbers directly, we choose an alternative representation of natural numbers that immediately satisfies the desired monoidal equalities, representing a number as the partial application of addition.

```
DNat : Set
DNat = Nat → Nat
```

In what follows, we will refer to these functions Nat → Nat as *difference naturals*. We can readily define the following conversions between natural numbers and difference naturals:

```
⟦_⟧ : Nat → DNat
⟦ n ⟧ = λ m → m + n

reify : DNat → Nat
reify m = m Zero
```

We have some choice of how to define the reify function. As addition is defined by induction on the *first* argument, we define reify by applying Zero to its argument. This choice ensures that the desired 'return trip' property between our two representations of naturals holds definitionally:

```
reify-correct : ∀ n → reify ⟦ n ⟧ ≡ n
reify-correct n = refl
```

Note that we have chosen to use the type Nat → Nat here, but there is nothing specific about natural numbers in these definitions. These definitions can be readily adapted to work for *any* monoid—an observation we will explore further in later sections. Indeed, this is an instance of Cayley's theorem for groups (Armstrong, 1988, Chapter 8), or the Yoneda embedding more generally (Boisseau & Gibbons, 2018; Awodey, 2010), that establishes an equivalence between the elements of a group and the partial application of the group's multiplication operation.

While this fixes the conversion between numbers and their representation using functions, we still need to define the monoidal operations on this representation. Just as for difference lists, the zero and addition operation correspond to the identity function and function composition respectively:

```
zero : DNat
zero = λ x → x
_⊕_ : DNat → DNat → DNat
n ⊕ m = λ x → m (n x)
```

Somewhat surprisingly, all three monoid laws hold *definitionally* using this functional representation of natural numbers:

```
zero-right : ∀ x → reify x ≡ reify (x ⊕ zero)
zero-right = λ x → refl
zero-left  : ∀ x → reify x ≡ reify (zero ⊕ x)
zero-left  = λ x → refl
⊕-assoc : ∀ x y z → reify (x ⊕ (y ⊕ z)) ≡ reify ((x ⊕ y) ⊕ z)
⊕-assoc = λ x y z → refl
```

As adding zero corresponds to applying the identity function and addition is mapped to function composition, the proof of these equalities follows immediately after evaluating the left- and right-hand sides of the equality.

To convince ourselves that our definition of addition is correct, we should also prove the following lemma, stating that addition on 'difference naturals' and natural numbers agree for all inputs:

```
⊕-correct : ∀ n m → n + m ≡ reify (⟦ n ⟧ ⊕ ⟦ m ⟧)
```

After simplifying both sides of the equation, the proof boils down to the associativity of addition. Proving this requires a simple inductive argument, and does not hold definitionally. The reverse function we will construct, however, does not rely on this property.

3 Revisiting reverse

Before we try to redefine our accumulating reverse function, we need one additional auxiliary definition. Besides zero and the $\oplus$ operation on these naturals—we will need a successor function to account for new elements added to the accumulating parameter. Given that Cons constructs a vector of length Succ n for some n, our first attempt at defining the successor operation on difference naturals becomes:

```
succ : DNat → DNat
succ m = λ n → Succ (m n)
```

With this definition in place, we can now fix the type of our accumulating reverse function:

```
revAcc : (m : DNat) → Vec a n → Vec a (reify m) → Vec a (m n)
```

As we want to define revAcc by induction over its first argument vector, we choose that vector to have length n, for some natural number n. Attempting to pattern match on a vector of length reify m creates unification problems that Agda cannot resolve: it cannot decide which constructors of the Vec datatype can be used to construct a vector of length reify m. As a result, we index the first argument vector by a Nat; the second argument vector has length reify m, for some m : DNat. The length of the vector returned by revAcc is the sum of the input lengths—reify (⟦ n ⟧ $\oplus$ m)—which simplifies to m n. We can now attempt to complete the definition as follows:

```
revAcc m Nil          ys = ys
revAcc m (Cons x xs) ys = {revAcc (succ m) xs (Cons x ys)}2
    Goal:  Vec a (m (Succ n))
    Have: Vec a (Succ (m n))
```

Unfortunately, the desired definition does not type check. The right-hand side produces a vector of the wrong length. To understand why, compare the types of the goal and expression we have produced. Using this definition of succ creates an outermost successor constructor, hence we cannot produce a vector of the right type.

Let us not give up just yet. We can still redefine our successor operation as follows:

```
succ : DNat → DNat
succ m = λ n → m (Succ n)
```

This definition should avoid the problem that arises from the outermost Succ constructor that we observed previously. If we now attempt to complete the definition of revAcc, we encounter a different problem:

```
revAcc : (m : DNat) → Vec a n → Vec a (reify m) → Vec a (m n)
revAcc m Nil ys          = ys
revAcc m (Cons x xs) ys = revAcc (succ m) xs {Cons x ys}3
    Goal:  Vec a (m (Succ Zero))
    Have: Vec a (Succ (m Zero)
```

Once again, the problem lies in the case for Cons. We would like to make a tail recursive call on the remaining list xs, passing succ m as the length of the accumulating parameter. This call now type checks—as the desired length m (Succ n) and computed length (succ m) n coincide. The problem, however, lies in constructing the accumulating parameter to pass to the recursive call. The recursive call requires a vector of length m (Succ Zero), whereas the Cons constructor used here returns a vector of length Succ (m Zero).

We might try to define an auxiliary function, analogous to the Cons constructor:

```
cons : (m : DNat) → a → Vec a (reify m) → Vec a (reify (succ m))
```

If we try to define this function directly, however, we get stuck immediately. The type requires that we produce a vector of length, m (Succ Zero). Without knowing anything further about m, we cannot even decide if the vector should be empty or not. Fortunately, we *do* know more about the difference natural m in the definition of revAcc. Initially, our accumulator will be empty—hence m will be the identity function. In each iteration of revAcc, we will compose m with an additional succ until our input vector is empty.

If we assume we are provided with a cons function of the right type, we can complete the definition of vector reverse as expected:

```
revAcc : ∀ m → (∀ {k} → a → Vec a (m k) → Vec a ((succ m) k)) →
  Vec a n → Vec a (reify m) → Vec a (m n)
revAcc m cons Nil           acc = acc
revAcc m cons (Cons x xs) acc = revAcc (succ m) cons xs (cons x acc)
```

This definition closely follows our previous attempt. Rather than applying the Cons constructor, this definition uses the argument cons function to extend the accumulating parameter. Here the cons function is assumed to commute with the successor constructor and an arbitrary difference natural m. In the recursive call, the first argument vector has length n, whereas the second has length reify (succ m). As the cons parameter extends a vector of length m k *for any* k, we use it in our recursive call, silently incrementing the implicit argument passed to cons. In this way, we count down from n, the length of the first vector, whilst incrementing the difference natural m in each recursive call.

But how are we ever going to call this function? We have already seen that it is impossible to define the cons function in general. Yet we do not need to define cons for *arbitrary* values of m—we only ever call the revAcc function from the vreverse function with an accumulating parameter that is initially empty. As a result, we only need to concern ourselves with the case that m is zero—or rather, the identity function. When m is the identity function, the type of the cons function required simply becomes:

```
  ∀ {k} → a → Vec a k → Vec a (Succ k)
```

Hence, it suffices to pass the Cons constructor to revAcc after all:

```
vreverse : Vec a n → Vec a n
vreverse xs = revAcc zero Cons xs Nil
```

This completes the first proof-free reconstruction of vector reverse.

Correctness

Reasoning about this definition of vector reverse, however, is a rather subtle affair. Suppose we want to prove vreverse is equal to the quadratic slowReverse function from the introduction:

```
vreverse-correct : (xs : Vec a n) → vreverse xs ≡ slowReverse xs
```

If we try to prove this using induction on xs directly, we quickly get stuck in the case for non-empty vectors: we cannot use our induction hypothesis, as the definition of vreverse assumes that the accumulator is the empty vector. To fix this, we need to formulate and prove a more general statement about calls to revAcc with an *arbitrary* accumulator, corresponding to a lemma of the following form:

```
  revAcc m cons xs ys ≡ vappend (slowReverse xs) ys
```

Here the vappend function refers to the append on vectors, defined in the introduction. There is a problem, however, formulating such a lemma: the vappend function uses the usual addition operation in its type, rather than the 'difference addition' used by revAcc. As a result, the vectors on both sides of the equality sign have different types. To fix this, we need the following variant of vappend, where the length of the second vector is represented by a difference natural:

```
dappend : ∀ m → (cons : ∀ {k} → a → Vec a (m k) → Vec a ((succ m) k)) →
            Vec a n → Vec a (reify m) → Vec a (m n)
dappend m cons Nil          ys = ys
dappend m cons (Cons x xs) ys = cons x (dappend m cons xs ys)
```

Using this 'difference append' operation, we can now formulate and prove the following correctness property, stating that revAcc pushes all the elements of xs onto the accumulating parameter ys:

```
revAcc-correct : (m : Nat → Nat) (xs : Vec a n) (ys : Vec a (reify m))
  (cons : ∀ {k} → a → Vec a (m k) → Vec a ((succ m) k)) →
  revAcc m cons xs ys ≡ dappend m cons (slowReverse xs) ys
```

The proof itself proceeds by induction on the vector xs and requires a single auxiliary lemma relating dappend and snoc. Using revAcc-correct and the fact that Nil is the right-unit of dappend, we can now complete the proof of vreverse-correct.

4 Using a left fold

The version of vector reverse defined in the Agda standard library uses a left fold. In this section, we will reconstruct this definition. A first attempt might use the following type for the fold on vectors:

```
foldl : (b → a → b) → b → Vec a n → b
foldl step base Nil          = base
foldl step base (Cons x xs) = foldl step (step base x) xs
```

Unfortunately, we cannot define vreverse using this fold. The first argument, f, of foldl has type b → a → b; we would like to pass the flip Cons function as this first argument, but it has type Vec a n → a → Vec a (Succ n)—which will not type check as the first argument and return type are not identical. We can solve this, by generalising the type of this function slightly, indexing the return type b by a natural number:

```
foldl : (b : Nat → Set) → (∀ {k} → b k → a → b (Succ k)) → b Zero → Vec a n → b n
foldl b step base Nil          = base
foldl b step base (Cons x xs)  = foldl (b ∘ Succ) step (step base x) xs
```

At heart, this definition is the same as the one above. There is one important distinction: the return type changes in each recursive call by precomposing with the successor constructor. In a way, this ‘reverses’ the natural number, as the outermost successor is mapped to the innermost successor in the type of the result. The accumulating nature of the foldl is reflected in how the return type changes across recursive calls.

We can use this version of foldl to define a simple vector reverse:

```
vreverse : Vec a n → Vec a n
vreverse = foldl (Vec _) (λ xs x → Cons x xs) Nil
```

This definition does not require any further proofs: the calculation of the return type follows the exact same recursive pattern as the accumulating vector under construction.

The foldl function on vectors is a useful abstraction for defining accumulating functions over vectors. For example, as Kidney (2019) has shown we can define the convolution of two vectors in a single pass in the style of Danvy & Goldberg (2005):

```
convolution : ∀ (a b : Set) → (n : Nat) → Vec a n → Vec b n → Vec (a × b) n
convolution a b n = foldl (λ n → Vec b n → Vec (a × b) n)
                          (λ {k x (Cons y ys) → Cons (x , y) (k ys)})
                          (λ {Nil → Nil})
```

Monoids indexed by monoids

A similar problem—monoidal equalities in indices—shows up when trying to prove that vectors form a monoid. Where proving the monoidal laws for natural numbers or lists is a straightforward exercise for students learning Agda, vectors pose more of a challenge. Crucially, if the lengths of two vectors are not (definitionally) equal, the statement that the vectors themselves are equal is not even *type correct*. For example, given a vector xs : Vec a n, we might try to state the following equality:

```
xs ≡ xs ++ Nil
```

The vector on the left-hand side of the equality has type Vec a n, while the vector on the right-hand side has type Vec a $(n + 0)$. As these two types are not the same—the vectors have different lengths—the statement of this equality is not type correct.

For *difference vectors*, however, this is not the case. To illustrate this, we begin by defining the type of difference vectors as follows:

```
DVec : Set → DNat → Set
DVec a d = ∀ {n} → Vec a n → Vec a (d n)
```

We can then define the usual zero and addition operations on difference vectors as follows:

```
vzero : DVec a zero
vzero = λ x → x
_++_ : {n m : DNat} → (xs : DVec a n) → (ys : DVec a m) → DVec a (n ⊕ m)
xs ++ ys = λ env → ys (xs env)
```

Next we can formulate the monoidal equalities and establish that these all hold trivially:

```
vzero-left         : (xs : DVec a n) → (vzero ++ xs) ≡ xs
vzero-left xs      = refl
vzero-right        : (xs : DVec a n) → (xs ++ vzero) ≡ xs
vzero-right xs     = refl
++-assoc           : (xs : DVec a n) → (ys : DVec a m) → (zs : DVec a k) →
  (xs ++ (ys ++ zs)) ≡ (xs ++ (ys ++ zs))
++-assoc xs ys zs  = refl
```

We have elided some implicit length arguments that Agda cannot infer automatically, but it should be clear that the monoidal operations on difference vectors are no different from the difference naturals we saw in Section 2.

It is worth pointing out that using the usual definitions of natural numbers and additions, the latter two definitions would not hold—*a fortiori*, the statement of the properties vzero-right and ++-assoc would not even *type check*. Consider the type of vzero-right, for instance: formulating this property using natural numbers and addition would yield a vector on the left-hand side of the equation of length $n + 0$, whereas xs has length n. As the equality type can only be used to compare vectors of equal length, the statement of vzero-right would be type incorrect. Addressing this requires coercing the lengths of the vectors involved—as we did in the very first definition of vreverse in the introduction—that quickly spread throughout any subsequent definitions.

5 Indexing beyond natural numbers

In this section, we will explore another application of the Cayley representation of monoids. Instead of indexing by a natural number, this section revolves around computations indexed by lists.

We begin by defining a small language of boolean expressions:

```
data Expr (vars : List a) : Set where
  T F  : Expr vars
  Not  : Expr vars → Expr vars
  And  : Expr vars → Expr vars → Expr vars
  Or   : Expr vars → Expr vars → Expr vars
  Var  : x ∈ vars → Expr vars
```

The Expr data type has constructors for truth, falsity, negation, conjunction and disjunction. Expressions are parametrised by a list of variables, vars : List a for some type a : Set. While we could model a finite collection of variables using the well known Fin type, we choose a slightly different representation here—allowing us to illustrate how the Cayley representation can be used for other indices beyond natural numbers. Each Var constructor stores a proof, $x \in vars$, that is used to denote the particular named variable to which is being referred. The proofs, $x \in xs$, can be constructed using a pair of constructors, Top and Pop, that refer to the elements in the head and tail of the list respectively:

```
data _∈_ : a → List a → Set where
  Top : x ∈ (x :: xs)
  Pop : x ∈ xs → x ∈ (y :: xs)
```

Indexing expressions by the list of variables they may contain, allows us to write a *total* evaluation function. The key idea is that our evaluator is passed an environment assigning a boolean value to each variable in our list:

```
data Env : List a → Set where
  Nil  : Env []
  Cons : Bool → Env xs → Env (x :: xs)
```

The evaluator itself is easy enough to define; it maps each constructor of the Expr data type to its corresponding operation on booleans.

```
eval : Expr vars → Env vars → Bool
eval T            env = True
eval F            env = False
eval (Not e)      env = ¬ (eval e env)
eval (And e1 e2)  env = eval e1 env ∧ eval e2 env
eval (Or e1 e2)   env = eval e1 env ∨ eval e2 env
eval (Var x)      env = lookup env x
```

The only interesting case is the one for variables, where we call an auxiliary lookup function to find the boolean value associated with the given variable.

For a large fixed expression, however, we may not want to call eval over and over again. Instead, it may be preferable to construct a *decision tree* associated with a given expression. The decision tree associated with an expression is a perfect binary tree, where each node branches over a single variable:

```
data DecTree : List a → Set where
  Node : DecTree vars → (x : a) → DecTree vars → DecTree (x :: vars)
  Leaf : Bool → DecTree []
```

Given any environment, we can still 'evaluate' the boolean expression corresponding to the tree, using the environment to navigate to the unique leaf corresponding to the series of true-false choices for each variable:

```
treeval : DecTree xs → Env xs → Bool
treeval (Leaf x)    Nil              = x
```

```
treeval (Node l x r) (Cons True env)  = treeval l env
treeval (Node l x r) (Cons False env) = treeval r env
```

We would now like to write a function that converts a boolean expression into its decision tree representation, while maintaining the scope hygiene that our expression data type enforces. We could imagine trying to do so by induction on the list of free variables, repeatedly substituting the variables one by one:

```
makeDecTree : (vars : List a) → Expr vars → DecTree vars
makeDecTree []         e = Leaf (eval e empty)
makeDecTree (x :: vars) e =
  let l = makeDecTree vars (subst T x e) in
  let r = makeDecTree vars (subst F x e) in
  Node l r
```

But this is not entirely satisfactory: to prove this function correct, we would need to prove various lemmas relating substitution and evaluation; furthermore, this function is inefficient, as it repeatedly traverses the expression to perform substitutions.

Instead, we would like to define an accumulating version of makeDecTree, that carries around a (partial) environment of those variables on which we have already branched. As we shall see, this causes problems similar to those that we saw previously for reversing a vector. A first attempt might proceed by induction on the free variables in our expression, that have not yet been captured in our environment:

```
makeDecTreeAcc : (xs ys : List a) → Expr (xs ++ ys) → Env ys → DecTree xs
makeDecTreeAcc []        ys expr env = Leaf (eval expr env)
makeDecTreeAcc (x :: xs) ys expr env = Node l x r
  where
  l = makeDecTreeAcc xs (x :: ys) {expr}4 (Cons True env)
  r = makeDecTreeAcc xs (x :: ys) {expr}5 (Cons False env)
    Goal: Expr (xs ++ x :: ys)
    Have: Expr (x :: xs ++ ys)
```

This definition, however, quickly gets stuck. In the recursive calls, the environment has grown, but the variables in the expression and environment no longer line up. The situation is similar to the very first attempt at defining the accumulating vector reverse function: the usual definition of addition is unsuitable for defining functions using an accumulating parameter. Fortunately, the solution is to define a function revAcc, akin to the one defined for vectors, that operates on lists:

```
revAcc : List a → List a → List a
revAcc []        ys = ys
revAcc (x :: xs) ys = revAcc xs (x :: ys)
```

We can now attempt to construct the desired decision tree, using the revAcc function in the type indices, as follows:

```
makeDecTreeAcc : (xs ys : List a) → Expr (revAcc xs ys) → Env ys → DecTree xs
makeDecTreeAcc []        ys expr env = Leaf (eval expr env)
```

```
makeDecTreeAcc (x :: xs) ys expr env = Node l x r
  where
  l = makeDecTreeAcc xs (x :: ys) expr (Cons True env)
  r = makeDecTreeAcc xs (x :: ys) expr (Cons False env)
```

Although this definition now type checks, just as we saw for one of our previous attempts for revAcc, the problem arises once we try to call this function with an initially empty environment:

```
makeDecTree : (xs : List a) → Expr xs → DecTree xs
makeDecTree xs expr = makeDecTreeAcc xs [] {expr}_6 Nil

    Goal: Expr (revAcc xs [])
    Have: Expr xs
```

Calling the accumulating version fails to produce a value of the desired type—in particular, it produces a tree branching over the variables revAcc xs [] rather than xs. To address this problem, however, we can move from an environment indexed by a regular lists to one indexed by a difference list, accumulating the values of the variables we have seen so far:

```
DEnv : (List a → List a) → Set
DEnv f = ∀ {vars} → Env vars → Env (f vars)
```

Note that we use the Cayley representation of monoids in both the *type* index of and the *value* representing environments.

We can now complete our definition as expected, performing induction without ever having to prove a single equality about the concatenation of lists:

```
makeDecTreeAcc : (xs : List a) → (ys : List a → List a) →
  DEnv ys → Expr (ys xs) → DecTree xs
makeDecTreeAcc []        ys denv expr = Leaf (eval expr (denv Nil))
makeDecTreeAcc (x :: xs) ys denv expr = Node l x r
  where
  l = makeDecTreeAcc xs (ys ∘ (x :: _)) (denv ∘ Cons True)  expr
  r = makeDecTreeAcc xs (ys ∘ (x :: _)) (denv ∘ Cons False) expr
```

Finally, we can kick off our accumulating function with a pair of identity functions, corresponding to the zero elements of the list of variables that have been branched on and the difference environment:

```
makeDecTree : (xs : List a) → Expr xs → DecTree xs
makeDecTree xs e = makeDecTreeAcc xs id id e
```

Interestingly, the type signature of this top-level function does not mention the 'difference environment' or 'difference lists' at all.

Can we verify that definition is correct? The obvious theorem we may want to prove states the eval and treeval functions agree on all possible expressions:

```
correctness : ∀ vars (e : Expr vars) (env : Env vars) →
  eval e env ≡ treeval (makeDecTree vars e) env
```

A direct proof by induction quickly fails, as we cannot use our induction hypothesis; we can, however, prove a more general lemma that implies this result:

```
lemma : ∀ {xs : List a} {ys : List a → List a} →
  (denv : DEnv ys) (expr : Expr (ys xs)) (env : Env xs) →
  eval expr (denv env) ≡ treeval (makeDecTreeAcc xs ys denv expr) env
lemma denv expr Nil                = refl
lemma denv expr (Cons False env) = lemma (denv ∘ Cons False) expr env
lemma denv expr (Cons True  env) = lemma (denv ∘ Cons True)  expr env
```

The proof is reassuringly simple; it has the same accumulating structure as the inductive definitions we have seen.

6 Solving any monoidal equation

In this last section, we show how this technique of mapping monoids to their Cayley representation can be used to solve equalities between any monoidal expressions. To generalise the constructions we have seen so far, we define the following Agda record representing monoids:

```
record Monoid (a : Set) : Set where
  field zero       : a
        _⊕_        : a → a → a
        zero-left  : ∀ x → (zero ⊕ x) ≡ x
        zero-right : ∀ x → (x ⊕ zero) ≡ x
        ⊕-assoc    : ∀ x y z → (x ⊕ (y ⊕ z)) ≡ ((x ⊕ y) ⊕ z)
```

We can represent expressions built from the monoidal operations using the following data type, MExpr:

```
data MExpr (a : Set) : Set where
  Add  : MExpr a → MExpr a → MExpr a
  Zero : MExpr a
  Var  : a → MExpr a
```

If we have a suitable monoid in scope, we can evaluate a monoidal expression, MExpr, in the obvious fashion:

```
eval : MExpr a → a
eval (Add e₁ e₂) = eval e₁ ⊕ eval e₂
eval (Zero)      = zero
eval (Var x)     = x
```

This is, however, not the only way to evaluation such expressions. As we have already seen, we can also define a pair of functions converting a monoidal expression to its Cayley representation and back:

```
⟦_⟧ : MExpr a → (MExpr a → MExpr a)
⟦ Add m₁ m₂ ⟧ = λ y → ⟦ m₁ ⟧ (⟦ m₂ ⟧ y)
```

```
⟦ Zero ⟧        = λ y → y
⟦ Var x ⟧       = λ y → Add (Var x) y
reify : (MExpr a → MExpr a) → MExpr a
reify f = f Zero
```

Finally, we can *normalise* any expression by composing these two functions:

```
normalise : MExpr a → MExpr a
normalise m = reify ⟦ m ⟧
```

Crucially, we can prove that this normalise function preserves the (monoidal) semantics of our monoidal expressions:

```
soundness : ∀ (x : MExpr a) → eval (normalise x) ≡ eval x
```

Where the cases for Zero and Var are straightforward, the addition case is more interesting. This final case requires a pair of auxiliary lemmas that rely on the monoid equalities:

```
∀ x y → eval (normalise (Add x y)) ≡ eval (⟦ x ⟧ y)
∀ x y → eval (⟦ x ⟧ y) ≡ eval (Add x y)
```

Using transitivity, we can complete this last case of the proof.

Finally, we can use this soundness result to prove that two expressions are equal under evaluation, provided their corresponding normalised expressions are equal under evaluation:

```
solve : ∀ (x y : MExpr a) → eval (normalise x) ≡ eval (normalise y) → eval x ≡ eval y
```

What have we gained? On the surface, these general constructions may not seem particularly useful or exciting. Yet the solve function establishes that to prove *any* equality between two monoidal expressions, it suffices to prove that their normalised forms are equal. Yet—as we have seen previously—the monoidal equalities hold definitionally in our Cayley representation. As a result, the only 'proof obligation' we need to provide to the solve function will be trivial.

Lets consider a simple example to drive home this point. Once we have established that lists are a monoid, we can use the solve function to prove the following equality:

```
example : (xs ys zs : List a) → ((xs ++ []) ++ (ys ++ zs)) ≡ ((xs ++ ys) ++ zs)
example xs ys zs =
  let e₁ = Add (Add (Var xs) Zero) (Add (Var ys) (Var zs)) in
  let e₂ = Add (Add (Var xs) (Var ys)) (Var zs) in
  solve e₁ e₂ refl
```

To complete the proof, we only needed to find monoidal expression representing the left- and right-hand sides of our equation—and this can be automated using Agda's meta-programming features (Van Der Walt & Swierstra, 2012). The only remaining proof obligation—that is, the third argument to the solve function—is indeed trivial. In this style, we can automatically solve any equality that relies exclusively on the three defining properties of any monoid.

We can also show that natural numbers form a monoid under addAcc and Zero. Using the associated solver, we can construct the proof obligations associated with the very first version of vector reverse from our introduction:

```
vreverse : (n : Nat) → Vec a n → Vec a n
vreverse n xs = coerce-length proof (revAcc xs Nil)
  where
    proof = solve (Add (Var n) Zero) (Var n) refl
```

Even if the proof constructed here is a simple call to one of the monoidal identities, automating this proof lets us come full circle.

7 Discussion

I first learned of that monoidal identities hold definitionally for the Cayley representation of monoids from a message Alan Jeffrey (2011) sent to the Agda mailing list. Since then, this construction has been used (implicitly) in several papers (Allais *et al.*, 2017; McBride, 2011; Jaber *et al.*, 2016) and developments (Kidney, 2020; Ko, 2020)—but the works cited here are far from complete. The observation that the Cayley representation can be used to normalise monoidal expressions dates back at least to Beylin & Dybjer (1995), although it is an instance of the more general technique of normalisation by evaluation (Berger & Schwichtenberg, 1991).

The two central examples from this paper, reversing vectors and constructing trees, share a common structure. Each function uses an accumulating parameter, indexed by a monoid, but relies on the monoid laws to type check. To avoid using explicit equalities, we use the Cayley representation of monoids in the *index* of the *accumulating parameter*. In the base case, this ensures that we can safely return the accumulating parameter; similarly, when calling the accumulating function with an initially empty argument, the Cayley representation ensures that the desired monoidal property holds by definition. In our second example, we also use the Cayley representation in the *value* of the accumulating parameter; we could also use this representation in the definition of vreverse, but it does not make things any simpler. In general, this technique works provided we *only* rely on the monoidal properties. As soon as the type indices contain richer expressions, we will need to prove equalities and coerce explicitly—or better yet, find types and definitions that more closely follow the structure of the functions we intend to write.

Acknowledgements I would like to thank Guillaume Allais, Joris Dral, Jeremy Gibbons, Donnacha Oisín Kidney and the anonymous reviewers for their insightful feedback.

Conflicts of Interest. None

References

Allais, G., Chapman, J., McBride, C. and McKinna, J. (2017) Type-and-scope safe programs and their proofs. *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2017, p. 195–207.

Armstrong, M. A. (1988) *Groups and symmetry*. Undergraduate Texts in Mathematics. Springer.

Awodey, S. (2010) *Category theory*. Oxford Logic Guides, no. 49. Oxford University Press.

Berger, U. and Schwichtenberg, H. (1991) An inverse of the evaluation functional for typed λ-calculus. *Proceedings - Symposium on Logic in Computer Science* pp. 203 – 211.

Beylin, I. and Dybjer, P. (1995) Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. *International Workshop on Types for Proofs and Programs* pp. 47–61. Springer.

Boisseau, G. and Gibbons, J. (2018) What you needa know about Yoneda: profunctor optics and the Yoneda lemma (Functional Pearl). *Proceedings of the ACM on Programming Languages* **2**(ICFP):84.

Danvy, O. and Goldberg, M. (2005) There and back again. *Fundamenta Informaticae* **66**(4):397–413.

Hughes, R. J. M. (1986) A novel representation of lists and its application to the function "reverse". *Information processing letters* **22**(3):141–144.

Jaber, G., Lewertowski, G., Pédrot, P.-M., Sozeau, M. and Tabareau, N. (2016) The definitional side of the forcing. *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* pp. 367–376.

Jeffrey, A. (2011) *Associativity for free!* `https://lists.chalmers.se/pipermail/agda/2011/003420.html`. Email to the Agda mailing list; accessed March 18, 2021.

Kidney, D. O. (2019) *How to do Binary Random-Access Lists Simply*. `https://doisinkidney.com/posts/2019-11-02-how-to-binary-random-access-list.html`. Accessed May 29, 2020.

Kidney, D. O. (2020) *Trees indexed by a Cayley Monoid*. `https://doisinkidney.com/posts/2020-12-27-cayley-trees.html`. Accessed May 29, 2020.

Ko, J. (2020) *McBride's Razor*. `https://josh-hs-ko.github.io/blog/0010/`. Accessed May 29, 2020.

McBride, C. (2011) *Ornamental Algebras, Algebraic Ornaments*. University of Strathclyde.

Norell, U. (2007) *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology.

Van Der Walt, P. and Swierstra, W. (2012) Engineering proof by reflection in Agda. *Symposium on Implementation and Application of Functional Languages* pp. 157–173. Springer.