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 \( \mathbb{N} : \text{Set} \) where

\[
\begin{align*}
\text{zero} & : \mathbb{N} \\
\text{succ} & : \mathbb{N} \to \mathbb{N}
\end{align*}
\]

\[
\begin{align*}
+ & : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\text{zero} + m & = m \\
(\text{succ } k) + m & = \text{succ } (k + m)
\end{align*}
\]
data Vec (A : Set) : N → Set where
  nil       : Vec A zero
  cons      : A → Vec A n → Vec A (succ n)
Vectors

```haskell
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)
```
V
def 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?

\[
\begin{align*}
\text{append} & : \text{Vec } A \ n \rightarrow \text{Vec } A \ m \rightarrow \text{Vec } A \ (n + m) \\
\text{append } (\text{cons } x \ xs) \ ys & = \{\text{cons } x \ (\text{append } xs \ ys)\} \\
\end{align*}
\]

\underline{Goal:} \quad \text{Vec } A \ ((\text{succ } k) + m) \\
\underline{Have:} \quad \text{Vec } A \ (\text{succ } (k + m))

By definition, \((\text{succ } k) + m\) is equal to \(\text{succ } (k + m)\).
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?

\[
\text{append : } \textbf{Vec} \ A \ n \rightarrow \textbf{Vec} \ A \ m \rightarrow \textbf{Vec} \ A \ (n + m)
\]

\[
\text{append } (\textbf{cons} \ x \ \textbf{xs}) \ \textbf{ys} = \{\textbf{cons} \ x \ (\text{append} \ \textbf{xs} \ \textbf{ys})\}
\]

---

**Goal:** \(\textbf{Vec} \ A \ ((\textbf{succ} \ k) + m)\)

**Have:** \(\textbf{Vec} \ A \ (\textbf{succ} \ (k + m))\)

---

*By definition,* \((\textbf{succ} \ k) + m\) is equal to \(\textbf{succ} \ (k + m)\).

The inductive structure of addition and append line up precisely.

The *only* equalities we get ‘for free’ are those that hold definitionally.
Vector reverse

\[
\text{sno}c : \text{Vec} \ A \ n \rightarrow A \rightarrow \text{Vec} \ A \ (\text{succ} \ n)
\]
\[
\text{sno}c \ \text{nil} \ y \ = \ \text{cons} \ y \ \text{nil}
\]
\[
\text{sno}c \ (\text{cons} \ x \ \text{x}s) \ y \ = \ \text{cons} \ x \ (\text{sno}c \ y \ \text{x}s)
\]

\[
\text{re}verse : \text{Vec} \ A \ n \rightarrow \text{Vec} \ A \ n
\]
\[
\text{re}verse \ \text{nil} \ = \ \text{nil}
\]
\[
\text{re}verse \ (\text{cons} \ x \ \text{x}s) = \text{sno}c \ (\text{re}verse \ \text{x}s) \ x
\]

Taking quadratic time to reverse a list is bad…
Vector reverse

\[
\text{snoc} : \text{Vec } A \ n \rightarrow A \rightarrow \text{Vec } A \ (\text{succ } n)
\]
\[
\text{snoc } \text{nil } y = \text{cons } y \text{ nil}
\]
\[
\text{snoc } (\text{cons } x \ x s) \ y = \text{cons } x \ (\text{snoc } y \ x s)
\]

\[
\text{reverse} : \text{Vec } A \ n \rightarrow \text{Vec } A \ n
\]
\[
\text{reverse } \text{nil} = \text{nil}
\]
\[
\text{reverse } (\text{cons } x \ x s) = \text{snoc } (\text{reverse } x s) \ x
\]

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

\[
\text{reverse-list} : \text{List A} \to \text{List A} \\
\text{reverse-list} \; \text{x} = \text{go} \; \text{x} \; \text{nil}
\]

\begin{verbatim}
  where
  go : \text{List A} \to (\text{List A} \to \text{List A})
  go \; \text{nil} = \text{id}
  go (\text{cons} \; \text{x} \; \text{x}s) = \text{go} \; \text{x}s \; . \; \text{cons} \; \text{x}
\end{verbatim}

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 : \( Vec A \ n \rightarrow Vec A \ m \rightarrow Vec A \ (n + m) \)

reverse \( \text{nil} \) acc = acc

reverse \( \text{cons} \ x \ \text{xs} \) acc = {!reverse \( \text{xs} \) (cons x acc)!}

Error

Goal: \( Vec A \ ((\text{succ} \ k) + m) \)

Have: \( Vec A \ (k + (\text{succ} m)) \)
Reversing vectors

\[ \text{reverse} : \text{Vec} \ A \ n \ \rightarrow \ \text{Vec} \ A \ m \ \rightarrow \ \text{Vec} \ A \ (n + m) \]

\[ \text{reverse} \ \text{nil} \quad \text{acc} \ = \ \text{acc} \]

\[ \text{reverse} \ (\text{cons} \ x \ \text{xs}) \ \text{acc} \ = \ \{ \text{!reverse} \ \text{xs} \ (\text{cons} \ x \ \text{acc})! \} \]

Error

Goal: \[ \text{Vec} \ A \ ((\text{succ} \ k) + m) \]

Have: \[ \text{Vec} \ A \ (k + (\text{succ} \ m)) \]

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

\[
\text{addAcc} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
\]

\[
\text{addAcc} \; \text{zero} \; m = m
\]

\[
\text{addAcc} \; (\text{succ} \; k) \; m = \text{addAcc} \; k \; (\text{succ} \; m)
\]

\[
\text{reverseAcc} : \text{Vec} \; A \; n \rightarrow \text{Vec} \; A \; m \rightarrow \text{Vec} \; A \; (\text{addAcc} \; n \; m)
\]

\[
\text{reverseAcc} \; \text{nil} \; \text{acc} = \text{acc}
\]

\[
\text{reverseAcc} \; (\text{cons} \; x \; \text{xs}) \; \text{acc} = \text{reverseAcc} \; \text{xs} \; (\text{cons} \; x \; \text{acc})
\]
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)
reverse : \text{Vec} A \ n \rightarrow \text{Vec} A \ n \\
reverse \ xs = \{!\text{reverseAcc} \ xs \ \text{nil}!\} \\

Error \\
Goal: \text{Vec} A \ n \\
Have: \text{Vec} A \ (\text{addAcc} \ n \ \text{zero}) \\

Remember the definition of addAcc: \\

addAcc : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
addAcc \ \text{zero} \quad m = m \\
addAcc \ (\text{succ} \ k) \quad m = \text{addAcc} \ k \ (\text{succ} \ m)
Showing Agda who's the boss

reverse : \( \text{Vec } A \ n \rightarrow \text{Vec } A \ n \)
reverse \( xs \) = coerceVec proof (reverseAcc xs \( \text{nil} \))

where
proof : addAcc \( n \) zero \( \equiv \) \( n \)
coerceVec : \( n \equiv m \rightarrow \text{Vec } A \ n \rightarrow \text{Vec } A \ m \)
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, naïve 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

\[ \text{DNat} : \text{Set} \]
\[ \text{DNat} = \text{Nat} \rightarrow \text{Nat} \]

\[ [\_\_] : \text{N} \rightarrow \text{DNat} \]
\[ [\ n\ ] = \lambda \ m \rightarrow m + n \]

\[ \text{reify} : \text{DNat} \rightarrow \text{N} \]
\[ \text{reify} \ dn = \ dn \ \text{zero} \]
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  : \( \forall \ dn \rightarrow \text{reify} \ dn \equiv \text{reify} \ (dn + \text{dzero}) \)
unit-left   : \( \forall \ dn \rightarrow \text{reify} \ dn \equiv \text{reify} \ (\text{dzero} + \ dn) \)
+-assoc     : \( \forall \ dn \ dm \ dk \rightarrow \text{reify} \ (dn + (dm + dk)) \equiv \text{reify} \ ((dn + dm) + dk) \)
Difference naturals are monoidal

\[\text{dzero} : \text{DNat}\]
\[\text{dzero} = \text{id}\]

\[\_+\_ : \text{DNat} \rightarrow \text{DNat} \rightarrow \text{DNat}\]
\[\text{dn + dm = dm . dn}\]

And three properties:

\[\text{unit-right} : \forall \text{dn} \rightarrow \text{reify dn} \equiv \text{reify (dn + dzero)}\]
\[\text{unit-left} : \forall \text{dn} \rightarrow \text{reify dn} \equiv \text{reify (dzero + dn)}\]
\[\text{+-assoc} : \forall \text{dn dm dk} \rightarrow \text{reify (dn + (dm + dk))} \equiv \text{reify ((dn + dm) + dk)}\]

Each of these properties holds \textit{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)
Can we define vector reverse using difference naturals?

We can almost complete the desired definition...

\[
\text{revAcc} : (\text{dm} : \text{DNat}) \rightarrow \text{Vec} A n \rightarrow \text{Vec} A (\text{reify dm}) \rightarrow \text{Vec} A (\text{dm n})
\]

\[
\text{revAcc dm \ nil \ acc} = \text{acc}
\]

\[
\text{revAcc dm (cons x xs) acc} = \text{revAcc (dsucc dm) xs \{!cons x acc!\}}
\]

\[\begin{align*}
\text{Goal:} & \quad \text{Vec} A (\text{dm (succ zero)}) \\
\text{Have:} & \quad \text{Vec} A (\text{succ (dm zero)})
\end{align*}\]

We are trying to extend the accumulator using cons – but we don’t know how \text{dm} and cons interact.
The type of \texttt{cons}

Adding new elements to a vector:

\texttt{cons} : \forall \ n \rightarrow A \rightarrow Vec A \ n \rightarrow Vec A \ (\text{succ} \ n)

But we would like to accumulate elements as follows:

\texttt{dcons} : \forall \ n \ dm \rightarrow A \rightarrow Vec A \ (dm \ n) \rightarrow Vec A \ (dm \ (\text{succ} \ n))
The type of \texttt{cons}

Adding new elements to a vector:

\[
\texttt{cons} : \forall \ n \rightarrow A \rightarrow \texttt{Vec} \ A \ n \rightarrow \texttt{Vec} \ A \ (\texttt{succ} \ n)
\]

But we would like to accumulate elements as follows:

\[
\texttt{dcons} : \forall \ n \ \texttt{dm} \rightarrow A \rightarrow \texttt{Vec} \ A \ (\texttt{dm} \ n) \rightarrow \texttt{Vec} \ A \ (\texttt{dm} \ (\texttt{succ} \ n))
\]

- But when we kick off the computation, \texttt{dm} is the identity function - \texttt{cons} would suffice.
Adding new elements to a vector:

\[
\text{cons} : \forall \ n \to A \to \text{Vec} \ A \ n \to \text{Vec} \ A \ (\text{succ} \ n)
\]

But we would like to accumulate elements as follows:

\[
\text{dcons} : \forall \ n \ dm \to A \to \text{Vec} \ A \ (dm \ n) \to \text{Vec} \ A \ (dm \ (\text{succ} \ n))
\]

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

\[
\text{revAcc} : \\
\forall \ dm \rightarrow (\forall \ k \rightarrow A \rightarrow \text{Vec} \ A (dm \ k) \rightarrow \text{Vec} \ A ((\text{dsucc} \ dm) \ k)) \rightarrow \\
\text{Vec} \ A \ n \rightarrow \text{Vec} \ A \ (\text{reify} \ dm) \rightarrow \text{Vec} \ A \ (dm \ n)
\]

\[
\text{revAcc} \ dm \ dcons \ \text{nil} \ acc = acc \\
\text{revAcc} \ dm \ dcons \ (\text{cons} \ x \ xs) \ acc = \text{revAcc} \ (\text{dsucc} \ m) \ dcons \ xs \ (dcons \ x \ acc)
\]

reverse : Vec A n \rightarrow Vec A n

\[
\text{reverse} \ xs = \text{revAcc} \ dzero \ \text{cons} \ xs \ \text{nil}
\]
Vector reverse

\[ \forall \ dm \to (\forall \ k \to A \to Vec A (dm \ k) \to Vec A ((dsucc \ dm) \ k)) \to \\
Vec A n \to Vec A (reify \ dm) \to Vec A (dm \ n) \]

\[
\text{revAcc } dm \ dcons \ \text{nil} \quad \text{acc} = \text{acc} \\
\text{revAcc } dm \ dcons (\text{cons} \ x \ xs) \ \text{acc} = \text{revAcc} (\text{dsucc} \ m) \ dcons \ xs \ (dcons \ x \ acc) \\
\]

reverse : \( \text{Vec} \ A \ n \to \text{Vec} \ A \ n \)

\[
\text{reverse } xs = \text{revAcc} \ dzero \ \text{cons} \ xs \ \text{nil} \\
\]
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)
Generalise `foldl` to work over a \( \mathbb{N} \) indexed \( B \):

\[
\text{foldl-vec} : (B : \mathbb{N} \rightarrow \text{Set}) \rightarrow (B \ k \rightarrow A \rightarrow B \ (\text{succ} \ k)) \rightarrow B \ \text{zero} \rightarrow \text{Vec} \ A \ n \rightarrow B \ n
\]

\[
\begin{align*}
\text{foldl-vec} \ B \ \text{step} \ \text{acc} \ \text{nil} & = \text{acc} \\
\text{foldl-vec} \ B \ \text{step} \ \text{acc} \ (\text{cons} \ x \ \text{xs}) & = \text{foldl-vec} \ (B \circ \text{succ}) \ \text{step} \ \text{(step acc x)} \ \text{xs}
\end{align*}
\]

The second case is not so obvious...

It counts down over (by induction on \( \text{xs} \)) and up (by \textit{precomposing} with \textit{Succ}) at the same time!
Generalise \texttt{foldl} to work over a \(\mathbb{N}\) indexed \(B\):

\[
\text{foldl-vec} : (B : \mathbb{N} \to \text{Set}) \to (B \, k \to A \to B \, (\text{succ} \, k)) \to B \, \text{zero} \to \text{Vec} \, A \, n \to B \, n
\]

\[
\text{foldl-vec} \, B \, \text{step} \, \text{acc} \, \text{nil} = \text{acc}
\]

\[
\text{foldl-vec} \, B \, \text{step} \, \text{acc} \, (\text{cons} \, x \, \text{xs}) = \text{foldl-vec} \, (B \circ \text{succ}) \, \text{step} \, (\text{step} \, \text{acc} \, x) \, \text{xs}
\]

The second case is not so obvious...

It counts down over (by induction on \(\text{xs}\)) and up (by \textit{precomposing} with \texttt{succ}) at the same time!

\[
\text{reverse} : \text{Vec} \, A \, n \to \text{Vec} \, A \, n
\]

\[
\text{reverse} = \text{foldl-vec} \, (\text{Vec} \, A) \, (\text{flip} \, \text{cons}) \, \text{nil}
\]
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.
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
Suppose we fix \( A : \text{Set} \) as (the carrier of) a monoid.

The monoidal expressions over \( A \) are given by:

\[
\text{data} \ \text{Expr} : \text{Set where}
\begin{align*}
\_ \oplus \_ & : \text{Expr} \ A \to \text{Expr} \ A \to \text{Expr} \ A \\
\text{zero} & : \text{Expr} \ A \\
\text{var} & : A \to \text{Expr} \ A
\end{align*}
\]

We can evaluate these expressions readily enough:

\[
\text{eval} : \text{Expr} \ A \to A
\]
We can define the mappings to/from their Cayley representation:

\[
\begin{align*}
\llbracket \_ \rrbracket & : \text{Expr } A \rightarrow (\text{Expr } A \rightarrow \text{Expr } A) \\
\text{reify} & : (\text{Expr } A \rightarrow \text{Expr } A) \rightarrow \text{Expr } A
\end{align*}
\]
We can define the mappings to/from their Cayley representation:

\[
\begin{align*}
\text{⟦}_\_\text{⟧} & : \text{Expr } A \rightarrow (\text{Expr } A \rightarrow \text{Expr } A) \\
\text{reify} & : (\text{Expr } A \rightarrow \text{Expr } A) \rightarrow \text{Expr } A
\end{align*}
\]

And we can use these to normalise any expression:

\[
\begin{align*}
\text{normalise} & : \text{Expr } A \rightarrow \text{Expr } A \\
\text{normalise } e & = \text{reify } \text{⟦ } e \text{⟧}
\end{align*}
\]
Proof sketch - part \textit{succ (succ \textit{zero})}

We need to prove one lemma:

soundness : (e : \textit{Expr} \ a) \rightarrow \text{eval (normalise e)} = \text{eval e}
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 \((\text{succ} (\text{succ} (\text{succ } \text{zero})))\)

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

\[
\text{example} : (xs \; ys \; zs : \text{List } A) \to \\
((xs \; \text{++ } []) \; \text{++ } (ys \; \text{++ } zs)) \equiv ((xs \; \text{++ } ys) \; \text{++ } zs )
\]

\[
\text{example } xs \; ys \; zs = \\
\text{let } e_1 = (\text{var } xs \; \text{⊕ } \text{zero}) \; \text{⊕ } (\text{var } ys \; \text{⊕ } \text{var } zs) \text{ in} \\
\text{let } e_2 = (\text{var } xs \; \text{⊕ } \text{var } ys) \; \text{⊕ } \text{var } zs \text{ in} \\
\text{solve } e_1 \; e_2 \; \text{refl}
\]

The quoting can be automated using Agda’s reflection mechanism.
This construction works for *any* monoid...

In particular, for the natural numbers using accumulating addition.
This construction works for *any* monoid...

In particular, for the natural numbers using accumulating addition.

```haskell
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
```
• The Cayley representation of a monoid satisfies the monoid laws by definition.
• 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).
• 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!