From monoids to near-semirings: the essence of MonadPlus and Alternative

Exequiel Rivas Mauro Jaskelioff
CIFASIS-CONICET
Universidad Nacional de Rosario, Argentina
jaskelioff@cifasis-conicet.gov.ar
rivas@cifasis-conicet.gov.ar

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract

It is well-known that monads are monoids in the category of endofunctors, and in fact so are applicative functors. Unfortunately, the benefits of this unified view are lost when the additional nondeterminism structure of MonadPlus or Alternative is required.

This article recovers the essence of these two type classes by extending monoids to near-semirings with both additive and multiplicative structure. This unified algebraic view enables us to generically define the free construction as well as a novel double Cayley representation that optimises both left-nested sums and left-nested products.

Keywords monoid, near-semiring, monad, monadplus, applicative functor, alternative, free construction, Cayley representation

1. Introduction

The monad interface provides a basic structure for computations:

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

where the operations return, for injecting values, and $\gg=$, for sequentially composing computations, are subject to the three monad laws. Study of this structure has led to many insights and applications. Two of these are especially notable:

1. The free instance of the monad interface, the free monad, has many applications in defining new monads and extending the capabilities of existing ones, e.g., in the form of algebraic effect handlers [16].

```
data Free f a = Return x | Op (f (Free f a))
instance Functor f ⇒ Monad (Free f) where
  return x  = Return x
  Return x >>= f = f x
  Op op    >>= f = Op (fmap (>>=f) op)
```

2. The codensity transformation CodT m provides a continuation-based representation for a monad m.

```
newtype CodT m a = CodT (∀x.(a → m x) → m x)
rep :: Monad m ⇒ m a → CodT m a
rep v = CodT (v>>=)
abs :: Monad m ⇒ CodT m a → m a
abs (CodT c) = c return
```

Since CodT m is a representation for m, we can lift m-computations to CodT m, compute in that monad, and when we are finished go back to m using abs. This change of representation is useful because it turns left-nested binds into right-nested binds [10, 22]. This is very convenient for monads (like the free monad) where left-nested binds are costly and right-nested binds are cheap.

```
instance Monad (CodT m) where
  return x     = CodT (λk → k x)
  CodT c >>= f = CodT (λk → c (λa →
                   let CodT g = f a in g k))
```

Both results can be derived by viewing a monad as a monoid in a monoidal category. The free monad is just the free monoid in that category and the codensity transformation arises as the *Cayley representation* of that monoid [17]. Moreover, useful generality is gained by this approach, as not only monads are monoids, but also applicative functors and arrows.

While the monad interface is well understood and comes with many useful results, it is also very limiting. When dealing with specific computations, we always require additional operations. A prominent example is non-determinism that occurs, e.g., in logic programming languages and parser combinators. Non-deterministic computations involve two additional operations: a failing computation (mzero) and a non-deterministic choice between two computations (mplus). These additional operations are captured in Haskell by the MonadPlus type class:

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

This type class comes with five additional laws that govern the interaction between the operations.

It is not difficult to see that the above two constructions for the Monad interface do not work for the MonadPlus interface. Firstly, the free monad has no provision for the two additional operations of MonadPlus and the five additional laws.

[Copyright notice will appear here once 'preprint' option is removed.]

Secondly, the codensity construction does not optimise left-nested uses of mplus. Consider for instance, the following program due to Fischer [5]:

```
anyof :: MonadPlus m ⇒ [a] → m a
anyof []       = mzero
anyof (x : xs) = anyof xs ‘mplus‘ return x
```

If we instantiate m with the list monad, whose MonadPlus instance is defined as follows

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

we obtain the naive-reverse program, which has a quadratic running time.

Let us now consider what happens if we use CodT [] instead. While there is no established CodT instance for MonadPlus, we can easily provide one[1] in terms of the underlying operations:

```
instance MonadPlus m ⇒ MonadPlus (CodT m) where
  mzero                    = CodT (λk → mzero)
  CodT p ‘mplus‘ CodT q = CodT (λk → p k ‘mplus‘ q k)
```

However, there is no improvement by running the computation on CodT []. The problem is that CodT [] just delegates the MonadPlus operations to the underlying instance. This obviously does not improve the running time.

This paper provides a new algebraic understanding of the operations of the MonadPlus type class, one that enables us to derive both the free structure and an optimised Cayley-like representation. As we have argued, the monoid view is insufficient for this purpose; we require a richer algebraic structure that augments monoids with additional operations. This algebraic structure is that of a *near-semiring*.

Specifically, our contributions are as follows:

- We present a generalised form of near-semirings (Section 3), and provide generic definitions for its free construction and a novel *double* Cayley representation.
- We establish that both MonadPlus (Section 4) and Alternative (Section 5) are instances of this generalised notion, and we specialise the constructions for both cases.
- We demonstrate the use of the constructions on two examples: combinatorial search and interleaving parsers (Section 6).

There is quite a bit of related work; this is discussed in Section 7.

2. Monoids and Near-Semirings

In this section we introduce ordinary monoids and near-semirings. That is, we present monoids and near-semirings over sets.

2.1 Background: Monoids

A *monoid* $(M, \otimes, \mathsf{e})$ is a triple consisting of a set M, together with an operation $\otimes : M \times M \to M$ and an element $\mathsf{e} \in M$ such that the following axioms hold for all a, b, and $c \in M$:

$$a \otimes \mathsf{e} = a \quad (1)$$

$$\mathsf{e} \otimes a = a \quad (2)$$

$$a \otimes (b \otimes c) = (a \otimes b) \otimes c \quad (3)$$

The operation $\otimes$ is called the multiplication of the monoid, while the element e is called the unit. We usually refer to a monoid $(M, \otimes_M, \mathsf{e}_M)$ simply by its carrier set M.

[1] The instance is derived using the MonadPlus laws and abs/rep.

Using type classes, we can describe monoids in Haskell as follows:

```
class Monoid m where
  mempty  :: m
  mappend :: m → m → m
```

Here, mempty is the unit element and mappend is the multiplication. Instances of this class are required to satisfy the monoid laws. However, these are not enforced by Haskell, and it is left to the programmer to verify them.

There are two important constructions that are important for this paper: free monoids and the Cayley representation for monoids.

Free Monoids The notion of free monoid is defined in terms of monoid homomorphisms. A *monoid homomorphism* is a function from one monoid to another that preserves the monoid structure.

Definition 2.1. *A* monoid homomorphism *from a monoid* $(M, \otimes_M, \mathsf{e}_M)$ *to a monoid* $(N, \otimes_N, \mathsf{e}_N)$ *is a function* $f : M \to N$ *such that* $f(\mathsf{e}_M) = \mathsf{e}_N$ *and* $f(m \otimes_M m') = f(m) \otimes_N f(m')$.

Now we can define the notion of free monoid.

Definition 2.2. *The* free monoid *over a set* X *is a monoid* $(X^*, \otimes_*, \mathsf{e}_*)$ *together with a function* $\mathsf{inj} : X \to X^*$ *such that for every monoid* $(M, \otimes_M, \mathsf{e}_M)$ *and function* $h : X \to M$*, there exists a unique monoid homomorphism* $\overline{h} : X^* \to M$ *such that* $\overline{h} \circ \mathsf{inj} = h$.

A concrete representation for the free monoid over a set X are lists with elements of that set; concatenation is its multiplication and the empty list is its unit.

The free monoid construction extends to a functor. That is, for every function $f : X \to Y$, we define the monoid homomorphism $f^* : X^* \to Y^*$ as $f^* = \overline{\mathsf{inj} \circ f}$.

For every monoid $(M, \otimes_M, \mathsf{e}_M)$, the monoid morphism

$$\overline{\mathsf{id}_M} : M^* \to M$$

behaves as an *evaluation algebra* in the following sense: M^* represents the syntax of programs constructed from the monoid operations and elements of M. The algebra $\overline{\mathsf{id}_M}$ simply gives semantics to these programs by replacing the syntactic operations in M^* with the corresponding monoid operations in M.

Monoid Representation A representation for a given monoid $(M, \otimes_M, \mathsf{e}_M)$ is a monoid $(R(M), \otimes_{R(M)}, \mathsf{e}_{R(M)})$, together with functions $\mathsf{rep} : M \to R(M)$ and $\mathsf{abs} : R(M) \to M$ such that the following diagram commutes.

$$\begin{array}{ccc} M^* & \xrightarrow{\mathsf{rep}^*} & R(M)^* \\ \downarrow{\scriptstyle \overline{\mathsf{id}_M}} & & \downarrow{\scriptstyle \overline{\mathsf{id}_{R(M)}}} \\ M & \xleftarrow{\mathsf{abs}} & R(M) \end{array}$$

Intuitively, the diagram states that running a monoid program on M is the same as first interpreting it as a monoid program on the representation $R(M)$, running it there, and then abstracting the result back into M.

Cayley Representation The Cayley representation of a monoid is an efficient representation of that monoid with a constant time multiplication.

The *monoid of endomorphisms* over a set X is $(X \to X, \circ, \mathsf{id})$, where $\circ$ is function composition and id is the identity function. Every monoid has an embedding into the monoid of endomorphisms over its carrier set, a result usually known as Cayley's theorem for monoids [17].

Theorem 2.3 (Cayley for (Set) monoids)**.** *Every monoid* $(M, \otimes, \mathsf{e})$ *embeds into the monoid of endomorphisms over the set* M*, namely* $(M \to M, \circ, \mathsf{id})$*. The embedding is given by the monoid morphism* $\mathsf{rep} : M \to (M \to M)$ *and function* $\mathsf{abs} : (M \to M) \to M$

$$\begin{array}{rcl} \mathsf{rep}(a) & = & \lambda b.\, a \otimes b \\ \mathsf{abs}(f) & = & f(\mathsf{e}) \end{array}$$

with the property that $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$.

A simple consequence of this theorem is the following

Corollary 2.4. *The monoid of endomorphisms* $(M \to M, \circ, \mathsf{id})$ *is a representation of the monoid* $(M, \otimes, \mathsf{e})$*, with* rep *and* abs *as in the theorem above.*

Proof. Because rep is a monoid homomorphism the following diagram commutes.

$$\begin{array}{ccc} M^* & \xrightarrow{\mathsf{rep}^*} & (M \to M)^* \\ \downarrow \overline{\mathsf{id}_M} & & \downarrow \overline{\mathsf{id}_{M \to M}} \\ M & \xrightarrow[\mathsf{rep}]{} & M \to M \end{array}$$

Since $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$, we conclude that $M \to M$ is a monoid representation for M. □

2.2 Near-Semirings

When two monoids align in a particular way, they form a near-semiring, the central structure in this paper. Formally, a near-semiring is defined as a quintuple[2] $(M, \otimes, \mathsf{e}, \oplus, \mathsf{z})$ where both $(M, \otimes, \mathsf{e})$ and $(M, \oplus, \mathsf{z})$ are monoids for the same set M; moreover, the following laws relate both structures:

$$\mathsf{z} \otimes a = \mathsf{z} \tag{4}$$

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c) \tag{5}$$

Here $\otimes$ is the multiplication of the near-semiring, $\oplus$ is the addition, e is the unit, and z is the zero.

The following Haskell type class models near-semirings, in the same way as the Monoid type class models monoids.

```
class Nearsemiring a where
  (⊗)  :: a → a → a
  one  :: a
  (⊕)  :: a → a → a
  zero :: a
```

Every instance of Nearsemiring is expected to satisfy the near-semiring axioms.

In Section 2.1, we presented two constructions for monoids: the free monoid and the Cayley representation of a monoid. It is natural to ask whether these two constructions carry over to near-semirings. The answer is yes!

Free Construction We define near-semiring homomorphisms in the same way as monoid homomorphisms.

Definition 2.5. *A* near-semiring homomorphism *from a given near-semiring* $(M, \otimes_M, \mathsf{e}_M, \oplus_M, \mathsf{z}_M)$ *to some near-semiring* $(N, \otimes_N, \mathsf{e}_N, \oplus_N, \mathsf{z}_N)$ *is a function* $f : M \to N$ *such that:*

$$\begin{array}{r} f(m \otimes_M n) = f(m) \otimes_N f(n) \\ f(\mathsf{e}_M) = \mathsf{e}_N \\ f(m \oplus_M n) = f(m) \oplus_N f(n) \\ f(\mathsf{z}_M) = \mathsf{z}_N \end{array}$$

[2] In the literature, sometimes the unit is not required: $(M, \otimes)$ is expected only to be a semigroup. In this article we consider only near-semirings with unit, and call them simply near-semirings.

Now we can define the free near-semiring over a set A.

Definition 2.6. *The free near-semiring over a set* A *is a near-semiring* A^* *together with a map* $\mathsf{inj} : A \to A^*$ *satisfying that for every near-semiring* $(N, \oplus, \mathsf{e}, \otimes, \mathsf{z})$ *and function* $h : A \to N$*, there exists a unique near-semiring homomorphism* $\overline{h} : A^* \to N$ *such that* $\overline{h} \circ \mathsf{inj} = h$*.*

Diagrammatically, we have the following commuting diagram:

$$\begin{array}{ccc} A & \xrightarrow{\mathsf{inj}} & A^* \\ & \searrow^{h} & \downarrow \overline{h} \\ & & N \end{array}$$

Just like lists are a concrete representation for free monoids, forests are a concrete representation for free near-semirings.

```
data Forest a = Forest [Tree a]
data Tree a = Leaf | Node a (Forest a)

instance Nearsemiring (Forest a) where
  zero                       = Forest []
  one                        = Forest [Leaf]
  (Forest xs) ⊕ (Forest ys) = Forest (xs ++ ys)
  (Forest xs) ⊗ (Forest ys) = Forest (concatMap g xs)
    where g Leaf       = ys
          g (Node a n) =
            [Node a (n ⊗ (Forest ys))]
```

The addition $\oplus$ combines the trees of two forests and has the empty forest as neutral element. The multiplication $\otimes$ substitutes all the leaves in one forest by the other forest; its neutral element is a forest that consists of a single leaf.

The inclusion of generators inj and the universal morphism univ are defined as follows:

```
inj :: a → Forest a
inj a = Forest [Node a one]

univ :: Nearsemiring n ⇒ (a → n) → Forest a → n
univ h (Forest xs) = foldr (⊕) zero (map univ_T xs)
  where univ_T Leaf        = one
        univ_T (Node a ts) = h a ⊗ univ h ts
```

Just like the free monoid, the free near-semiring also extends to a functor: given a function $f : X \to Y$ we define the near-semiring homomorphism $f^* : X^* \to Y^*$ as $f^* = \overline{\mathsf{inj} \circ f}$. Also analogous to the monoid case, we have an evaluation algebra $\overline{\mathsf{id}_N} : N^* \to N$ for every near-semiring N.

Near-semiring Representation We define a representation of a near-semiring N as a near-semiring $R(N)$, together with functions $\mathsf{rep} : N \to R(N)$ and $\mathsf{abs} : R(N) \to N$ such that the following diagram commutes.

$$\begin{array}{ccc} N^* & \xrightarrow{\mathsf{rep}^*} & R(N)^* \\ \downarrow \overline{\mathsf{id}_N} & & \downarrow \overline{\mathsf{id}_{R(N)}} \\ N & \xleftarrow[\mathsf{abs}]{} & R(N) \end{array}$$

Intuitively, the diagram states that running a near-semiring program on N is the same as first interpreting it as a near-semiring program on the representation $R(N)$, running it there, and then abstracting the result back into N.

Double Cayley Representation The *double Cayley* representation is obtained by applying the Cayley representation for monoids twice, first for the additive monoid structure and then for the multiplicative monoid structure. However, if we do this naively, we do not get a good representation: we want to represent a near-semiring, and therefore the whole near-semiring structure must be taken into account and not just one chosen monoid structure.

If we take a near-semiring $(N, \otimes, \mathsf{e}, \oplus, \mathsf{z})$ and apply Cayley for monoids on the monoid $(N, \oplus, \mathsf{z})$, we obtain the monoid $(N \to N, \circ, \mathsf{id})$ with representation and abstraction functions

$$\mathsf{rep}_{\oplus}(x) = \lambda y.\, x \oplus y$$
$$\mathsf{abs}_{\oplus}(f) = f\ \mathsf{z}$$

where $\mathsf{abs}_{\oplus} \circ \mathsf{rep}_{\oplus} = \mathsf{id}$ holds.

However, it is not clear how to extend this monoid to a near-semiring. For instance, using a point-wise product does not yield a near-semiring.

A solution to this problem is to restrict the representation to the image of $\mathsf{rep}_{\oplus}$, where the representation is *exact*. That is, N is isomorphic to the image of $\mathsf{rep}_{\oplus}$, as it is not difficult to see that if $x = \mathsf{rep}_{\oplus}(a)$ for some $a \in N$, then

$$\mathsf{rep}_{\oplus}(\mathsf{abs}_{\oplus}(x)) = \mathsf{rep}_{\oplus}(\mathsf{abs}_{\oplus}(\mathsf{rep}_{\oplus}(a))) = \mathsf{rep}_{\oplus}(a) = x$$

We define the set $N \overset{\bullet}{\to} N$ as the functions in the image of $\mathsf{rep}_{\oplus}$. That is, the set $N \overset{\bullet}{\to} N$ is the set of functions h such that

$$h\ y = \mathsf{abs}_{\oplus}(h) \oplus y = h\ \mathsf{z} \oplus y.$$

It is easy to extend the monoid $(N \overset{\bullet}{\to} N, \circ, \mathsf{id})$ to a near-semiring because we can use the isomorphism between $N \overset{\bullet}{\to} N$ and N in order to reuse the multiplicative structure of N. We obtain the near-semiring $(N \overset{\bullet}{\to} N, \otimes', \mathsf{e}', \circ, \mathsf{id})$ where

$$\mathsf{e}' = \mathsf{rep}_{\oplus}(\mathsf{e}) = \lambda y.\, \mathsf{e} \oplus y$$
$$f \otimes' g = \mathsf{rep}_{\oplus}(\mathsf{abs}_{\oplus}(f) \otimes \mathsf{abs}_{\oplus}(g)) = \lambda y.\, f\ \mathsf{z} \otimes g\ \mathsf{z} \oplus y$$

Now $\mathsf{rep}_{\oplus}$ is a near-semiring homomorphism, and therefore we have a representation that accounts for the additive structure while preserving the multiplicative structure.

Next, we apply Cayley again, but this time over the multiplicative structure: we take the near-semiring $(N \overset{\bullet}{\to} N, \otimes', \mathsf{e}', \circ, \mathsf{id})$, apply Cayley for monoids on the monoid $(N \overset{\bullet}{\to} N, \otimes', \mathsf{e}')$, and obtain the monoid $((N \overset{\bullet}{\to} N) \to (N \overset{\bullet}{\to} N), \circ, \mathsf{id})$ with representation and abstraction functions

$$\mathsf{rep}_{\otimes}(f) = \lambda g.\, f \otimes' g = \lambda g\ y.\, f\ \mathsf{z} \otimes g\ \mathsf{z} \oplus y$$
$$\mathsf{abs}_{\otimes}(f) = f\ \mathsf{e}' = f\ (\lambda y.\, \mathsf{e} \oplus y)$$

where $\mathsf{abs}_{\otimes} \circ \mathsf{rep}_{\otimes} = \mathsf{id}$ holds. The extension of the obtained monoid to a near-semiring is simple in this case as the sum and zero can be defined point-wise. We arrive at the near-semiring $((N \overset{\bullet}{\to} N) \to (N \overset{\bullet}{\to} N), \circ, \mathsf{id}, \oplus', \mathsf{z}')$ where

$$\mathsf{z}' = \lambda x.\, \mathsf{id}$$
$$f \oplus' g = \lambda x.\, f\ x \circ g\ x$$

and the representation and abstraction functions are:

$$\mathsf{rep}(x) = \mathsf{rep}_{\otimes}(\mathsf{rep}_{\oplus}(x)) = \lambda h\ y.\, x \otimes h(\mathsf{z}) \oplus y$$
$$\mathsf{abs}(f) = \mathsf{abs}_{\otimes}(\mathsf{abs}_{\oplus}(f)) = f\ (\lambda x.\, \mathsf{e} \oplus x)(\mathsf{z})$$

Unfortunately, the type $(N \overset{\bullet}{\to} N) \to (N \overset{\bullet}{\to} N)$ is not usually available in programming languages due to the side conditions on the type constructor $\overset{\bullet}{\to}$. One has to compromise and use the type $DC(N) = (N \to N) \to (N \to N)$, for which we obtain the semiring of endomorphisms over endomorphisms $(DC(N), \circ, \mathsf{id}, \oplus', \mathsf{z}')$ as in the following Haskell implementation:

```
type DC n = (n → n) → (n → n)
instance Monoid n ⇒ Nearsemiring (DC n) where
  f ⊗ g = f ∘ g
  one   = id
  f ⊕ g = λh → f h ∘ g h
  zero  = const id
```

Note that rep is not a near-semiring homomorphism from N into $DC(N)$ as it does not preserve the unit:

$$\mathsf{rep}(\mathsf{e}) = \lambda h\ y.\, h(\mathsf{z}) \oplus y \quad \neq \quad \lambda h\ y.\, h(y) = \mathsf{e}.$$

Nevertheless, the semiring of endomorphisms over endomorphisms is a valid representation for N as shown by the following theorem.

Theorem 2.7. *Let $(N, \otimes, \mathsf{e}, \oplus, \mathsf{z})$ be a near-semiring. Then the near-semiring $(DC(N), \circ, \mathsf{id}, \oplus', \mathsf{z}')$ is a representation of N. That is, the following diagram commutes.*

$$\begin{array}{ccc} N^* & \xrightarrow{\mathsf{rep}^*} & DC(N)^* \\ \downarrow{\scriptstyle \overline{\mathsf{id}_N}} & & \downarrow{\scriptstyle \overline{\mathsf{id}_{DC(N)}}} \\ N & \xleftarrow{\mathsf{abs}} & DC(N) \end{array}$$

Proof. By definition of free near-semiring, $\overline{\mathsf{id}}$ is the *unique* near-semiring homomorphism such that $\overline{\mathsf{id}} \circ \mathsf{inj} = \mathsf{id}$. Doing some calculations, it can be shown that $\mathsf{abs} \circ \overline{\mathsf{id}} \circ \mathsf{rep}^*$ is a near-semiring homomorphism and it also satisfies the property:

$$\begin{aligned} \mathsf{abs} \circ \overline{\mathsf{id}} \circ \mathsf{rep}^* \circ \mathsf{inj} &= \mathsf{abs} \circ \overline{\mathsf{id}} \circ \overline{\mathsf{inj} \circ \mathsf{rep}} \circ \mathsf{inj} \\ &= \mathsf{abs} \circ \overline{\mathsf{id}} \circ \mathsf{inj} \circ \mathsf{rep} \\ &= \mathsf{abs} \circ \mathsf{id} \circ \mathsf{rep} \\ &= \mathsf{abs} \circ \mathsf{rep} \\ &= \mathsf{id} \end{aligned}$$

Therefore, $\overline{\mathsf{id}} = \mathsf{abs} \circ \overline{\mathsf{id}} \circ \mathsf{rep}^*$. □

Hence, the semantics of a computation over a near-semiring will be preserved if we lift values to the representation, do the near-semiring computation there, and then go back to the original near-semiring.

3. Generalisation

In this section we generalise the notion of monoid and near-semiring over sets to monoidal categories. The next sections show that these generalised notions enable us to identify monads and applicative functors with a near-semiring structure.

3.1 Background: Monoidal Categories

Monoidal categories generalise the notion of monoids from sets A to categories $\mathcal{C}$. In order to prepare for the generalisation, we first express both operations $\otimes$ and e of as unary morphisms[3]:

$$m : A \times A \to A$$
$$e : \{*\} \to A$$

Observe that we have introduced an argument from the singleton set $\{*\}$ to the unit.

Now we can generalise to a category $\mathcal{C}$ by making two replacements:

[3] m is short for *multiply*, as a for *add* and z for *zero*.

1. The Cartesian product, which is a bifunctor on sets $- \times - : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set}$, becomes a bifunctor $- \otimes - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$.
2. The singleton set $\{*\}$ becomes an object $I \in \mathcal{C}$.

This yields the morphisms:

$$m : M \otimes M \to M$$
$$e : I \to M$$

We expect $\otimes$ and I to work like $\times$ and $\{*\}$, in the sense that $\times$ is associative and $\{*\}$ behaves as its unit (up to isomorphism). For that, we require the following natural isomorphisms

$$\alpha : A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$$
$$\lambda : I \otimes A \cong A$$
$$\rho : A \otimes I \cong A$$

which are expected to interact coherently. A category $\mathcal{C}$ with such structure is known as a *monoidal category*, and a monoid in it is an object M, together with operations m and e as above, for which the following laws hold:

$$\lambda = m \circ (e \otimes \mathsf{id})$$
$$\rho = m \circ (\mathsf{id} \otimes e)$$
$$m \circ (m \otimes \mathsf{id}) \circ \alpha = m \circ (\mathsf{id} \otimes m)$$

Rivas and Jaskelioff [17] show that the free and Cayley constructions carry over from monoids to generalised monoids. Here we investigate whether the same is true for generalised near-semirings.

3.2 Generalised Near-Semirings

Just like a near-semiring combines two monoids, a generalised near-semiring combines two generalised monoids. This requires a category $\mathcal{C}$ that is monoidal in two ways. Hence, it is equipped with four families of morphisms:

$$\begin{array}{rlrl} m & : M \otimes M \to M & a & : M \oplus M \to M \\ e & : I_\otimes \to M & z & : I_\oplus \to M \end{array}$$

with $M, I_\otimes, I_\oplus$ objects of $\mathcal{C}$, and $- \otimes - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ and $- \oplus - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ bifunctors. In addition, there are the six natural isomorphisms of the two monoidal categories: $\alpha_\otimes, \lambda_\otimes, \rho_\otimes$ and $\alpha_\oplus, \lambda_\oplus, \rho_\oplus$.

A near-semiring category requires two additional natural isomorphisms that connect the multiplicative and additive structure:

$$\kappa : I_\oplus \otimes M \to I_\oplus$$
$$\delta : (M_1 \oplus M_2) \otimes M_3 \to (M_1 \otimes M_3) \oplus (M_2 \otimes M_3)$$

We define a generalised near-semiring in a near-semiring category $\mathcal{C}$ as an object M in $\mathcal{C}$ which is a monoid in both monoidal structures, and for which the following interaction laws hold:

$$a \circ (m \oplus m) \circ \delta = m \circ (a \otimes \mathsf{id})$$
$$z \circ \kappa = m \circ (z \otimes \mathsf{id})$$

In particular, we recover ordinary near-semirings by setting $\mathcal{C} = \mathsf{Set}$, $\otimes = \oplus = \times$ and $I_\otimes = I_\oplus = \{*\}$.

Cartesian Structure Because the additive structure of all the instances in the remainder of this paper is Cartesian, we only consider this case from now on.[4] This means that $\oplus$ is the Cartesian bifunctor $\times$ and $I_\oplus$ is the terminal object 1. As usual with the Cartesian structure, the three natural isomorphisms $\alpha_\oplus$, $\lambda_\oplus$, and $\rho_\oplus$ are named assoc, π_1, and π_2. Moreover, the distribution law δ can now be defined as $\delta = \langle \pi_1 \otimes \mathsf{id}, \pi_2 \otimes \mathsf{id} \rangle$ and the annihilation law κ can be defined as $\kappa = \,!$ where $! : M \to 1$ is the family of unique homomorphisms associated with the terminal object 1.

[4] If desired, the following definitions are easily generalised again.

For each object X there are morphisms

$$\mathsf{nil} : 1 \to \mathsf{List}(X)$$
$$\mathsf{cons} : X \times \mathsf{List}(X) \to \mathsf{List}(X)$$
$$\mathsf{concat} : \mathsf{List}(X) \times \mathsf{List}(X) \to \mathsf{List}(X)$$
$$\mathsf{wrap} : X \to \mathsf{List}(X)$$

representing, the empty list, consing an element to a list, list concatenation, and the singleton list, respectively.

Figure 1. List operations

Having fixed the additive structure to be the Cartesian structure, there is no ambiguity and hence we omit the subscripts from the multiplicative structure, e.g. we write I instead of $I_\otimes$.

3.3 Generalised Free Near-semiring

To construct the free general near-semiring, we need the category to have coproducts $- + - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, and also that both $- \times -$ and $- \otimes -$ are *closed*. A monoidal structure $- \odot -$ is closed if there exists a bifunctor $- \overset{\odot}{\Rightarrow} -$ together with an isomorphism:

$$\lfloor \cdot \rfloor_\odot : \mathcal{C}(A \odot B, C) \cong \mathcal{C}(A, B \overset{\odot}{\Rightarrow} C) : \lceil \cdot \rceil_\odot$$

natural in A, B, and C. Here $\lfloor \cdot \rfloor_\odot$ and $\lceil \cdot \rceil_\odot$ are the bijections between the hom-sets. The evaluation morphism $\mathsf{ev}_\odot$ is defined as:

$$\mathsf{ev}_\odot = \lceil \mathsf{id}_{B \overset{\odot}{\Rightarrow} C} \rceil : (B \overset{\odot}{\Rightarrow} C) \odot B \to C$$

In some sense, this is a generalisation of the bijection between types $(a, b) \to c$ and $a \to b \to c$, witnessed by curry and $\mathsf{uncurry}$. When an operator is closed, then a distributive law with respect to the coproduct is available:

$$\delta_\odot : (A + B) \odot C \to A \odot C + B \odot C$$
$$\delta_\odot = \lceil [\lfloor \mathsf{inl} \rfloor, \lfloor \mathsf{inr} \rfloor] \rceil$$

where $[_, _]$ is case analysis on a coproduct.

Given an object A, if the initial algebra for the endofunctor $1 + (I + A \otimes -) \times -$ exists, then the free near-semiring over A has the following carrier:

$$\mu X.\, 1 + (I + A \otimes X) \times X \quad \cong \quad \mu X.\, \mathsf{List}\,(I + A \otimes X)$$

where $\mathsf{List} : \mathcal{C} \to \mathcal{C}$ is the list endofunctor (see Figure 1).

The near-semiring operations for the free near-semiring are similar to those defined for the ordinary free near-semiring and are given in Figure 2. These definitions are highly abstract; the following sections provide more accessible concrete instances of this general construction.

3.4 Generalised Double Cayley Representation

A generalised version of the double Cayley representation shown in Section 2.2 is constructed as follows. If A is an object, then

$$(A \overset{\times}{\Rightarrow} A) \overset{\otimes}{\Rightarrow} (A \overset{\times}{\Rightarrow} A)$$

has a generalised near-semiring structure. Notice that both $- \overset{\times}{\Rightarrow} -$ and $- \overset{\otimes}{\Rightarrow} -$ are used. This was hidden in the case of sets, as the multiplicative structure was also the Cartesian product.

The definition of the generalised double Cayley representation is given in Figure 3.

4. Non-Determinism Monads

We now instantiate the generalisation presented in the previous section to monads in order to obtain $\mathsf{MonadPlus}$.

$$\mathsf{List}(I + A \otimes A^*) \underset{\mathsf{out}}{\overset{\mathsf{in}}{\cong}} A^*$$

$$
\begin{aligned}
&a : A^* \times A^* \to A^* \\
&a = \mathsf{in} \circ \mathsf{concat} \circ (\mathsf{out} \times \mathsf{out}) \\
&z : 1 \to A^* \\
&z = \mathsf{in} \circ \mathsf{nil} \\
&m : A^* \otimes A^* \to A^* \\
&m = \lceil (\!(\![\lfloor \mathsf{nil} \circ \kappa \rfloor, \lfloor r \rfloor]\!)\!) \rceil \\
&\quad \text{where } r = \mathsf{concat} \circ ([\lambda, \mathsf{wrap} \circ \mathsf{inr}] \circ \delta_\otimes) \times \mathsf{id} \circ \delta \\
&e : I_\otimes \to A^* \\
&e = \mathsf{in} \circ \mathsf{wrap} \circ \mathsf{inl} \\
&\overline{f} : A^* \to N \\
&\overline{f} = (\!(\![z, [a \circ (e \times \mathsf{id}), a \circ ((m \circ (f \otimes \mathsf{id})) \times \mathsf{id})] \circ \delta_\times]\!)\!) \\
&\quad \text{where } f : A \to N \text{ and } N \text{ is a near-semiring}
\end{aligned}
$$

Figure 2. Operations of the free near-semiring

Near-semiring operations:

$$
\begin{aligned}
z &= \lfloor \lfloor \pi_2 \rfloor_\times \circ ! \rfloor_\otimes \\
a &= \lfloor \lfloor \mathsf{ev}_\times \circ (\mathsf{id} \times \mathsf{ev}_\times) \circ \alpha^{-1} \rfloor_\times \circ (\mathsf{ev}_\otimes \times \mathsf{ev}_\otimes) \circ \delta \rfloor_\otimes \\
e &= \lfloor \lambda \rfloor_\otimes \\
m &= \lfloor \mathsf{ev}_\otimes \circ (\mathsf{id} \otimes \mathsf{ev}_\otimes) \circ \alpha^{-1} \rfloor_\otimes
\end{aligned}
$$

Representation and abstraction functions:

$$
\begin{aligned}
\mathsf{rep} &= \lfloor\, \lfloor a \rfloor_\times \circ m \circ (\mathsf{id} \otimes (\mathsf{ev}_\times \circ \langle \mathsf{id}, z \circ ! \rangle))\, \rfloor_\otimes \\
\mathsf{abs} &= \mathsf{ev}_\times \circ \langle \mathsf{id}, z \circ ! \rangle \circ \mathsf{ev}_\otimes \circ (\mathsf{id} \otimes \lfloor a \circ (e \times \mathsf{id}) \rfloor_\times) \circ \rho^{-1}
\end{aligned}
$$

Figure 3. Double Cayley Representation

Monads are Monoids in the Category of Endofunctors Our starting point is the category End of endofunctors on a category $\mathcal{C}$, which consists of endofunctors as objects and natural transformations as morphisms.

The monoids in this category are monads if we choose the tensor $\otimes$ to be $\circ$, the composition of endofunctors. Formally, composition of functors F and G is $(F \circ G)(X) = F(G(X))$. The unit of the monoid is the identity functor $\mathsf{Id}(X) = X$. This structure is strict, which means that the three of $\lambda_\circ$, $\rho_\circ$ and $\alpha_\circ$ are identities.

The two associated natural transformations are:

$$
\begin{aligned}
m &: M \circ M \to M \\
e &: \mathsf{Id} \to M
\end{aligned}
$$

The following Haskell type class captures these natural transformations in idiomatic Haskell code:

```
class Functor m ⇒ Triple m where
  return :: a → m a
  join   :: m (m a) → m a
```

where m corresponds to join and e to return.

The three generalised monoid laws are the usual monad laws found in category theory textbooks. This presentation of monads contrasts with their standard presentation in programming languages, where an operation ($\gg\!=$) is used instead of join. While both presentations are in fact equivalent, we prefer the presentation in terms of join here, as its easier to see how it fits in the monoidal framework.

Non-Determinism Monads are Near-Semirings in the Category of Endofunctors In order to obtain a near-semiring, we need to complement the above monoidal structure with a Cartesian one. If we assume that the underlying category $\mathcal{C}$ of End has a Cartesian structure, then such a structure exists for End as well: If F and G are two endofunctors on $\mathcal{C}$, then their binary product is defined point-wisely as

$$(F \times G)(X) = F(X) \times G(X)$$

The terminal endofunctor 1 is simply the constant endofunctor K_1 which maps every object to the terminal object 1 of $\mathcal{C}$, and every morphism to id_1.

From the Cartesian structure we get two additional natural transformations:

$$
\begin{aligned}
a &: M \times M \to M \\
z &: K_1 \to M
\end{aligned}
$$

In Haskell, we write their types as $a :: \forall a. m\ a \to m\ a \to m\ a$ and $z :: \forall a. m\ \ a$, and aptly call them mplus and mzero after MonadPlus's methods. This additional information is presented by extending the Triple type class.

```
class Triple m ⇒ TriplePlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

With this type class, the next three laws of near-semirings, which talk about mplus and mzero forming a monoid for the Cartesian structure, read as:

$$m \text{ `mplus` mzero} = m \tag{6}$$

$$\text{mzero `mplus` } m = m \tag{7}$$

$$m_1 \text{ `mplus` } (m_2 \text{ `mplus` } m_3) = (m_1 \text{ `mplus` } m_2) \text{ `mplus` } m_3 \tag{8}$$

The final two laws relate the two monoidal structures as follows.

$$\text{join mzero} = \text{mzero} \tag{9}$$

$$\text{join } (m_1 \text{ `mplus` } m_2) = \text{join } m_1 \text{ `mplus` join } m_2 \tag{10}$$

In order to compare this structure with MonadPlus, we translate these axioms to the more common Monad presentation. Equations 6-8 remain the same in the MonadPlus type class, as they do not involve ($\gg\!=$). The last two laws, 9 and 10, translate to:

$$\text{mzero} \gg\!= k = \text{mzero} \tag{11}$$

$$(m_1 \text{ `mplus` } m_2) \gg\!= k = (m_1 \gg\!= k) \text{ `mplus` } (m_2 \gg\!= k) \tag{12}$$

These two axioms are commonly known as *left zero* and *left distribution*. A MonadPlus instance satisfying equations 6, 7, 8, 11, and 12 is an instance of a generalised near-semiring, and we call it a *non-determinism monad*.

Examples A familiar example of a non-determinism monad is the list monad, with its MonadPlus instance defined in the introduction. Another example is the power-set monad, with union of sets as mplus, and the empty set as mzero. These two monads are usually used to represent non-determinism, where mplus collects the possible answers.

Another popular example is Hutton & Meijer's monadic parsers using lists [11]. In this case, the non-determinism accounts for the parser collecting the multiple ways of parsing a given string.

```
newtype Parser[] a =
  Parser[] {unParser[] :: String → [(a, String)]}
```

```
instance Functor f ⇒ Monad (Free∘ f) where
  return x = Free∘ [Pure∘ x]
  Free∘ xs >>= f = Free∘ (concatMap g xs)
    where g (Pure∘ x) = unFree∘ (f x)
          g (Con∘ x)  = [Con∘ (fmap (>>=f) x)]
instance Functor f ⇒ Functor (Free∘ f) where
  fmap f x = x >>= return ∘ f
instance Functor f ⇒ MonadPlus (Free∘ f) where
  mzero = Free∘ []
  Free∘ xs `mplus` Free∘ ys = Free∘ (xs ++ ys)
```

Figure 4. Instances for the free non-determinism monad

```
instance Monad Parser[] where
  return x = Parser[] (λs → return (x, s))
  x >>= f  = Parser[] (λs → unParser[] x s >>=
                        λ(y, s') → unParser[] (f y) s')
instance MonadPlus Parser[] where
  mzero        = Parser[] (λs → mzero)
  x `mplus` y = Parser[] (λs →
                   unParser[] x s `mplus` unParser[] y s)
```

But not every MonadPlus instance satisfies the five near-semiring axioms. An interesting class of such instances is that of those MonadPlus instances that satisfy the *left catch* law rather than left distribution:[5]

$$\text{return } a \text{ `mplus` } b = \text{return } a \qquad (13)$$

An example of this kind is Maybe. A counter-example to left distribution is obtained by instantiating m_1 = Just Nothing and m_2 = Just (Just False) in eq. 10.

A difference between left distribution and left catch laws is that the first relates mplus with join, while the latter relates mplus with return. Left catch is related to an algebraic structure called *dioids* [6], but in this article we only study near-semiring structures.

The MonadPlus instance for Parser[] is based on the instance of MonadPlus for lists. With the same definitions, similar parser combinators can be constructed using Maybe instead of lists, but these parsers do not form a non-determinism monad.

4.1 The Free Non-Determinism Monad

While the generalisation of the free near-semirings asks $\times$ and $\circ$ to be closed, the closures are not explicitly used in the free construction. Hence we postpone their introduction until we construct the double Cayley representation.

The following datatype constructs the free near-semiring. We introduce it as two mutually recursive datatypes, one representing the free construction over the Cartesian product, and the other over the composition of functors.

```
data Free∘ f x = Free∘ {unFree∘ :: [FFree∘ f x]}
data FFree∘ f x = Pure∘ x | Con∘ (f (Free∘ f x))
```

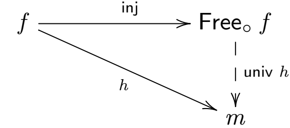
The implementation of the operations, listed in Figure 4, follows directly from the types; it is analogous to the free construction for ordinary near-semirings.

[5] The MonadPlus Reform Proposal suggests that such instances belong in a separate MonadOr typeclass. See https://wiki.haskell.org/MonadPlus_reform_proposal

```
instance Functor (g ⇒∘ h) where
  fmap f m = Ran (λk → unRan m (k ∘ f))
⌊·⌋∘ :: (Functor f, Functor g, Functor h) ⇒
        (∀x.f (g x) → h x) → (∀x.f x → (g ⇒∘ h) x)
⌊f⌋∘ = λgx → Ran (λk → f (fmap k gx))
⌈·⌉∘ :: (Functor f, Functor g, Functor h) ⇒
        (∀x.f x → (g ⇒∘ h) x) → (∀x.f (g x) → h x)
⌈f⌉∘ = λfgx → unRan (f fgx) id
instance Functor (f ⇒× g) where
  fmap f m = Exp (λk → unExp m (k ∘ f))
⌊·⌋× :: (Functor f, Functor g, Functor h) ⇒
        (∀x.(f x, g x) → h x) → (∀x.f x → (g ⇒× h) x)
⌊f⌋× = λfx → Exp (λt → λgy → f (fmap t fx, gy))
⌈·⌉× :: (Functor f, Functor g, Functor h) ⇒
        (∀x.f x → (g ⇒× h) x) → (∀x.(f x, g x) → h x)
⌈f⌉× = λfgx → unExp (f (fst fgx)) id (snd fgx)
```

Figure 5. Right Kan extension and exponential

The universal property for the free near-semirings is given by the following functions that make the universal diagram commute:



inj embeds values of the original functor in the free structure and univ f uniquely maps the free structure onto another near-semiring.

```
inj :: Functor f ⇒ f a → Free∘ f a
inj x = Free∘ [Con∘ (fmap return x)]
univ :: (MonadPlus m, Functor f) ⇒
        (∀x.f x → m x) → Free∘ f x → m x
univ h (Free∘ l) = foldr mplus mzero (map univ⊤ l)
  where univ⊤ (Pure∘ x)  = return x
        univ⊤ (Con∘ op) = h op >>= univ h
```

4.2 The Double Cayley Representation

We now define the closures of the two bifunctors and construct the double Cayley representation from them.

Closures The first Cayley representation uses the codensity monad, which is the exponential with respect to the composition of functors:

```
newtype (⇒∘) f g x = Ran {unRan :: ∀y.(x → f y) → g y}
```

The second Cayley representation uses the following exponential of the Cartesian product:

```
newtype (⇒×) f g x =
          Exp {unExp :: ∀y.(x → y) → (f y → g y)}
```

The corresponding isomorphisms for both exponentials are given in Figure 5.

Double Cayley Representation Given a non-determinism monad m, we can embed it in the data-type $(m \overset{\times}{\Rightarrow} m) \overset{\circ}{\Rightarrow} (m \overset{\times}{\Rightarrow} m)$, obtaining the following construction:

```
newtype DC f x = DC {unDC :: ((f ⇒× f) ⇒∘ (f ⇒× f)) x}
```

```
instance Monad (DC f) where
  return x = DC (Ran (λf → f x))
  DC (Ran m) ≫= f = DC (Ran (λg → m (λa →
    unRan (unDC (f a)) g)))
instance MonadPlus (DC f) where
  mzero = DC (Ran (λk → Exp (λc x → x)))
  mplus m n = DC (Ran (λsk →
    Exp (λf fk → unExp (a sk) f (unExp (b sk) f fk))))
    where DC (Ran a) = m
          DC (Ran b) = n
```

Figure 6. Operations of the double Cayley representation

Figure 6 defines the associated near-semiring operations.

The values of any non-determinism monad m can be embedded in the double Cayley construction DC m with the function rep. After performing computation in DC m, the m value can be recovered using abs.

```
rep :: Monad m ⇒ m a → DC m a
rep x = DC (Ran (λg → Exp
          (λh m → (x ≫= λa → unExp (g a) h m))))
abs :: MonadPlus m ⇒ DC m a → m a
abs m = unExp (f (λx → Exp
                (λh m → return (h x) ‘mplus‘ m)))
              id mzero
        where DC (Ran f) = m
```

Example Now we can solve the anyof performance problem presented in the introduction. First, we rewrite anyof for a general MonadPlus instance:

```
anyof :: MonadPlus m ⇒ [a] → m a
anyof []       = mzero
anyof (x : xs) = anyof xs ‘mplus‘ return x
```

Then, instead of running it directly over lists, we run it first on DC [], and then we flatten back the result to lists:

```
anyof′ :: [a] → [a]
anyof′ xs = abs (anyof xs)
```

Here abs forces the type m in anyof such that the double Cayley representation over lists is used. The complexity of the new implementation is linear in the length of the input list, while the direct implementation was quadratic.

5. Alternative Applicatives

Monads are not the only monoids in the category of endofunctors. Another important class are *applicative functors*, introduced by McBride and Patterson [14] as a way to capture certain effectful computations that do not fit well in the monadic framework.

Multiplicative Structure Applicative functors are based on a different multiplicative structure than monads: the *Day convolution*. As first indicated by Day [4] and later by Rivas and Jaskelioff [17], there are different equivalent representations of the Day convolution. We pick here the one that directly determines the familiar Applicative type class.

```
data (⋆) f g a = ∀b.Day (f (b → a)) (g b)
instance (Functor f, Functor g) ⇒ Functor (f ⋆ g) where
  fmap h (Day ff gx) = Day (fmap (λf → h ◦ f) ff) gx
```

Just like for $- \circ -$, the unit for Day convolution is the identity functor. The isomorphisms $\lambda_\star$, $\rho_\star$ and $\alpha_\star$ are defined as follows.

```
λ :: Functor f ⇒ f a → (Identity ⋆ f) a
λ fa = Day (Identity id) fa
ρ :: Functor f ⇒ f a → (f ⋆ Identity) a
ρ fa = Day (fmap const fa) (Identity ())
α :: (Functor f, Functor g, Functor h) ⇒
     ((f ⋆ g) ⋆ h) a → (f ⋆ (g ⋆ h)) a
α (Day (Day ff gx) hy) =
       Day (fmap uncurry ff) (Day (fmap (,) gx) hy)
```

The monoidal operations directly lead to the well-known operations of the Applicative type class:

```
class Functor m ⇒ Applicative m where
  pure :: a → m a
  (⊛) :: m (a → b) → m a → m b
```

Additive Structure The additive structure is the same as for monads: the Cartesian product of endofunctors. Hence, we obtain a near-semiring structure by adding to applicative functors two operations, one of type $\forall a.f\ a \to f\ a \to f\ a$ and other of type $\forall a.f\ a$. The established Haskell type class for applicative functors with these two operations is Alternative:

```
class Applicative f ⇒ Alternative f where
  empty :: f a
  (⟨|⟩)  :: f a → f a → f a
```

As far as we know, the only three established laws that every Alternative instance should obey are those of a monoid $f\ a$ with operation $\langle|\rangle$ as multiplication, and empty as the identity element.

Instantiating the near-semiring laws proposed, we obtain two additional laws:

$$\mathsf{empty} \circledast x = \mathsf{empty}$$
$$(f \langle|\rangle g) \circledast x = (f \circledast x) \langle|\rangle (g \circledast x)$$

5.1 Examples

Maybe It is well-known that every monad is also an applicative functor. This transformation taking monads to applicative functors can be extended such that every MonadPlus is an Alternative.

The converse is not true: there are Alternative instances that do not arise from a non-determinism monad. In the previous section we saw that Maybe is not a MonadPlus as the left-distribution law fails. However, when restricted to the applicative interface, Maybe does satisfy the Alternative laws:

```
instance Applicative Maybe where
  pure x = Just x
  Just f ⊛ Just x  = Just (f x)
  _      ⊛ _       = Nothing
instance Alternative Maybe where
  empty = Nothing
  Nothing ⟨|⟩ y = y
  (Just v) ⟨|⟩ _ = Just v
```

The Applicative instance captures a *conjunction*-semantics: two computations are successfully combined iff both computations are successful. In contrast, the instance of Alternative reflects a left-biased *disjunction*-semantics (rather than non-determinism).

Non-Example: Maybe Parsers Even though Maybe satisfies the Alternative laws, the following Maybe-based parser datatype does not.

```
newtype Parser_Maybe a = P (String → Maybe (a, String))
```

```
instance Functor Parser_Maybe where
  fmap f (P p) = P (λs → case p s of
                         Nothing     → Nothing
                         Just (v, s') → Just (f v, s'))
instance Applicative Parser_Maybe where
  pure x = P (λs → Just (x, s))
  (P f) ⊛ (P v) = P (λs → case f s of
    Nothing     → Nothing
    Just (g, s') → case v s' of
        Nothing      → Nothing
        Just (w, s'') → Just (g w, s''))
instance Alternative Parser_Maybe where
  empty = P (λs → Nothing)
  (P p) ⟨|⟩ (P q) = P (λs → case p s of
                            Nothing    → q s
                            Just (v, s) → Just (v, s))

char :: Char → Parser_Maybe ()
char d = P (λs → case s of
  ""      → Nothing
  (c : s') → if c ≡ d then Just ((), s')
                      else  Nothing)
```

Figure 7. Invalid Applicative instance: Maybe-based parsers

Figure 7 provides the instances for Functor, Applicative and Alternative. However, this last one is invalid. Consider for instance these two parsers:

```
p1 = (char 'a' ⟨|⟩ pure ()) ∗ char 'a'
p2 = (char 'a' ∗ char 'a') ⟨|⟩ (pure () ∗ char 'a')
```

where ∗ is a variant of ⊛ that ignores the computed values.

```
(∗) :: Applicative p ⇒ p a → p b → p ()
p1 ∗ p2 = pure (λx y → ()) ⊛ p1 ⊛ p2
```

According to the distributive law, p_1 and p_2 should be equivalent. However, they are in fact distinct and hence $\mathsf{Parser}_{\mathsf{Maybe}}$ is not a valid Alternative instance.

```
> runParser p1 "a"
Nothing
> runParser p2 "a"
Just ((), "")
```

In the first parser, the left branch succeeds consuming the entire input. Because the left branch succeeds, the right branch is discarded. Because the entire input is consumed, the subsequent char 'a' fails and, as there are no more alternatives to try, the overall parser fails. In the second parser, the left branch does not succeed and there is still a second branch that does.

Composition of Alternative and Applicative Composition of applicative functors is an applicative functor. Therefore, if f and g are applicative their composition $f \circ g$ is going to be applicative.

```
newtype (f ∘ g) x = Comp (f (g x))
instance (Applicative f, Applicative g) ⇒
         Applicative (f ∘ g) where
  pure x = Comp (pure (pure x))
  Comp fs ⊛ Comp xs = Comp (pure (⊛) ⊛ fs ⊛ xs)
```

If additionally f is an Alternative, then the composition is also an Alternative.

```
instance Functor f ⇒ Functor (FFree⋆ f) where
  fmap f (Pure⋆ a)   = Pure⋆ (f a)
  fmap f (Con⋆ g x) = Con⋆ (fmap (f∘) g) x
instance Functor f ⇒ Functor (Free⋆ f) where
  fmap f (Free⋆ xs) = Free⋆ (fmap (fmap f) xs)
instance Functor f ⇒ Applicative (Free⋆ f) where
  pure x = Free⋆ [Pure⋆ x]
  Free⋆ xs ⊛ v = Free⋆ (concatMap g xs)
    where g (Pure⋆ f)   = unFree⋆ (fmap f v)
          g (Con⋆ f c) =
            [Con⋆ (fmap uncurry f) (pure (,) ⊛ c ⊛ v)]
instance Functor f ⇒ Alternative (Free⋆ f) where
  empty = Free⋆ []
  Free⋆ xs ⟨|⟩ Free⋆ ys = Free⋆ (xs ++ ys)
inj :: Functor f ⇒ f a → Free⋆ f a
inj x = Free⋆ [Con⋆ (fmap (λz () → z) x) (pure ())]
univ :: Alternative g ⇒
        (∀x.f x → g x) → Free⋆ f x → g x
univ h (Free⋆ [])                = empty
univ h (Free⋆ ((Pure⋆ x) : xs)) =
                     pure x ⟨|⟩ univ h (Free⋆ xs)
univ h (Free⋆ ((Con⋆ q c) : xs)) =
                     (h q ⊛ univ h c) ⟨|⟩ univ h (Free⋆ xs)
```

Figure 8. Supporting code for the free Alternative

```
instance (Alternative f, Applicative g) ⇒
         Alternative (f ∘ g) where
  empty = Comp empty
  Comp xs ⟨|⟩ Comp ys = Comp (xs ⟨|⟩ ys)
```

For example, one can extend an arbitrary applicative functor f to an Alternative, by composing it with the list functor.

Ziplists Additionally to the usual instance, lists have another instance of Applicative. This is a typical example of an applicative functor that is not derived from a Monad instance.

```
newtype ZipList a = ZL [a]

instance Applicative ZipList where
  pure x          = ZL (repeat x)
  ZL fs ⊛ ZL xs = ZL (zipWith ($) fs xs)
```

Perhaps surprisingly, ZipLists have an Alternative instance. Like the Alternative instance for Maybe, the one for ZipList has a left bias.

```
instance Alternative ZipList where
  empty            = ZL []
  ZL xs ⟨|⟩ ZL ys = ZL (xs ++ drop (length xs) ys)
```

5.2 The Free Alternative

Based on the generic recipe we obtain the following definition for the free Alternative.

```
data Free⋆ f a = Free⋆ {unFree⋆ :: [FFree⋆ f a]}
data FFree⋆ f a = Pure⋆ a
                | ∀b.Con⋆ (f (b → a)) (Free⋆ f b)
```

The supporting definitions of the operations, the injection and the family of unique homomorphisms can all be found in Figure 8.

```
instance Functor g ⇒ Functor (f ⇒⋆ g) where
  fmap f (ED g) =
    ED (λy → fmap (λ(a, b) → (f a, b)) (g y))
⌊·⌋⋆ :: (Functor f, Functor g, Functor h) ⇒
    (∀x.(f ⋆ g) x → h x) → (∀x.f x → (g ⇒⋆ h) x)
⌊f⌋⋆ = λfx → ED (λgy → f (Day (fmap (, ) fx) gy))
⌈·⌉⋆ :: (Functor f, Functor g, Functor h) ⇒
    (∀x.f x → (g ⇒⋆ h) x) → (∀x.(f ⋆ g) x → h x)
⌈f⌉⋆ = λ(Day ff gy) → fmap (uncurry ($))
                              (unED (f ff) gy)
instance Functor f ⇒ Functor (DC f) where
  fmap f (DC z) = DC (ED (λfx →
    fmap (λ(x, y) → (f x, y)) (unED z fx)))
instance Functor f ⇒ Applicative (DC f) where
  pure v = DC (ED (λf → fmap (λy → (v, y)) f))
  (DC (ED h)) ⊛ (DC (ED v)) = fmap (uncurry ($))
    (DC (ED (λf →
      fmap (λ(b, (a, y)) → ((a, b), y)) (v (h f)))))
instance Functor f ⇒ Alternative (DC f) where
  empty = DC (ED (const (Exp (λh x → x))))
  DC f ⟨|⟩ DC g = DC (ED (λh →
    Exp (λj i →
      unExp (unED f h) j (unExp (unED g h) j i))))
```

Figure 9. Supporting code for the double Cayley Alternative

5.3 The Double Cayley Representation

We now instantiate the general double Cayley representation. We have already covered the closed structure ($\overset{\times}{\Rightarrow}$) for the Cartesian functor in the previous section. The following datatype is the closure of the Day convolution:

```
data (⇒⋆) f g x = ED {unED :: ∀y.f y → g (x, y)}
```

With this definition the double Cayley representation instantiates to the following functor:

```
newtype DC f x = DC {unDC :: ((f ⇒× f) ⇒⋆ (f ⇒× f)) x}
```

See Figure 9 for the supporting code.

The functions rep and abs convert between the double Cayley representation and the original applicative functor.

```
rep :: Alternative f ⇒ f a → DC f a
rep x = DC (ED (λg → Exp (λf fx →
                   unExp g (flip (curry f)) empty ⊛ x
                   ⟨|⟩ fx)))
abs :: Alternative f ⇒ DC f a → f a
abs (DC (ED f)) = unExp (f (Exp (λg i →
                   pure (g ()) ⟨|⟩ i))) fst empty
```

6. Applications

We illustrate the constructions presented in the previous sections on a number of examples. The free structure is a highly convenient way to extend arbitrary semi-rings with additional capabilities. We show this on two examples: combinatorial search and parsers. The double Cayley representation is complementary: it makes building the free structure efficient.

6.1 Advanced Combinatorial Search

Bunches Spivey proposed an algebraic structure, called a *bunch*, for expressing combinatorial search strategies such as depth-first (DFS) and breadth-first (BFS) search [19]. A bunch is in fact a non-determinism monad with one additional operation:

```
class MonadPlus m ⇒ Bunch m where
  wrap :: m a → m a
```

In addition to the axioms of a non-determinism monad, a bunch also satisfies the following axiom relating (≫=) and wrap:

$$\mathsf{wrap}\ m \gg\!= k = \mathsf{wrap}\ (m \gg\!= k)$$

This requirement is automatically fulfilled if we instead require an operation $\mathsf{wrap}' :: a \to m\ a$, and define $\mathsf{wrap}\ x = x \gg\!= \mathsf{wrap}'$.

Spivey introduces the initial Bunch algebra in the form of the Forest datatype:

```
type Forest a = [Tree a]
data Tree a = Leaf a | Fork (Forest a)
```

This is nothing more than the free non-determinism monad on the identity functor, i.e. Forest a is isomorphic to $\mathsf{Free}_\circ$ Identity a where the identity functor is

```
newtype Identity a = Id {runId :: a}
```

The wrap operator for this type is expressed as follows.

```
wrap :: Forest a → Forest a
wrap xf = [Fork xf]
```

In terms of $\mathsf{Free}_\circ$ Identity this operator is expressed as:

```
wrap :: Free∘ Identity a → Free∘ Identity a
wrap xf = Free∘ [Id xf]
```

The unique morphism from Forest a to any other Bunch, such as DFS or BFS, is given as follows.

```
search :: Bunch m ⇒ Forest a → m a
search ts = msum (map go ts)
  where go (Leaf x)  = return x
        go (Fork ts) = wrap (search ts)
```

Equivalently, using the representation in terms of the free non-determinism monad we can define search as:

```
search :: Bunch m ⇒ Free∘ Identity a → m a
search = univ (wrap ∘ return ∘ runId)
```

Heuristic Search The free structure is a great way to generically extend existing search strategies with pruning *search heuristics*. Such heuristics are commonly used in the case of large search spaces whose entire exploration is either infeasible or impractical. They remove parts of the search space that are less likely to yield (interesting) solutions and where otherwise the search would dwell too long.

One of the best known heuristics is depth-bounded search which bounds the search tree to a certain depth, pruning everything underneath.

```
dbs :: Functor f ⇒ Int → Forest a → Forest a
dbs 0 _  = mzero
dbs n ts = map go ts
  where go (Leaf a) = Leaf a
        go (Fork f) = Fork (dbs (n − 1) f)
```

By means of search ∘ dbs n we can combine this heuristic with any search strategy.

Double Cayley Speed-Up In order to evaluate the impact of the double Cayley representation, we consider the following extreme benchmark.

```
bench :: Int → Int
bench n    = solutionCount (forest n)

forest :: Int → Forest ()
forest 0 = return ()
forest n = (forest (n − 1) >>= wrap′) ‘mplus‘ wrap′ ()

solutionCount :: Forest () → Int
solutionCount f = sum (map go f)
  where go (Leaf _) = 1
        go (Fork f) = solutionCount f
```

Note that forest generates the ideal situation for the double Cayley representation: alternated left-nested occurrences of $\gg\!=$ and mplus.

We ran the benchmark for different problem sizes both using Forest directly and indirectly through the double Cayley representation. All runs took place in the Criterion benchmarking harness using GHC 7.8.2 on a MacBook Pro with a 2 GHz Intel Core i7 processor, 8 GB RAM and Mac OS 10.10.1. All values are in milliseconds.

Size	Forest	DC Forest
50	2	0
100	26	1
250	569	8
500	5,472	36
1,000	60,070	172
2,500	T/O	1,484

The double Cayley representation clearly provides a tremendous improvement over the basic free construction in terms of absolute runtimes. Moreover, the former seems to exhibit a cubic time complexity, while the latter seems to have the expected quadratic complexity that corresponds to the size of the generated forest.

6.2 Interleaving Alternative Parsers

Swierstra and Dijkstra [21] show how to extend Alternative parser combinators with a new combinator ($\langle||\rangle$) to *interleave* two given parsers. For example, suppose we have a parser digits for digit sequences and another parser letters for letter sequences. Then the interleaved parser (digits $\langle||\rangle$ letters) accepts strings like "a1b45cda19" and produces the result ("abcda", 14519).

Instead of defining the new combinator directly for the given Alternative parser type P, Swierstra and Dijkstra define it for a new parser type Gram P. This type Gram P is almost, but not quite the free Alternative construction. In particular, it does not respect the left distributive law set out in this paper. Moreover, while Swierstra and Dijkstra manage to avoid duplicating results, their approach does drop results. For instance,

```
(pure 'a' ⟨|⟩ pure 'b') ⟨||⟩ pure 1
```

only yields the result ('a', 1) and drops the result ('b', 1).

Our free Alternative construction provides a cleaner slate for interleaved parsers. As a consequence, our solution respects all the Alternative laws and neither duplicates nor drops results. Moreover, we obtain a solution that is not parser-specific, but works for any Alternative instance.

The interleaving operator $\langle||\rangle$ is defined as follows:

```
(⟨||⟩) :: Functor f ⇒ Free⋆ f a → Free⋆ f b → Free⋆ f (a, b)
ga ⟨||⟩ gb =
                  Free⋆ [fa ‘fwdby‘ gb | fa ← fas]
    ⟨|⟩   (swap ⟨$⟩ Free⋆ [fb ‘fwdby‘ ga | fb ← fbs])
    ⟨|⟩           Free⋆ [Pure⋆ (a, b)  | a ← as, b ← bs]
  where
    swap (a, b) = (b, a)
    (as, fas)    = split ga
    (bs, fbs)    = split gb
```

At the sequential FFree$_\star$-level, it considers three different scenarios:

1. Computation ga goes first, i.e. performs its first primitive action, and then the remainder is interleaved recursively.
2. Dually, computation gb goes first.
3. Finally, in the base case both computations have no more action to perform and terminate with a result.

These three scenarios are lifted to the non-deterministic Free$_\star$-level in the obvious way, and the auxiliary function split separates the base cases from the recursive cases.

```
split :: Free⋆ f a → ([a], [FFree⋆ f a])
split (Free⋆ l) = ([a | Pure⋆ a ← l]
                  , [f | f@(Con⋆ p r) ← l])
```

The key to the first two scenarios is to decompose a computation into its first primitive action p and the remainder r. This decomposition comes for free in the Con$_\star$ p r constructor of the free Alternative.

```
fwdby :: Functor f ⇒
         FFree⋆ f a → Free⋆ f b → FFree⋆ f (a, b)
(Con⋆ pa ra) ‘fwdby‘ fbs = Con⋆ (fmap first pa) (ra ⟨||⟩ fbs)
                     where first f (c, b) = (f c, b)
```

Finally, using the injection inj :: Parser$_{[]}$ $a \to$ Free$_\star$ Parser$_{[]}$ a, we can embed Parser$_{[]}$ in the free construction, and afterwards recover it with the help of univ id :: Free$_\star$ Parser$_{[]}$ $a \to$ Parser$_{[]}$ a.

7. Related Work

7.1 Codensity Monad and Cayley Representation

Cayley representations appear under different guises in the literature. Hughes uses it to optimise list concatenation [9] and Voigtländer [22] uses the codensity monad transformer to optimise monadic computations. Rivas and Jaskelioff [17] show that these two optimisations are instances of the Cayley representation for monoids in a generalised setting, and extend it to applicative functors. Our work extends this representation to include the additional operators present in non-deterministic computations by moving from generalised monoids to generalised near-semirings.

7.2 Representation of Near-semirings

Statman [20] provides a connection between lambda calculus and the algebra of near-semirings. He introduces a generalisation of Hoogewijs' representation [8] on which we base our double Cayley representation. Krishna and Chatterjee [13] study the representation of near-semirings in categories, but they only consider Cartesian structures and thus exclude monads and applicative functors.

7.3 Backtracking Monad Transformers

Hinze [7] introduces the backtracking monad transformer, a monad transformer that augments any monad with backtracking capabilities (mplus and mzero):

As opposed to our approach Hinze provides a transformer, i.e. a construction that adds a capability (backtracking). Our construction is a change of representation that can be applied to a monad which

already has this capability. In our approach one can use a simple transformer to obtain the backtracking capability and then change the representation to speed up such computations. The advantage of our approach is that, at the end, when one goes back to original transformer, the results are easier to analyse (for example, if one wants to obtain the first n results).

Van der Ploeg and Kiselyov [15] provide a technique for improving theoretical running times of different constructions that handle reflection, including the backtracking monad transformer. While theoretically good, their implementation has big constant factors, and our optimisation achieves better running times (as long as reflection is only needed at the end of the computation).

Jaskelioff and Rivas [12] present a simple technique for obtaining efficient implementations of non-determinism monads and alternative applicative functors.

7.4 Free Alternatives

Capriotti and Kaposi [3] study the free Applicative construction, but admittedly have no approach for the free Alternative.

Kmett's `free`[6] package does contain a definition of the free Alternative construction that is, implicitly, based on the right-based definition of the Day convolution:

$$\mathbf{data}\ (\star')\ f\ g\ a = \forall b.\mathsf{Day}'\ (f\ b)\ (g\ (b \to a))$$

7.5 Applications

Search Heuristics Schrijvers et al. [18] construct a free monad transformer for the non-deterministic choice operator in order to expose the search tree structure and apply pruning heuristics. After pruning, the resulting search tree is reflected back into the underlying non-determinism monad. Their work differs from ours in that they do not enforce any of the non-determinism axioms, in fact, they pertinently wish to observe the original syntactic structure.

Interleaved Parsers Swierstra and Dijkstra [21] have proposed their interleaving combinator as a generalisation of earlier combinators for permutations [1] and merged lists. Brown [2] provides a transformer for interleaving Alternatives that are also Monads. His approach provides both more features (e.g., early termination) and fewer (e.g., the transformed Alternative is only Applicative).

8. Conclusions

This paper has introduced the generalised notion of near-semirings, and defined the free and novel double Cayley construction generically. By exposing the fact that the instances of MonadPlus and Alternative are near-semirings in the category of endofunctors, we have then obtained these useful constructions for free.

We have shown how the free construction provides a clean slate for applying search heuristics to non-determinism monads and for interleaving applicative parsers. Moreover, our experimental evaluation witnesses the time complexity improvement brought by the double Cayley construction.

Not all MonadPlus and Alternative instances proposed in the literature or found "in the wild" are near-semirings. It would be interesting to investigate what algebraic structures underpin them. In particular, MonadPlus instances satisfying the left-catch axiom could be related to *dioids*, and their categorical generalisation [6].

Acknowledgments

The authors are grateful to the members of IFIP WG 2.1 for preliminary discussions about the topic of this paper.

Part of this work was funded by the Flemish Fund for Scientific Research (FWO) and Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) PICT 2009-15.

[6] `http://hackage.haskell.org/package/free`

References

[1] Baars, A.I., Löh, A., Swierstra, S.D.: Parsing permutation phrases. J. Funct. Program. 14(6), 635–646 (2004)

[2] Brown, N.: The InterleaveT Abstraction: Alternative with Flexible Ordering. The Monad.Reader 17, 13–33 (2011)

[3] Capriotti, P., Kaposi, A.: Free applicative functors. In: Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP 2014, Grenoble, France, 12 April 2014. pp. 2–30 (2014)

[4] Day, B.: Note on monoidal localisation. Bulletin of the Australian Mathematical Society 8, 1–16 (2 1973)

[5] Fischer, S.: Reinventing haskell backtracking. In: Fischer, S., Maehle, E., Reischuk, R. (eds.) GI Jahrestagung. LNI, vol. 154, pp. 2875–2888. GI (2009)

[6] Grandis, M.: Cubical monads and their symmetries. Rendiconti dell'Istituto di Matematica dell'Universitá di Trieste 25, 223–261 (1993)

[7] Hinze, R.: Deriving backtracking monad transformers. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 186–197. ACM, New York, NY, USA (2000)

[8] Hoogewijs, A.: Semi-Nearring Embeddings. Mededelingen van de Koninklijke Academie voor Wetenschappen, Letteren en Schone Kunsten van België, Klasse der Wetenschappen, Paleis der Academiën (1970)

[9] Hughes, R.J.M.: A novel representation of lists and its application to the function reverse. Information Processing Letters 22(3), 141 – 144 (1986)

[10] Hutton, G., Jaskelioff, M., Gill, A.: Factorising folds for faster functions. Journal of Functional Programming 20(Special Issue 3-4), 353–373 (2010)

[11] Hutton, G., Meijer, E.: Monadic Parsing in Haskell. Journal of Functional Programming 8(4), 437–444 (Jul 1998)

[12] Jaskelioff, M., Rivas, E.: A smart view on datatypes. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (2015)

[13] Krishna, K.V., Chatterjee, N.: Representation of near-semirings and approximation of their categories. Southeast Asian Bulletin of Mathematics 31, 903 – 914 (2007)

[14] McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(01), 1–13 (2008)

[15] Ploeg, A.v.d., Kiselyov, O.: Reflection without remorse: Revealing a hidden sequence to speed up monadic reflection. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 133–144. Haskell '14, ACM, New York, NY, USA (2014)

[16] Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: ESOP. LNCS, vol. 5502, pp. 80–94. Springer (2009)

[17] Rivas, E., Jaskelioff, M.: Notions of computation as monoids. CoRR abs/1406.4823 (2014), `http://arxiv.org/abs/1406.4823`

[18] Schrijvers, T., Wu, N., Desouter, B., Demoen, B.: Heuristics entwined with handlers combined. In: PPDP 2014 (2014)

[19] Spivey, J.M.: Algebras for combinatorial search. Journal of Functional Programming 19, 469–487 (7 2009)

[20] Statman, R.: Near semi-rings and lambda calculus. In: Dowek, G. (ed.) Rewriting and Typed Lambda Calculi, LNCS, vol. 8560, pp. 410–424. Springer International Publishing (2014)

[21] Swierstra, D., Dijkstra, A.: Parse your options. In: Achten, P., Koopman, P. (eds.) The Beauty of Functional Code, LNCS, vol. 8106, pp. 234–249. Springer Berlin Heidelberg (2013)

[22] Voigtländer, J.: Asymptotic improvement of computations over free monads. In: Paulin-Mohring, C., Audebaud, P. (eds.) Mathematics of Program Construction, Marseille, France, Proceedings. LNCS, vol. 5133, pp. 388–403. Springer-Verlag (Jul 2008)