JFP, 13 pages, 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 tutorials stay well clear of any function on vectors whose definition requires non-trivial equalities between natural numbers to type check. This paper demonstrates 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 begin by defining 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)
```

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 in green, 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 + Succ m, whereas the function's type signature requires a vector of length (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 easily enough 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)
```

Once again, the obvious candidate 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. At this point, 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. Rather than work with natural numbers directly, we choose an alternative representation of natural numbers that immediately satisfies the desired monoidal equalities. Just as Hughes represents a list as the partial application of append, we can represent a number as the partial application of addition.

```
⟦_⟧ : Nat → (Nat → Nat)
⟦ n ⟧ = λ m → m + n
reify : (Nat → Nat) → Nat
reify f = f Zero
```

We have some choice of how to define the reify function. As addition is defined by induction on the *first* argument, we choose reify to partially apply the second 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 will explore further in Section 6. 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).

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

```
zero : Nat → Nat
zero = λ x → x
_⊕_ : (Nat → Nat) → (Nat → Nat) → (Nat → Nat)
f ⊕ g = λ x → g (f 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 is immediate.

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:

$\oplus\text{-correct} : \forall\, n\, m\, k \to [\![\, n + m \,]\!]\, k \equiv ([\![\, n \,]\!] \oplus [\![\, m \,]\!])\, k$

The proof relies on the associativity of addition; the definition of reverse we will construct will not use 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, we choose to define the following successor operation at first:

```
succ : (Nat → Nat) → (Nat → Nat)
succ f n = Succ (f n)
```

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

$\mathsf{revAcc} : (m : \mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Vec}\, a\, n \to \mathsf{Vec}\, a\, (\mathsf{reify}\, m) \to \mathsf{Vec}\, a\, (\mathsf{reify}\, ([\![\, n \,]\!] \oplus m))$

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 easily—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 : Nat → Nat. The length of the vector returned by revAcc is expressed using the $\oplus$ operator, in an attempt to avoid the problems we encountered in the introduction. 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: $\mathsf{Vec}\, a\, (\mathsf{reify}\, ([\![\, \mathsf{Succ}\, n \,]\!] \oplus m))$
Normalised Vec a (m (Succ n))
Have: $\mathsf{Vec}\, a\, (\mathsf{reify}\, ([\![\, n \,]\!] \oplus \mathsf{succ}\, m))$
Normalised Vec a (Succ (m n))

Unfortunately, the desired definition does not type check. While the right-hand side of the definition is type correct, it produces a vector of the wrong length. To understand why, compare the normalised 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 : (Nat → Nat) → (Nat → Nat)
succ f n = f (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 : Nat → Nat) → Vec a n → Vec a (reify m) → Vec a (reify (⟦ n ⟧ ⊕ m))
revAcc m Nil ys          = ys
revAcc m (Cons x xs) ys = revAcc (succ m) xs {Cons x ys}₃
    Goal: Vec a (reify (succ m))
Normalised Vec a (m (Succ Zero))
    Have: Vec a (Succ (reify m))
Normalised 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 reify ((Succ n) ⊕ m) and computed length reify (⟦ n ⟧ ⊕ succ m) coincide. The problem, however, lies in constructing the accumulating parameter to pass to the recursive call. The recursive call requires a vector of length reify (succ m), whereas the Cons constructor returns a vector of length Succ (reify m).

We seem to be no further than before. We might try to define an auxiliary function of the following type:

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

Unfortunately, there is no way to produce a vector of the desired length, m (Succ Zero), without knowing anything further about m. If we appeal to the reader's suspension of disbelief and pretend that we are provided with a cons function of the right type, we can complete the definition as expected:

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

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—and the Cons constructor suffices after all:

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

Note that this definition is only type correct because the equations reify $[\![n]\!] \equiv n$ and $[\![n]\!] \oplus \mathsf{zero} \equiv [\![n]\!]$ hold definitionally. A different choice of $[\![\_]\!]$ function, for example, mapping n to $\lambda\ m \rightarrow n + m$ would break the first property.

4 Using a left fold

The version of vector reverse defined in the Agda standard library, however, 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 $b \rightarrow a \rightarrow b$; we would like to pass the flip Cons function as this first argument, but it has type $\mathsf{Vec\ a\ n} \rightarrow a \rightarrow \mathsf{Vec\ a\ (Succ\ n)}$—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) → (∀ {n} → b n → a → b (Succ n)) → 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.

Reasoning about left folds

This definition does, however, have one notable drawback: it is rather difficult to prove properties of functions defined using foldl. In particular, we may want to try and prove that the definition of vreverse above and the quadratic version from the introduction produce identical results for all inputs:

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

While the base case for the empty list holds trivially, we immediately 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 always the empty vector, Nil. After processing the head of the vector, however, the accumulator will no longer be empty in subsequent recursive calls—and correspondingly we cannot use our induction hypothesis. Although this can be fixed—generalising the definition of vreverse to start with an arbitrary initial accumulating argument—doing so requires a very careful treatment of equality between vectors (of potentially different lengths) and exposes the hidden complexity behind this simple definition.

Foldl and foldr on vectors

The subtle nature of the left-fold on vectors becomes even more apparent when we define foldl in terms of foldr, a restricted version of the elimination principle of vectors where the return type may only depend on the length of the vector:

```
foldr : (b : Nat → Set) → (∀ {n} → a → b n → b (Succ n)) → b Zero → Vec a n → b n
foldr b c n Nil            = n
foldr b c n (Cons x xs)  = c x (foldr b c n xs)
```

Defining foldl in terms of foldr poses an interesting challenge. The definition in Haskell typically uses the foldr to construct a function, which is then applied to the initial value of the accumulator:

```
foldl :: (a → b → a) → a → [b] → a
foldl step base xs = foldr (λ x rec acc → step (rec acc) x) id xs base
```

How can we adapt this definition to work with vectors? In particular, we will need to account for the changes in size as we recurse over the argument vector and construct the resulting function.

The first choice we must make is the type of the argument b that we pass to foldr. We clearly want to accumulate a function of the form λ n → b ... → b .... The question is how to account for the natural numbers involved. One obvious choice for the type is:

```
(λ n → ∀ m → b m → b (n + m))
```

that is, given any initial accumulating value b m, we can use the n elements from our input vector to produce a value of type b (n + m). Once we have made this choice, the remainder of the function closely follows the Haskell implementation above:

```
foldl : (b : Nat → Set) → (∀ n → b n → a → b (Succ n)) → b Zero → Vec a n → b n
foldl b step base xs =
  let result = foldr (λ n → ∀ m → b m → b (n + m))
                     (λ x rec m acc → step _ (rec m acc) x)
                     (λ m x → x)
  in {result xs Zero base}4
    Goal: b n
    Have: b (n + Zero)
```

Unfortunately, we have run into a familiar problem: once we kick-off the foldl, we produce a value of type b (n + Zero) rather than the desired b n. To address this, we introduce an auxiliary function that counts using our difference naturals.

```
foldlAcc : (b : Nat → Set) → (m : Nat → Nat) →
  (step : ∀ n → b (m n) → a → b (m (Succ n))) →
  Vec a n → b (reify m) → b (m n)
foldlAcc b m step xs =
  foldr (λ k → b (reify m) → b (m k)) (λ x rec acc → step _ (rec acc) x) (λ x → x) xs
```

In essence, here we once again assume the existence of an 'impossible' step function for combining our recursive results that somehow commutes Succ and addition with the difference natural m in the arguments to b. When we call foldlAcc, however, we instantiate m to be the identity and the step function we are provided suffices:

```
foldl : (b : Nat → Set) → (∀ n → b n → a → b (Succ n)) → b Zero → Vec a n → b n
foldl b step base xs = foldlAcc b zero step xs base
```

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})
```

5 Beyond vectors

In this section, we will explore another application of this representation of monoids. We begin by defining a small language of boolean expressions:

```
data Expr (n : Nat) : Set where
  Var : Fin n → Expr n
  Not : Expr n → Expr n
  And : Expr n → Expr n → Expr n
  Or  : Expr n → Expr n → Expr n
```

The Expr data type has constructors for negation, conjunction and disjunction. Variables are represented using the finite type, Fin n, that has exactly n inhabitants.

Indexing expressions by the number of variables they contain, allows us to write a *total* evaluation function. The key idea is that our evaluator is passed an environment assigning a boolean to each of the n possible variables; we can represent this environment as a vector of booleans:

```
Env : Nat → Set
Env n = Vec Bool n
```

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 n → Env n → Bool
eval (Var x)      env = lookup env x
```

```
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
```

The only interesting case is the one for variables, where we lookup the value of a variable in the current environment.

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 with n variables is a perfect binary tree of depth n:

```
data DecTree : Nat → Set where
  Node : DecTree n → DecTree n → DecTree (Succ n)
  Leaf  : Bool → DecTree Zero
```

Given any environment, we can still 'evaluate' the boolean expression corresponding to the tree, using the environment to navigate to the designated leaf:

```
treeval : DecTree n → Env n → Bool
treeval (Leaf x)   Nil                  = x
treeval (Node l r) (Cons True env)  = treeval l env
treeval (Node l r) (Cons False env) = treeval r env
```

We 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 number of free variables, repeatedly substituting the variables one by one:

```
makeDecTree : (n : Nat) → Expr n → DecTree n
makeDecTree Zero      e = evaluate e Nil
makeDecTree (Succ k) e =
  let l = makeDecTree k (subst True e) in
  let r = makeDecTree k (subst False e) in
  Node l r
```

But this is slightly unsatisfactory: to prove this function correct, we would need to prove various lemmas about substitutions; it 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 number of free variables in our expression, that have not yet captured in our environment:

```
makeDecTreeAcc : (n m : Nat) → Expr (n + m) → Env m → DecTree n
makeDecTreeAcc Zero      m expr env = Leaf (eval expr env)
makeDecTreeAcc (Succ k) m expr env = Node l r
  where
```

```
  l = makeDecTreeAcc k (Succ m) {expr}4 (Cons True env)
  r = makeDecTreeAcc k (Succ m) {expr}5 (Cons False env)
    Goal: Expr (k + Succ m)
    Have: Expr (Succ (k + m))
```

This definition, however, quickly gets stuck. In the recursive calls, the number of variables in the environment grows, but this growth is not captured in the type of the corresponding expression. The situation is similar to the very first attempt at defining the accumulating vector reverse function, revAcc: the usual definition of addition is unsuitable for defining functions using an accumulating parameter.

To remedy this, we could use the accumulating version of addition instead:

```
makeTreeAcc : (n m : Nat) → Expr (addAcc n m) → Env m → DecTree n
makeTreeAcc Zero     m expr env = Leaf (eval expr env)
makeTreeAcc (Succ n) m expr env = Node l r
  where
  l = makeTreeAcc n (Succ m) expr (Cons True env)
  r = makeTreeAcc n (Succ m) 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 it:

```
makeDecTree : (n : Nat) → Expr n → DecTree n
makeDecTree n expr = makeTreeAcc n Zero {expr}6 Nil
    Goal: Expr (addAcc n Zero)
    Have: Expr n
```

Just as we saw previously, calling the accumulating version fails to produce a value of the desired type—in particular, it produces a tree of depth addAcc n Zero rather than depth n. To address this problem, however, we can move from regular vectors to 'difference vectors' that accumulate the values of the variables we have seen so far:

```
DEnv : (Nat → Nat) → Set
DEnv m = ∀ {n} → Env n → Env (m n)
```

Note that we use the Cayley representation of monoids in both the *type* and the *value* associated with these difference vectors.

We can now complete our definition as expected, performing straightforward induction without ever having to prove a single equality between natural numbers:

```
makeTreeAcc : ∀ n m → DEnv m → Expr (reify (⟦ n ⟧ ⊕ m)) → DecTree n
makeTreeAcc Zero     m denv e = Leaf (eval e (denv Nil))
makeTreeAcc (Succ n) m denv e = Node l r
  where
  l = makeTreeAcc n (succ m) (denv · Cons True) e
  r = makeTreeAcc n (succ m) (denv · Cons False) e
```

Finally, we can kick off our accumulating function with a pair of identity functions, corresponding to the zero elements of the natural numbers and lists:

```
makeDecTree : (n : Nat) → Expr n → DecTree n
makeDecTree n e = makeTreeAcc n zero (λ env → env) e
```

Interestingly, the type signature of this top-level function does not mention the ‘difference naturals’ 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 : ∀ n (e : Expr n) (env : Env n) →
  eval e env ≡ treeval (makeDecTree n e) env
```

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

```
correctnessAux : ∀ n m (denv : DEnv m) (e : Expr (reify (⟦ n ⟧ ⊕ m))) (env : Env n) →
  eval e (denv env) ≡ treeval (makeTreeAcc n m denv e) env
```

This proof of this lemma is entirely straightforward.

Monoids indexed by monoids

Where proving the monoidal laws for natural numbers or lists is a straightforward exercise for students learning Agda, the monoidal laws for vectors are 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 our difference vectors, however, this is not the case. Just as we saw previously for the difference natural numbers, we can show that all the desired monoidal equalities hold *definitionally*.

To establish this, we begin by defining the monoidal operations on our difference vectors:

```
vzero : DEnv zero
vzero = λ x → x

_++_ : (xs : DEnv n) → (ys : DEnv m) → DEnv (n ⊕ m)
xs ++ ys = λ env → ys (xs env)
```

We have elided some implicit 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. Once again, we can formulate the monoidal equalities and establish that these all hold trivially.

```
vzero-left        : (xs : DEnv n) → (vzero ++ xs) ≡ xs
vzero-left xs     = refl

vzero-right       : (xs : DEnv n) → (xs ++ vzero) ≡ xs
vzero-right xs    = refl

++-assoc          : (xs : DEnv n) → (ys : DEnv m) → (zs : DEnv k) →
  (xs ++ (ys ++ zs)) ≡ (xs ++ (ys ++ zs))
++-assoc xs ys zs = refl
```

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
```

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 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 metaprogramming 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 a monoid.

7 Discussion

I first learned of that the 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) and developments (Kidney, 2020; Ko, 2020)—but the works cited here are far from complete. The observation that the Cayley representation 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).

Acknowledgements I would like to thank Guillaume Allais, Joris Dral, Jeremy Gibbons, and Donnacha Oisín Kidney for their insightful feedback on an early version of this paper.

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. Association for Computing Machinery.

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.

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.