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

A pearl for JFP; presented at ICFP

Wouter Swierstra

Utrecht University

What is a definition?

definition, n.

A precise statement of the essential nature of a thing; a statement or form of words by which anything is defined.

Natural numbers and addition

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

Natural numbers and addition

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

_+_ : ℕ → ℕ → ℕ
zero     + m = m
(succ k) + m = succ (k + m)
```

Vectors

```
data Vec (A : Set) : ℕ → Set where
  nil   : Vec A zero
  cons  : A → Vec A n → Vec A (succ n)
```

Vectors

```
data Vec (A : Set) : ℕ → Set where
  nil   : Vec A zero
  cons  : A → Vec A n → Vec A (succ n)

append : Vec A n → Vec A m → Vec A (n + m)
append nil         ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

Vectors

```
data Vec (A : Set) : ℕ → Set where
  nil   : Vec A zero
  cons  : A → Vec A n → Vec A (succ n)

append : Vec A n → Vec A m → Vec A (n + m)
append nil          ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

Why does this typecheck?

Why does this type check?

```
append : Vec A n → Vec A m → Vec A (n + m)
append nil ys          = {ys}
```

Goal:	`Vec A (zero + m)`
Have:	`Vec A m`

By definition, `zero + m` is equal to `m`.

Why does this type check?

```
append : Vec A n → Vec A m → Vec A (n + m)
append (cons x xs) ys = {cons x (append xs ys)}
```

Goal:	`Vec A ((succ k) + m)`
Have:	`Vec A (succ (k + m))`

By definition, `(succ k) + m` is equal to `succ (k + m)`.

Why does this type check?

```
append : Vec A n → Vec A m → Vec A (n + m)
append (cons x xs) ys = {cons x (append xs ys)}
```

Goal:	`Vec A ((succ k) + m)`
Have:	`Vec A (succ (k + m))`

By definition, `(succ k) + m` is equal to `succ (k + m)`.

The inductive structure of addition and append line up precisely.

Why does this type check?

```
append : Vec A n → Vec A m → Vec A (n + m)
append (cons x xs) ys = {cons x (append xs ys)}
```

Goal:	`Vec A ((succ k) + m)`
Have:	`Vec A (succ (k + m))`

By definition, `(succ k) + m` is equal to `succ (k + m)`.

The inductive structure of addition and append line up precisely.

The *only* equalities we get 'for free' are those that hold definitionally.

Vector reverse

```
snoc : Vec A n → A → Vec A (succ n)
snoc nil y         = cons y nil
snoc (cons x xs) y = cons x (snoc y xs)

reverse : Vec A n → Vec A n
reverse nil         = nil
reverse (cons x xs) = snoc (reverse xs) x
```

Vector reverse

```
snoc : Vec A n → A → Vec A (succ n)
snoc nil y          = cons y nil
snoc (cons x xs) y = cons x (snoc y xs)

reverse : Vec A n → Vec A n
reverse nil           = nil
reverse (cons x xs) = snoc (reverse xs) x
```

Taking quadratic time to reverse a list is bad...

A NOVEL REPRESENTATION OF LISTS AND ITS APPLICATION TO THE FUNCTION "REVERSE"

R. John Muir HUGHES *

Institute for Dataprocessing, Chalmers Technical University, 41296 Göteborg, Sweden

Communicated by L. Boasson
Received November 1984
Revised May 1985

A representation of lists as first-class functions is proposed. Lists represented in this way can be appended together in constant time, and can be converted back into ordinary lists in time proportional to their length. Programs which construct lists using append can often be improved by using this representation. For example, naive reverse can be made to run in linear time, and the conventional 'fast reverse' can then be derived easily. Examples are given in KRC (Turner, 1982), the notation being explained as it is introduced. The method can be compared to Sleep and Holmström's proposal (1982) to achieve a similar effect by a change to the interpreter.

A ~~different~~ difference list reversal

```
reverse-list : List A → List A
reverse-list xs = go xs nil
  where
  go : List A → (List A → List A)
  go nil          = id
  go (cons x xs)  = go xs . cons x
```

We can represent a list as a function from lists to lists, appending its elements to argument.

Eta expanding this definition gives rise to the ‘usual’ definition using an accumulating parameter.

A ~~different~~ difference list reversal

```
reverse-list : List A → List A
reverse-list xs = go xs nil
  where
  go : List A → (List A → List A)
  go nil          = id
  go (cons x xs)  = go xs . cons x
```

We can represent a list as a function from lists to lists, appending its elements to argument.

Eta expanding this definition gives rise to the ‘usual’ definition using an accumulating parameter.

`\begin{shameless-self-promotion}` And if you want to know how to reverse a list in constant *space*, don’t miss Anton’s talk tomorrow. `\end{shameless-self-promotion}`

Reversing vectors

```
reverse : Vec A n → Vec A m → Vec A (n + m)
reverse nil         acc  = acc
reverse (cons x xs) acc  = {!reverse xs (cons x acc)!}
```

Error

Goal: `Vec A ((succ k) + m))`

Have: `Vec A (k + (succ m))`

Reversing vectors

```
reverse : Vec A n → Vec A m → Vec A (n + m)
reverse nil         acc  = acc
reverse (cons x xs) acc  = {!reverse xs (cons x acc)!}
```

Error

Goal:	`Vec A ((succ k) + m))`
Have:	`Vec A (k + (succ m))`

This definition of reverse is a tail-recursive, using accumulating parameter – the structure is very differently from addition!

What definition of addition lines up with this reversal function?

Accumulating addition

```
addAcc : ℕ → ℕ → ℕ
addAcc zero     m = m
addAcc (succ k) m = addAcc k (succ m)

reverseAcc : Vec A n → Vec A m → Vec A (addAcc n m)
reverseAcc nil         acc  = acc
reverseAcc (cons x xs) acc  = reverseAcc xs (cons x acc)
```

Not quite...

```
reverse : Vec A n → Vec A n
reverse xs = {!reverseAcc xs nil!}
```

Error	
Goal:	`Vec A n`
Have:	`Vec A (addAcc n zero)`

Not quite...

```
reverse : Vec A n → Vec A n
reverse xs = {!reverseAcc xs nil!}
```

Error	
Goal:	`Vec A n`
Have:	`Vec A (addAcc n zero)`

Remember the definition of addAcc:

```
addAcc : ℕ → ℕ → ℕ
addAcc zero     m = m
addAcc (succ k) m = addAcc k (succ m)
```

Showing Agda who's the boss

```
reverse : Vec A n → Vec A n
reverse xs = coerceVec proof (reverseAcc xs nil)
  where
  proof : addAcc n zero ≡ n
  coerceVec : n ≡ m → Vec A n → Vec A m
```

Showing Agda who's the boss

```
reverse : Vec A n → Vec A n
reverse xs = coerceVec proof (reverseAcc xs nil)
  where
  proof : addAcc n zero ≡ n
  coerceVec : n ≡ m → Vec A n → Vec A m
```

A NOVEL REPRESENTATION OF LISTS AND ITS APPLICATION TO THE FUNCTION "REVERSE"

R. John Muir HUGHES *

Institute for Dataprocessing, Chalmers Technical University, 41296 Göteborg, Sweden

Communicated by L. Boasson
Received November 1984
Revised May 1985

A representation of lists as first-class functions is proposed. Lists represented in this way can be appended together in constant time, and can be converted back into ordinary lists in time proportional to their length. Programs which construct lists using append can often be improved by using this representation. For example, naive reverse can be made to run in linear time, and the conventional 'fast reverse' can then be derived easily. Examples are given in KRC (Turner, 1982), the notation being explained as it is introduced. The method can be compared to Sleep and Holmström's proposal (1982) to achieve a similar effect by a change to the interpreter.

Difference naturals

```
DNat : Set
DNat = Nat → Nat

⟦_⟧   : ℕ → DNat
⟦ n ⟧ = λ m → m + n

reify : DNat → ℕ
reify dn = dn zero
```

Difference naturals are monoidal

```
dzero : DNat
dzero = id

_+_ : DNat → DNat → DNat
dn + dm = dm . dn
```

Difference naturals are monoidal

```
dzero : DNat
dzero = id

_+_ : DNat → DNat → DNat
dn + dm = dm . dn
```

And three properties:

```
unit-right  : ∀ dn → reify dn ≡ reify (dn + dzero)
unit-left   : ∀ dn → reify dn ≡ reify (dzero + dn)
+-assoc     : ∀ dn dm dk → reify (dn + (dm + dk)) ≡ reify ((dn + dm) + dk)
```

Difference naturals are monoidal

```
dzero : DNat
dzero = id

_+_ : DNat → DNat → DNat
dn + dm = dm . dn
```

And three properties:

```
unit-right  : ∀ dn → reify dn ≡ reify (dn + dzero)
unit-left   : ∀ dn → reify dn ≡ reify (dzero + dn)
+-assoc     : ∀ dn dm dk → reify (dn + (dm + dk)) ≡ reify ((dn + dm) + dk)
```

Each of these properties holds **by definition**.

Proof by expanding definitions

```
reify dn
   = -- definition of reify
dn zero
   = -- definition of id
dn (id zero)
   = -- definition of reify
reify (dn . id)
   = -- definition of dzero
reify (dn . dzero)
   = -- definition of addition
reify (dzero + dn)
```

Indexing with difference naturals

Can we define vector reverse using difference naturals?

We can almost complete the desired definition...

```
revAcc : (dm : DNat) → Vec A n → Vec A (reify dm) → Vec A (dm n)
revAcc dm nil          acc  = acc
revAcc dm (cons x xs) acc  = revAcc (dsucc dm) xs {!cons x acc!}
```

Goal:	`Vec A (dm (succ zero))`
Have:	`Vec A (succ (dm zero)`

We are trying to extend the accumulator using cons – but we don't know how `dm` and cons interact.

The type of `cons`

Adding new elements to a vector:

```
cons : ∀ n → A → Vec A n → Vec A (succ n)
```

But we would like to accumulate elements as follows:

```
dcons : ∀ n dm → A → Vec A (dm n) → Vec A (dm (succ n))
```

The type of `cons`

Adding new elements to a vector:

```
cons : ∀ n → A → Vec A n → Vec A (succ n)
```

But we would like to accumulate elements as follows:

```
dcons : ∀ n dm → A → Vec A (dm n) → Vec A (dm (succ n))
```

- But when we kick off the computation, `dm` is the identity function - `cons` would suffice.

The type of `cons`

Adding new elements to a vector:

```
cons : ∀ n → A → Vec A n → Vec A (succ n)
```

But we would like to accumulate elements as follows:

```
dcons : ∀ n dm → A → Vec A (dm n) → Vec A (dm (succ n))
```

- But when we kick off the computation, `dm` is the identity function - `cons` would suffice.
- In each recursive step, we increment `dm` and decrement `n` - allowing us to (re)use `cons`.

Vector reverse

```
revAcc :
  ∀ dm → (∀ k → A → Vec A (dm k) → Vec A ((dsucc dm) k)) →
  Vec A n → Vec A (reify dm) → Vec A (dm n)
revAcc dm dcons nil          acc  = acc
revAcc dm dcons (cons x xs)  acc  = revAcc (dsucc m) dcons xs (dcons x acc)

reverse : Vec A n → Vec A n
reverse xs = revAcc dzero cons xs nil
```

Vector reverse

```
revAcc :
  ∀ dm → (∀ k → A → Vec A (dm k) → Vec A ((dsucc dm) k)) →
  Vec A n → Vec A (reify dm) → Vec A (dm n)
revAcc dm dcons nil          acc  = acc
revAcc dm dcons (cons x xs)  acc  = revAcc (dsucc m) dcons xs (dcons x acc)

reverse : Vec A n → Vec A n
reverse xs = revAcc dzero cons xs nil
```

Using foldl

Functions with accumulating arguments can be written in terms of left folds:

```
reverse-list : List A → List A
reverse-list = foldl (flip cons) nil
  where
  foldl : (B → A → B) → B → List A → B
```

Why won't this work for vectors?

Left folding vectors

```
reverse-vec : Vec A n → Vec A n
reverse-vec = foldl (flip {!cons!}) {!nil!}
```

Goal: `A → Vec A n → Vec A n`
Have: `A → Vec A n → Vec A (succ n)`

Left folding vectors

Generalise `foldl` to work over a `ℕ` indexed `B`:

```
foldl-vec : (B : ℕ → Set) → (B k → A → B (succ k)) → B zero → Vec A n → B n
foldl-vec B step acc nil          = acc
foldl-vec B step acc (cons x xs)  = foldl-vec (B ∘ succ) step (step acc x) xs
```

The second case is not so obvious…

It counts down over (by induction on `xs`) and up (by *precomposing* with `succ`) at the same time!

Left folding vectors

Generalise `foldl` to work over a `ℕ` indexed `B`:

```
foldl-vec : (B : ℕ → Set) → (B k → A → B (succ k)) → B zero → Vec A n → B n
foldl-vec B step acc nil          = acc
foldl-vec B step acc (cons x xs)  = foldl-vec (B ∘ succ) step (step acc x) xs
```

The second case is not so obvious...

It counts down over (by induction on `xs`) and up (by *precomposing* with `succ`) at the same time!

```
reverse : Vec A n → Vec A n
reverse = foldl-vec (Vec A) (flip cons) nil
```

But wait... there's more!

There is nothing particular about natural numbers.

The *Cayley representation* of monoids as endofunctions works for *any* monoid – it's not quite as novel as the title of Hughes's paper suggests.

Example: indexing a (decision) tree by a list of variables in scope.

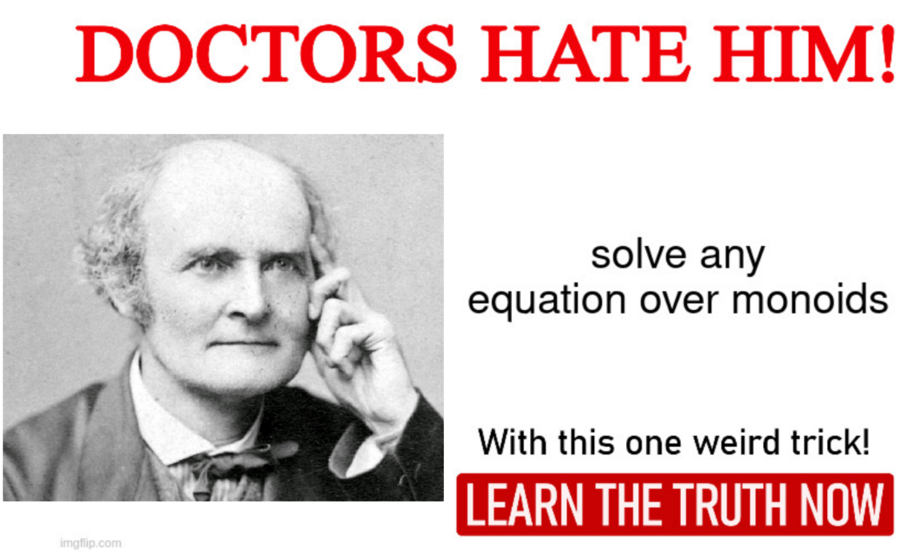
But wait… there's more!

There is nothing particular about natural numbers.

The *Cayley representation* of monoids as endofunctions works for *any* monoid – it's not quite as novel as the title of Hughes's paper suggests.

Example: indexing a (decision) tree by a list of variables in scope.

But if we can get the monoidal equalities to hold definitionally…


DOCTORS HATE HIM!
solve any
equation over monoids
With this one weird trick!
LEARN THE TRUTH NOW
imgflip.com

Proof sketch - part 0

Suppose we fix `A : Set` as (the carrier of) a monoid.

The monoidal expressions over `A` are given by:

```
data Expr : Set where
  _⊕_    : Expr A → Expr A → Expr A
  zero   : Expr A
  var    : A → Expr A
```

We can evaluate these expressions readily enough:

```
eval : Expr A → A
```

Proof sketch - part `succ zero`

We can define the mappings to/from their Cayley representation:

```
⟦_⟧    : Expr A → (Expr A → Expr A)
reify  : (Expr A → Expr A) → Expr A
```

Proof sketch - part `succ zero`

We can define the mappings to/from their Cayley representation:

```
⟦_⟧     : Expr A → (Expr A → Expr A)
reify   : (Expr A → Expr A) → Expr A
```

And we can use these to normalise any expression:

```
normalise : Expr A → Expr A
normalise e = reify ⟦ e ⟧
```

Proof sketch - part `succ (succ zero))`

We need to prove one lemma:

```
soundness : (e : Expr a) → eval (normalise e) ≡ eval e
```

Proof sketch - part `succ (succ zero)`

We need to prove one lemma:

```
soundness : (e : Expr a) → eval (normalise e) ≡ eval e
```

And use this to write our monoid solver:

```
solve : (l r : Expr A)
    -- both sides of an equation
  → eval (normalise l) ≡ eval (normalise r)
    -- hopefully just refl
  → eval l ≡ eval r
```

Proof sketch - part (succ (succ (succ zero)))

To call our solver - we only need to 'quote' the two sides of the equality:

```
example : (xs ys zs : List A) →
 ((xs ++ []) ++ (ys ++ zs))  ≡  ((xs ++ ys) ++ zs )
example xs ys zs =
    let e₁ = (var xs ⊕ zero) ⊕ (var ys ⊕ var zs) in
    let e₂ = (var xs ⊕ var ys) ⊕ var zs in
    solve e₁ e₂ refl
```

The quoting can be automated using Agda's reflection mechanism.

Back to the beginning

This construction works for *any* monoid...

In particular, for the natural numbers using accumulating addition.

Back to the beginning

This construction works for *any* monoid...

In particular, for the natural numbers using accumulating addition.

```
reverse : Vec A n → Vec A n
reverse xs = coerceVec proof (reverseAcc xs nil)
  where
  proof : addAcc n zero ≡ n
  proof = solve (var n ⊕ zero) (var n) refl
```

Recap

- The Cayley representation of a monoid satisfies the monoid laws by definition.

Recap

- The Cayley representation of a monoid satisfies the monoid laws by definition.
- This observation may be useful when writing functions accumulating monoid-indexed results (depending on your tolerance for complicated type signatures).

Recap

- The Cayley representation of a monoid satisfies the monoid laws by definition.
- This observation may be useful when writing functions accumulating monoid-indexed results (depending on your tolerance for complicated type signatures).
- We can use this to write a monoid solver for equations that follow (exclusively) from the monoidal identities.

Thank you!