Dioids for computational effects

Ezequiel Postan
DCC–ECEN
Facultad de Cs. Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Exequiel Rivas and Mauro Jaskelioff
DCC–ECEN
Facultad de Cs. Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
and
CIFASIS–CONICET

***Abstract*—There are different algebraic structures that one can use to model notions of computation. The most well-known are monads, but lately, applicative functors have been gaining popularity. These two structures can be understood as instances of the unifying notion of monoid in a monoidal category. When dealing with non-determinism, it is usual to extend monads and applicative functors with additional structure. However, depending on the desired non-determinism, there are different options of interaction between the existing and the additional structure. This article studies one of those options, which is captured algebraically by dioids. We generalise dioids to dioid categories and show how dioids in such a category model non-determinism in monads and applicative functors. Moreover, we study the construction of free dioids in a programming context.**

I. Introduction

Algebraic structures have been central to the modelling of computational effects. For example, monads [1], [2], [3], [4] have been used to model many computational effects such as global state, exceptions, environments, input/output, and continuations. More recently, applicative functors [5] are becoming popular in diverse applications such as modelling parsers [6], characterising traversals [7], [8], and in combination with monads to obtain concurrent queries [9].

While monads and applicative functors are two different algebraic structures, they have a unified framework. Rivas and Jaskelioff [10] have shown that both of them can be seen as instances of a same unifying concept: monoids in a monoidal category [11]. This unification of concepts is extremely useful, as it allows us to translate concepts, properties, and techniques from one structure to the other. For example, through the unified framework, an old optimisation technique for lists [12] is shown to be essentially the same as a newer one for monads [13], and led to the discovery of a new one for applicative functors [10] by means of a simple translation.

In many applications of monads and applicative functors one has to deal with non-determinism. There are different flavours of non-determinism [14], but in functional programming the most common are deep backtracking and shallow backtracking [15]. When modelling deep backtracking, the algebraic structure that arises is near-semirings. This insight lead to a unified framework for deep-backtracking non-determinism in monads and applicative functors [16]. If, on the other hand, one wants to model shallow backtracking, then one arrives at the algebraic structure of dioids [17].

This article studies the shallow-backtracking variant of non-determinism by studying the categories that support the definition of dioids, namely dioid categories. Working at this level of abstraction allows us to obtain a unified model of shallow-backtracking non-determinism for both monads and applicative functors. Moreover, we study the construction of free dioids. Intuitively, free dioids can be thought of as the programs that can be written when only the dioid interface is exposed, and therefore provide a canonical representation for programs structured as a computation with shallow-backtracking non-determinism.

The article is structured as follows: In Section II we introduce monoids and monoidal categories, and show how they provide a unified framework to study the notions of monads and applicative functors. In Section III, we introduce dioids and dioid categories. Moreover, we show how these categories provide a unified framework to study shallow non-determinism in monads and applicative functors. In Section IV, we turn to the construction of free dioids. We provide a formula that allows to construct dioids on Set (the category of sets and functions) and to construct the free dioid applicative. Unfortunately, it does not allow us to express the free dioid monad. Finally, in Section V, we conclude.

In the rest of this article, unless we explicitly say otherwise, when we write non-determinism we mean shallow-backtracking non-determinism.

II. Monoids

We start by studying monoids and its generalisation: monoids in monoidal categories. In order to keep the ideas close to programming practice, we express the different concrete constructions in an idealised functional programming language with syntax inspired by Haskell.

A. Monoids in Programming

In functional languages, such as Haskell, algebraic structures may be implemented using type classes. For example, for monoids we may declare:

978-1-5386-3057-0/17/$31.00 © 2017 IEEE

```
class Monoid m where
  u  ::  ()  → m
  (⊗) :: m × m → m
```

which means that a type m is a monoid whenever it is equipped with a chosen binary operation $\otimes$ (the *multiplication*) and a nullary operation u (the *unit*). Instances of this class are expected to satisfy the monoid laws

$$a \otimes (\mathsf{u}\ ()) = a \tag{1}$$
$$(\mathsf{u}\ ()) \otimes a = a \tag{2}$$
$$a \otimes (b \otimes c) = (a \otimes b) \otimes c \tag{3}$$

which state that $\otimes$ is associative, and that u is a right and left unit for it. For example, we may declare that the type Integer of arbitrary-precision integers is a monoid of addition and zero:

```
instance Monoid Integer where
  u ()  = 0
  x ⊗ y = x + y
```

Another important example is the monoid of endofunctions over a type a. Without mathematical jargon, they are just functions from a to a with composition as multiplication and the identity function as unit.

```
data Endo a where
  Endo :: (a → a) → Endo a
instance Monoid (Endo a) where
  u ()                = Endo id
  (Endo f) ⊗ (Endo g) = Endo (f ∘ g)
```

When studying an algebraic structure, an important concept is that of homomorphism: structure-preserving maps between instances of the algebraic structure. In the case of monoids, they are defined as follows: let M_1 and M_2 be instances of Monoid. A *monoid homomorphism* is a function $f :: \mathsf{M}_1 \to \mathsf{M}_2$ such that the monoid instances are preserved:

$$f\ (\mathsf{u}\ ()) = \mathsf{u}\ ()$$
$$f\ (a \otimes b) = f\ a \otimes f\ b$$

The type of f determines that the operations u and $\otimes$ on the left-hand side come from the Monoid instance of M_1, while those on the right-hand side come from the Monoid instance of M_2.

For example, the following is a monoid homomorphism from Integer to Endo Integer.

```
rep :: Integer → Endo Integer
rep x = Endo (λy → x + y)
```

We want to program with algebraic structures such as Monoid above, but we may also want to generalise from types to types constructors, and from the Cartesian product $\times$ to other ways of putting two things together. The appropriate generalisation is the notion of monoid in a monoidal category.

B. Categorification

Monoidal categories generalise the notion of monoids to categories $\mathcal{C}$.

Definition II.1. A *monoidal category* is a sextuple $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$, where:

- $\mathcal{C}$ is a category;
- $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is a bifunctor;
- I is an object of $\mathcal{C}$;
  The bifunctor $\otimes$ and I generalise $\times$ and $()$ in the code above.
- α, λ, and ρ are natural isomorphisms:

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

All these natural transformations should obey coherence laws [11].

Given a monoidal category, we can define the notion of a monoid in it.

Definition II.2. A *monoid in a monoidal category* is an object M, together with operations $e : I \to M$ and $m : M \otimes M \to M$, for which the following laws hold:

$$m \circ (e \otimes \mathsf{id}) = \lambda \tag{4}$$
$$m \circ (\mathsf{id} \otimes e) = \rho \tag{5}$$
$$m \circ (\mathsf{id} \otimes m) = m \circ (m \otimes \mathsf{id}) \circ \alpha \tag{6}$$

The laws for monoids in monoidal categories are the corresponding generalisations of equations (1), (2), and (3).

For example, the category Set of sets and functions is a monoidal category with the Cartesian product as its tensor, and singleton sets as unit object, and monoids in this monoidal category reduce to ordinary monoids.

C. Functors

In this article we are mainly interested in the category of endofunctors and natural transformations. Endofunctors are type constructors that can map a function on the underlying type. For example, lists are functors since

1) given a type, say Integer, they construct a type of list of Integers;
2) given a function, say from Integers to Booleans, they can apply the function to every Integer to obtain a list of Booleans.

More precisely, a functor is an instance of the following class

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

where the laws fmap id = id, and fmap $(f \circ g)$ = fmap $f \circ$ fmap g hold.

For example, the identity type constructor is a functor.

```
data Identity where
  Id :: a → Identity a
instance Functor Identity where
  fmap f (Id a) = Id (f a)
```

Furthermore, the composition of two functors is a functor.

```
data (f ∘ g) a where
  Comp :: f (g a) → (f ∘ g) a
instance (Functor f, Functor g) ⇒ Functor (f ∘ g) where
  fmap f (Comp x) = Comp (fmap (fmap f) x)
```

In this instance, to the left of the ⇒ symbol, we note the requirement that f and g must be functors.

Natural transformations are functions between functors which are polymorphic on the underlying type [18].

```
type (f ·→ g) = ∀x. f x → g x
```

As a last functor example, we consider the Maybe data type constructor. This functor is commonly used to represent either a value or a failure, and it is sometimes known as Option.

```
data Maybe a where
  Nothing :: Maybe a
  Just    :: a → Maybe a
```

The functor instance simply maps a function if we have a value, or it does nothing otherwise.

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

D. Monads

The category of endofunctors and natural transformations can be given a monoidal structure by choosing the tensor $\otimes$ to be the composition of functors $\circ$, and the object I to be the identity functor Identity. This monoidal category is *strict*, which means that the three natural transformations λ, ρ, and α, which complete the monoidal category, are identities.

The monoids in this monoidal category are functors m with operations e and m of type Identity $\dot{\to}$ m, and $m \circ m \dot{\to} m$ respectively. Expanding the definitions of natural transformation, identity functor, and functor composition, and renaming e to η and m to μ, we arrive at the following type class:

```
class Functor m ⇒ Triple m where
  η :: a → m a
  μ :: m (m a) → m a
```

and the general monoid laws (4), (5), and (6) become:

$$\mu \circ \eta = \mathsf{id}$$
$$\mu \circ \mathsf{fmap}\ \eta = \mathsf{id}$$
$$\mu \circ \mathsf{fmap}\ \mu = \mu \circ \mu$$

A monoid in the monoidal category with functor composition as tensor is none other than a *monad*. Monads have other presentations. For example, in Haskell, monads are defined as:

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

subject to some laws which correspond to the laws above under the following equivalence. The classes Triple and Monad can be seen to be equivalent by noting that η = return, that μ can be defined as $\mu\ x = (x >\!\!>\!\!= \mathsf{id})$, and that $(>\!\!>\!\!=)$ can be defined as $(x >\!\!>\!\!= k) = \mu\ (\mathsf{fmap}\ k\ x)$. Notice that Monad m does not require the type constructor m to be a functor: the fmap operation is derivable from the Monad instance by defining fmap $f\ v = (v >\!\!>\!\!= \mathsf{return} \circ f)$.

Remark II.3 (Currying)**.** The function $(>\!\!>\!\!=)$ in the Monad class does not take two arguments; instead it takes only one argument and returns a function which takes the other argument and finally delivers the result. Writing functions in this style is equivalent to writing functions with two arguments, as the following conversion functions show:

```
curry :: (a × b → c) → (a → b → c)
curry f a b = f (a, b)
uncurry :: (a → b → c) → (a × b → c)
uncurry f (a, b) = f a b
```

In the rest of the presentation we use two argument functions or their curried form indistinctly.

Both lists and Maybe are functors which are also Monad instances. We provide the instance for Maybe, which will be one of our main examples.

```
instance Monad Maybe where
  return x       = Just x
  Nothing >>= f  = Nothing
  (Just x) >>= f = f x
```

The category of endofunctors can be given other monoidal structures, and therefore monads are not the only monoids in the category of endofunctors. Another important class are *applicative functors*, introduced by McBride and Paterson [5] as a way to capture certain effectful computations that do not fit well in the monadic framework.

E. Applicative functors

Applicative functors are based on a category of endofunctors, but with different tensor than monads: the *Day convolution* [19]. The Day convolution may be implemented as follows:

```
data (⋆) f g a where
  Day :: f (b → a) × (g b) → (f ⋆ g) a
instance (Functor f, Functor g) ⇒ Functor (f ⋆ g) where
  fmap h (Day ff gx) = Day (fmap (λf → h ∘ f) ff) gx
```

Just like for composition of functors $(\circ)$, the I object for Day convolution is the identity functor. However, in this case the monoidal category is not strict. The isomorphisms λ, ρ and α for this monoidal category are as follows (we give only one direction of each isomorphism).

$\lambda :: \mathsf{Functor}\ f \Rightarrow (\mathsf{Identity} \star f)\ a \to f\ a$
$\lambda\ (\mathsf{Day}\ (\mathsf{Identity}\ f)\ x) = \mathsf{fmap}\ f\ x$

$\rho :: \mathsf{Functor}\ f \Rightarrow (f \star \mathsf{Identity})\ a \to f\ a$
$\rho\ (\mathsf{Day}\ f\ (\mathsf{Identity}\ b)) = \mathsf{fmap}\ (\lambda h \to h\ b)\ f$

$\alpha :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g) \Rightarrow$
$\quad (f \star (g \star h))\ a \to ((f \star g) \star h)\ a$
$\alpha\ (\mathsf{Day}\ f\ (\mathsf{Day}\ g\ z)) = \mathsf{Day}\ (\mathsf{Day}\ (\mathsf{fmap}\ (\circ)\ f)\ g)\ z$

The monoids in this monoidal category are functors f with operations e and m of type $\mathsf{Identity} \dot{\to} f$, and $f \star f \dot{\to} f$ respectively. Expanding the definitions of natural transformation, identity functor, and Day convolution, and renaming e to pure and m to $\circledast$, we arrive at the following type class:

class $\mathsf{Functor}\ f \Rightarrow \mathsf{Applicative}\ f$ **where**
$\mathsf{pure} :: a \to f\ a$
$(\circledast) :: f\ (b \to a) \times f\ b \to f\ a$

Instantiating the general monoid laws (4), (5), and (6) to this monoidal category, we obtain the applicative laws:

$$\mathsf{pure}\ f \circledast u = \mathsf{fmap}\ f\ u$$
$$u \circledast \mathsf{pure}\ x = \mathsf{fmap}\ (\lambda h \to h\ x)\ u$$
$$(\mathsf{fmap}\ (\circ)\ u \circledast v) \circledast w = u \circledast (v \circledast w)$$

By generalising monoids to monoids in monoidal categories, we were able to show that two different structures used in programming are instances of the same abstract construction.

Remark II.4. Monad and Applicative type classes are not totally independent: every instance of the former is an instance of the latter, as it is reflected in the next code.

instance $\mathsf{Monad}\ m \Rightarrow \mathsf{Applicative}\ m$ **where**
$\mathsf{pure}\ x = \mathsf{return}\ x$
$u \circledast v = u \gg\!= \lambda f \to v \gg\!= \lambda x \to \mathsf{return}\ (f\ x)$

Therefore every Monad determines an Applicative, but not the other way. As an example of an Applicative which is not a Monad, consider the constant functor on a monoid:

data $\mathsf{K}\ x\ a$ **where**
$\mathsf{MK} :: x \to \mathsf{K}\ x\ a$

instance $\mathsf{Functor}\ (\mathsf{K}\ x)$ **where**
$\mathsf{fmap}\ f\ (\mathsf{MK}\ x) = \mathsf{MK}\ x$

instance $\mathsf{Monoid}\ x \Rightarrow \mathsf{Applicative}\ (\mathsf{K}\ x)$ **where**
$\mathsf{pure}\ a = \mathsf{MK}\ \mathsf{u}$
$(\mathsf{MK}\ x) \circledast (\mathsf{MK}\ y) = \mathsf{MK}\ (x \otimes y)$

III. Dioids

We extend the notion of monoids in order to account for non-determinism. More precisely, we introduce *dioids*, which extend monoids with additional monoid operations, which we denote with $\oplus$ and z. Whereas the multiplicative monoid gives a model of sequencing, the $\oplus$ operation models a non-deterministic choice, and models z the absence of choice. What makes this structure a good model for shallow-backtracking non-determinism, and what differentiates it from other models, is its interaction with the existing monoid structure.

A. Set dioids

Dioids are an algebraic structure, so we might declare them as a type class, just like we did with monoids:

class $\mathsf{Dioid}\ d$ **where**
$\mathsf{z} :: () \to d$
$\mathsf{u} :: () \to d$
$(\oplus) :: d \times d \to d$
$(\otimes) :: d \times d \to d$

This time, we expect the following laws to be satisfied:

$$a \otimes (\mathsf{u}\ ()) = a \tag{7}$$
$$(\mathsf{u}\ ()) \otimes a = a \tag{8}$$
$$a \otimes (b \otimes c) = (a \otimes b) \otimes c \tag{9}$$
$$a \oplus (\mathsf{z}\ ()) = a \tag{10}$$
$$(\mathsf{z}\ ()) \oplus a = a \tag{11}$$
$$a \oplus (b \oplus c) = (a \oplus b) \oplus c \tag{12}$$
$$(\mathsf{z}\ ()) \otimes a = \mathsf{z}\ () \tag{13}$$
$$(\mathsf{u}\ ()) \oplus a = \mathsf{u}\ () \tag{14}$$

The laws 7 to 12 express that d is a monoid with respect to $(\otimes, \mathsf{u})$ and $(\oplus, \mathsf{z})$. Laws 13 and 14 relate these (otherwise independent) monoid structures, by saying that the unit of one is left absorbent of the other. Because of this left-bias, we might call these *left dioids* instead of just *dioids*.

Every bounded distributive lattice is a dioid for which $(\otimes)$ commutes, perhaps the most classical example is Bool. The following is an instance in which $(\otimes)$ does not commute:

type $\mathsf{BinFun}\ a = a \times a \to a$
instance $\mathsf{Dioid}\ (\mathsf{BinFun}\ a)$ **where**
$\mathsf{z} = \lambda(a, b) \to a$
$\mathsf{u} = \lambda(a, b) \to b$
$f \oplus g = \lambda(a, b) \to f\ (g\ (a, b), b)$
$f \otimes g = \lambda(a, b) \to f\ (a, g\ (a, b))$

In models of deep-backtracking non-determinism, law 14 is replaced by a distribution.

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

This makes the structure a near-semiring. See the work of Rivas, Jaskelioff and Schrijvers [16] for details. In this case it is possible to explore different results, whereas in the shallow case, we explore possible results in order only until one is found.

Let D_1 and D_2 be instances of Dioid. A *dioid homomorphism* is a function $f :: \mathsf{D}_1 \to \mathsf{D}_2$ such that the dioid instances are preserved:

$$f\ (\mathsf{u}\ ()) = \mathsf{u}\ ()$$
$$f\ (\mathsf{z}\ ()) = \mathsf{z}\ ()$$
$$f\ (a \otimes b) = f\ a \otimes f\ b$$
$$f\ (a \oplus b) = f\ a \oplus f\ b$$

The type of f determines that the operations u, z, $\otimes$, and $\oplus$ on the left-hand side come from the Dioid instance of

D_1, while those on the right-hand side come from the Dioid instance of D_2.

B. Categorification

Just as monoidal categories provide the right setting to express the notion of monoid in full generality, we now look for the analogous categorical structure to express the notion of dioid.

Definition III.1. A *dioid category* is a tuple $(\mathcal{C}, \otimes, I, \alpha_\otimes, \lambda_\otimes, \rho_\otimes, \oplus, Z, \alpha_\oplus, \lambda_\oplus, \rho_\oplus, \kappa_\otimes, \kappa_\oplus)$ where:

- $\mathcal{C}$ is a category;
- $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ and $\oplus : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ are bifunctors;
- I and Z are objects of $\mathcal{C}$;
- $\alpha_\otimes$, $\lambda_\otimes$, $\rho_\otimes$, $\alpha_\oplus$, $\lambda_\oplus$, and $\rho_\oplus$ are natural isomorphisms:

$$\begin{aligned}
\alpha_\otimes &: A \otimes (B \otimes C) \cong (A \otimes B) \otimes C \\
\lambda_\otimes &: I \otimes A \cong A \\
\rho_\otimes &: A \otimes I \cong A \\
\alpha_\oplus &: A \oplus (B \oplus C) \cong (A \oplus B) \oplus C \\
\lambda_\oplus &: Z \oplus A \cong A \\
\rho_\oplus &: A \oplus Z \cong A
\end{aligned}$$

- $\kappa_\otimes$ and $\kappa_\oplus$ are natural transformations:

$$\begin{aligned}
\kappa_\otimes &: Z \otimes A \to Z \\
\kappa_\oplus &: I \oplus A \to I
\end{aligned}$$

Again, we expect these natural transformations to obey some coherence laws, which include those of $(\mathcal{C}, \otimes, I, \alpha_\otimes, \lambda_\otimes, \rho_\otimes)$ and $(\mathcal{C}, \oplus, Z, \alpha_\oplus, \lambda_\oplus, \rho_\oplus)$ being monoidal categories.

Given a dioid category, we can define what a dioid is.

Definition III.2. A *dioid in a dioid category* is an object D, together with operations $z : Z \to D$, $e : I \to D$, $s : D \oplus D \to D$ and $m : D \otimes D \to D$ for which the following laws hold:

$$m \circ (e \otimes \mathsf{id}) = \lambda_\otimes \tag{15}$$
$$m \circ (\mathsf{id} \otimes e) = \rho_\otimes \tag{16}$$
$$m \circ (\mathsf{id} \otimes m) = m \circ (m \otimes \mathsf{id}) \circ \alpha_\otimes \tag{17}$$
$$s \circ (z \oplus \mathsf{id}) = \lambda_\oplus \tag{18}$$
$$s \circ (\mathsf{id} \oplus z) = \rho_\oplus \tag{19}$$
$$s \circ (\mathsf{id} \oplus s) = s \circ (s \oplus \mathsf{id}) \circ \alpha_\oplus \tag{20}$$
$$m \circ (z \otimes \mathsf{id}) = z \circ \kappa_\otimes \tag{21}$$
$$s \circ (e \oplus \mathsf{id}) = e \circ \kappa_\oplus \tag{22}$$

The laws for dioids in dioid categories are the corresponding generalisations of equations for dioids.

To recover ordinary dioids, we take the category of sets and functions Set with $Z = I = \{*\}$ and $\oplus = \otimes = \times$. Notice that $\kappa_\otimes = \kappa_\oplus = \pi_1 : \{*\} \times A \to \{*\}$ is not a natural isomorphism, but only a natural transformation.

Lemma III.3. In general, from a monoidal category $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$ with Cartesian structure (terminal object and binary products), we obtain a dioid category $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho, \times, 1, \alpha_\times, \pi_2, \pi_1, \kappa_\otimes, \pi_1)$, where:

$$\begin{aligned}
\alpha_\times &= \langle\langle \pi_1, \pi_1 \circ \pi_2\rangle, \pi_2 \circ \pi_2\rangle : A \times (B \times C) \to (A \times B) \times C \\
\kappa_\otimes &= !_{1 \otimes A} : 1 \otimes A \to 1
\end{aligned}$$

We close our discussion on categorification of dioids by giving the generalisation of dioid homomorphism, which is a direct generalisation of that for ordinary dioids.

Definition III.4 (Dioid homomorphism)**.** A *dioid homomorphism* from a dioid $(D_1, z_1, e_1, s_1, m_1)$ to a dioid $(D_2, z_2, e_2, s_2, m_2)$ is a morphism $f : D_1 \to D_2$ such that the following equations hold:

$$\begin{aligned}
f \circ e_1 &= e_2 \\
f \circ z_1 &= z_2 \\
f \circ m_1 &= m_2 \circ (f \otimes f) \\
f \circ s_1 &= s_2 \circ (f \oplus f)
\end{aligned}$$

C. Cartesian structure for functors

We can use Lemma III.3 to extend the monoidal categories of endofunctors discussed in Sections II-D and II-E to dioid categories. We need to establish that the category of endofunctors on a category $\mathcal{C}$ has terminal object and binary products. If the base category $\mathcal{C}$ has terminal object, then the constant functor to the terminal object is the terminal object on the category of endofunctors.

```
data K1 a where
  K1 :: () → K1
instance Functor K1 where
  fmap f (K1 ()) = K1 ()
```

Here, the unit type `()` represents the terminal object. Similarly, a product of endofunctors is defined in terms of product for objects in the base category, in a point-wise fashion:

```
data (f × g) a where
  Pair :: f a × g a → (f × g) a
instance (Functor f, Functor g) ⇒ Functor (f × g) where
  fmap h (Pair (fa, ga)) = Pair (fmap h fa, fmap h ga)
```

Thus, the endofunctors form a monoidal category with the Cartesian structure, and the monoidal categories supporting monads and applicative functors can be extended to dioid categories.

D. Non-determinism Monads

By Lemma III.3 and the Cartesian structure introduced above, we know that the category of endofunctors forms a dioid category by choosing tensor $\otimes$ to be composition of functors $\circ$, tensor $\oplus$ to be the binary product of functors $\times$, the object I to be the identity functor $\mathsf{Identity}$ and the object Z to be the constant terminal functor K_1.

The dioids in this dioid category are functors d with operations ζ, η, σ and μ of type $\mathsf{K}_1 \dot{\to} d$, $\mathsf{Identity} \dot{\to} d$, $d \times d \dot{\to} d$ and $d \circ d \dot{\to} d$ respectively. If we expand the definitions, we can present this information in a type class:

```
class DioidM d where
  ζ :: ()          → d a
  η :: a           → d a
  σ :: d a × d a   → d a
  μ :: d (d a)     → d a
```

subject to the laws:

$$\mu \circ \eta = \mathsf{id}$$
$$\mu \circ \mathsf{fmap}\ \eta = \mathsf{id}$$
$$\mu \circ \mathsf{fmap}\ \mu = \mu \circ \mu$$
$$\sigma \circ \mathsf{pair}\ \mathsf{id}\ \mathsf{zero} = \mathsf{fst}$$
$$\sigma \circ \mathsf{pair}\ \mathsf{zero}\ \mathsf{id} = \mathsf{snd}$$
$$\sigma \circ \mathsf{pair}\ \mathsf{id}\ \sigma = \sigma \circ \mathsf{pair}\ \sigma\ \mathsf{id} \circ \alpha$$
$$\mu \circ \zeta = \zeta$$
$$\sigma \circ \mathsf{pair}\ \eta\ \mathsf{id} = \eta \circ \mathsf{fst}$$

where $\mathsf{pair}\ f\ g\ (x, y) = (f\ x, g\ y)$ and $\alpha\ (x, (y, z)) = ((x, y), z)$. The operations η and μ form an instance of Triple. In this way, a dioid might be seen as an extended monad. This justifies the equivalent type class

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

subject to the following laws

$$\mathsf{mplus}\ \mathsf{mzero}\ u = u$$
$$\mathsf{mplus}\ u\ \mathsf{mzero} = u$$
$$\mathsf{mplus}\ u\ (\mathsf{mplus}\ v\ w) = \mathsf{mplus}\ (\mathsf{mplus}\ u\ v)\ w$$
$$\mathsf{mzero} \gg\!= f = \mathsf{mzero}$$
$$\mathsf{mplus}\ (\mathsf{return}\ x)\ u = \mathsf{return}\ x$$

in addition to those of monads.

Perhaps the most representative instance of MonadPlus subject to these axioms is Maybe.

```
instance MonadPlus Maybe where
  mzero             = Nothing
  mplus Nothing v   = v
  mplus (Just x) v  = Just x
```

An important non-example of MonadPlus subject to these axioms are lists. While the empty list and list concatenation would give an implementation for mzero and mplus, such implementation would not satisfy the law $\mathsf{mplus}\ (\mathsf{return}\ x)\ u = \mathsf{return}\ x$. (In fact, lists are the canonical example of deep-backtracking non-determinism.)

E. Non-determinism Applicative Functors

We turn again to Lemma III.3 to obtain a dioid category of endofunctors, but this time with the Day convolution as a tensor instead of functor composition.

The dioids in this dioid category are functors d with operations empty, pure, $(\langle|\rangle)$ and $(\circledast)$ of type $\mathsf{K}_1 \dot{\to} d$, $\mathsf{Identity} \dot{\to} d$, $d \times d \dot{\to} d$ and $d \star d \dot{\to} d$ respectively. If we expand the definitions, we can present this information in a type class:

```
class DioidF d where
  empty :: ()              → d a
  pure  :: a               → d a
  (⟨|⟩) :: d a × d a       → d a
  (⊛)   :: d (b → a) × d b → d a
```

As we did with monads, we separate the applicative functor contained in this type class, and create a class that extends applicative functors with the additional information:

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

By instantiating the laws for dioids, we obtain the following laws for Alternative, which are additional to those of the underlying Applicative instance.

$$\mathsf{empty}\ \langle|\rangle\ u = u$$
$$u\ \langle|\rangle\ \mathsf{empty} = u$$
$$u\ \langle|\rangle\ (v\ \langle|\rangle\ w) = (u\ \langle|\rangle\ v)\ \langle|\rangle\ w$$
$$\mathsf{empty} \circledast u = \mathsf{empty}$$
$$\mathsf{pure}\ x\ \langle|\rangle\ u = \mathsf{pure}\ x$$

We can extend Remark II.4 to the type classes MonadPlus and Alternative, and obtain the following result.

```
instance MonadPlus m ⇒ Alternative m where
  empty = mzero
  u ⟨|⟩ v = mplus u v
```

In fact, most Alternative instances found in programming practice are actually MonadPlus instances. An example of an Alternative which is not a MonadPlus is the constant functor on a Dioid.

```
instance Dioid d ⇒ Alternative (K d) where
  empty               = MK z
  (MK d1) ⟨|⟩ (MK d2) = MK (d1 ⊕ d2)
```

IV. Free structures

Free structures are a fundamental tool in universal algebra, as in some sense they provide the most *general* models of an algebraic structure, free of any additional equation over terms. In computer science, this phenomenon is often referred to as the *no junk, no confusion* principle [20]. In our setting, we employ free structures as a device to work with those programs that only use the structure under analysis.

A. Free monoids

The notion of free ordinary monoid is captured by a universal property. Formally, we say that the type $\mathsf{FreeMon}_a$ is the free monoid over a when:

- $\mathsf{FreeMon}_a$ is an instance of Monoid;
- there is a function $\mathsf{ins} :: a \to \mathsf{FreeMon}_a$;

- for any Monoid instance m and function $f :: a \to m$, there exists a unique monoid homomorphism $\mathsf{univ}\ f :: \mathsf{FreeMon}_a \to m$ such that $\mathsf{univ}\ f \circ \mathsf{ins} = f$.

While mathematically precise, this definition is not constructive: it does not provide a procedure to construct such $\mathsf{FreeMon}_a$ from a given a. A possible technique to find a concrete construction is to provide a unique form for monoidal terms, such that two terms that are equal by the monoid laws are represented by the same term. For example, $a \otimes (\mathsf{u}\ () \otimes (b \otimes c))$ and $(a \otimes b) \otimes (c \otimes \mathsf{u}\ ())$ should have a unique representation in the data type representing the free monoid over a set which includes a, b and c. To see that two monoid expressions are equivalent, we can apply the monoid laws in a methodological way:

- every atom a is replaced by $a \otimes u\ ()$;
- every expression associated to the left is re-associated to the right;
- every expression $u\ () \otimes t$ is reduced to t.

Applying this method to the expressions above, we obtain the term $a \otimes (b \otimes (c \otimes u\ ()))$ in both cases. After some thinking, one can conclude that every term reduces to a list of atoms ending in $u\ ()$. This observation inspires the following data type for representing canonical forms.

```
data FreeMon a where
  Nil  :: FreeMon a
  Cons :: a × FreeMon a → FreeMon a
```

This data type is equivalent to a list of as, and therefore has a monoid instance given by the empty list and list concatenation:

```
instance Monoid (FreeMon a) where
  u ()              = Nil
  Nil        ⊗ bs   = bs
  (Cons a as) ⊗ bs  = Cons a (as ⊗ bs)
```

The insertion function represents an a atom by a singleton list.

```
ins :: a → FreeMon a
ins a = Cons a Nil
```

The function univ is written by recursion on $\mathsf{FreeMon}\ a$:

```
univ :: Monoid m ⇒ (a → m) → FreeMon a → m
univ f Nil        = u ()
univ f (Cons a as) = f a ⊗ univ f as
```

Using set theory, the free monoid over a set a, i.e. lists of a, can be seen as the least solution a^* to the recursive equation:

$$a^* = \{*\} \uplus a \times a^*$$

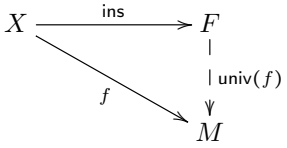
where $\uplus$ represents the disjoint union of sets.

Generalising to monoidal categories the equation becomes

$$A^* \cong I + A \otimes A^* \tag{23}$$

which gives a candidate for the free monoid in a monoidal category. Before instantiating this formula to other cases, we first review the general universal property for a free monoid in a monoidal category $(\mathcal{C}, \otimes, I)$.

Definition IV.1 (Free monoid)**.** Let X be an object, the *free monoid* over X is a monoid (F, e_F, m_F) together with a morphism $\mathsf{ins} : X \to F$ such that for any monoid (M, e_M, m_M) and morphism $f : X \to M$ there exists a unique monoid homomorphism $\mathsf{univ}(f) : F \to M$ such that $\mathsf{univ}(f) \circ \mathsf{ins} = f$. Diagrammatically, we have:



The morphism ins is called the *insertion* morphism, and univ f is known as the lifting of f.

As in the case of ordinary monoids, this definition provides an abstract characterisation for the free monoid. To obtain a concrete description, we instantiate formula 23 to the corresponding monoidal category.

For obtaining the free Monad, we apply formula 23 to the monoidal category of endofunctors with functor composition as tensor, which yields the equation

$$f^* \cong \mathsf{Identity} + f \circ f^*$$

that leads to the following data type:

```
data Free f a where
  Nil∘  :: a → Free f a
  Cons∘ :: f (Free f a) → Free f a
```

This is indeed the free monad on a functor f, with monad instance:

```
instance Functor f ⇒ Monad (Free f) where
  return x      = Nil∘ x
  Nil∘ x   >>= f = f x
  Cons∘ v >>= f = Cons∘ (fmap (>>=f) v)
```

The insertion morphism and lifting are as follows:

```
ins :: Functor f ⇒ f ·→ Free f
ins v = Cons∘ (fmap Nil∘ v)
univ :: (Functor f, Monad m) ⇒ (f ·→ m) → (Free f ·→ m)
univ f (Nil∘ x)  = return x
univ f (Cons∘ v) = f (fmap (univ f) v) >>= id
```

We now turn our focus to the monoidal category of endofunctors with Day convolution as tensor. Instantiating formula 23 to this monoidal category results in:

$$f^* \cong \mathsf{Identity} + f \star f^*$$

which leads to the following data type

```
data FreeA f a where
  Nil⋆  :: a → FreeA f a
  Cons⋆ :: f (b → a) × FreeA f b → FreeA f a
```

Again, we find the instantiation of the general formula yields the free applicative functor on a functor f, with applicative instance:

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Applicative}\ (\mathsf{FreeA}\ f)\ \textbf{where}\\
&\quad \mathsf{pure}\ x \qquad\qquad = \mathsf{Nil}_\star\ x\\
&\quad \mathsf{Nil}_\star\ h \qquad \circledast\ x = \mathsf{fmap}\ h\ x\\
&\quad \mathsf{Cons}_\star\ (h, x) \circledast y = \mathsf{Cons}_\star\ (\mathsf{fmap\ uncurry}\ h)\\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{fmap}\ (,)\ x \circledast y)
\end{aligned}
$$

where $(,)$ is the pair constructor. The insertion morphism and lifting are implemented as follows.

$$
\begin{aligned}
&\mathsf{ins} :: \mathsf{Functor}\ f \Rightarrow f \dot{\to} \mathsf{FreeA}\ f\\
&\mathsf{ins}\ v = \mathsf{Cons}_\star\ (\mathsf{fmap\ const}\ v)\ (\mathsf{Nil}_\star\ ())\\
&\mathsf{univ} :: (\mathsf{Functor}\ f, \mathsf{Applicative}\ g) \Rightarrow (f \dot{\to} g) \to (\mathsf{FreeA}\ f \dot{\to} g)\\
&\mathsf{univ}\ f\ (\mathsf{Nil}_\star\ x) \qquad = \mathsf{pure}\ x\\
&\mathsf{univ}\ f\ (\mathsf{Cons}_\star\ v\ r) = f\ v \circledast \mathsf{univ}\ f\ r
\end{aligned}
$$

Starting with the analysis of the free monoid, we have generalised the solution to monoidal categories, and then we have used this formula to obtain the free monad and the free applicative. General conditions for the existence of free monoids can be found in the work of Kelly [21]. The case of free monads and free applicative functors is analysed in detail by Rivas and Jaskelioff [10].

B. Free dioids

For constructing free dioids, it would be desirable to adapt the methodology we followed to obtain free monoids. This is, we expect to come up with a formula for ordinary free dioids, and then generalise this formula to obtain a candidate for free dioids in a dioid category.

Instead of introducing first the notion of free ordinary dioid and then generalising it, we present directly free dioids in a dioid category $(\mathcal{C}, \otimes, I, \alpha_\otimes, \lambda_\otimes, \rho_\otimes, \oplus, Z, \alpha_\oplus, \lambda_\oplus, \rho_\oplus, \kappa_\otimes, \kappa_\oplus)$, and then obtain the ordinary notion for the dioid category Set with binary products and terminal object as both additive and multiplicative structures.

Definition IV.2 (Free dioid)**.** Let X be an object, the *free dioid* over X is a dioid (F, z_F, e_F, s_F, m_F) together with a morphism $\mathsf{ins} : X \to F$ such that for any dioid (D, z_D, e_D, s_D, m_G) and morphism $f : X \to G$ there exists a unique dioid homomorphism $\mathsf{univ}(f) : F \to G$ such that $\mathsf{univ}(f) \circ \mathsf{ins} = f$.

As in the case of monoids, the presentation by universal property does not give a concrete construction for free dioids. To obtain a concrete presentation for the free ordinary dioid over a set X, we need to come up with a canonical form for dioid terms. We propose the least solution to the following recursive equations of sets:

$$X^* = 0 \uplus 1 \uplus T \tag{24}$$

$$T = X \uplus (S \times_\oplus 1) \uplus (S \times_\oplus T) \uplus (M \times_\otimes 0) \uplus (M \times_\otimes T) \tag{25}$$

$$S = X \uplus (M \times_\otimes 0) \uplus (M \times_\otimes T) \tag{26}$$

$$M = X \uplus (S \times_\oplus 1) \uplus (S \times_\oplus T) \tag{27}$$

where $0 = 1 = \{*\}$ and $\times_\otimes = \times_\oplus = \times$. Although these last renamings are unnecessary at this point, they will become useful when we generalise these equations to dioid categories. Performing some simplifications, we can implement these equations as a data type:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Free}\ a\ \textbf{where}\\
&\quad \mathsf{Zero} :: \mathsf{Free}\ a\\
&\quad \mathsf{Unit} :: \mathsf{Free}\ a\\
&\quad \mathsf{Term} :: \mathsf{Term}\ a \to \mathsf{Free}\ a\\
&\textbf{data}\ \mathsf{Term}\ a\ \textbf{where}\\
&\quad \mathsf{LiftT} :: a \to \mathsf{Term}\ a\\
&\quad \mathsf{SumT}^1 :: \mathsf{Sum}\ a \to \mathsf{Term}\ a\\
&\quad \mathsf{SumT}^2 :: \mathsf{Sum}\ a \times \mathsf{Term}\ a \to \mathsf{Term}\ a\\
&\quad \mathsf{MultT}^1 :: \mathsf{Mult}\ a \to \mathsf{Term}\ a\\
&\quad \mathsf{MultT}^2 :: \mathsf{Mult}\ a \times \mathsf{Term}\ a \to \mathsf{Term}\ a\\
&\textbf{data}\ \mathsf{Sum}\ a\ \textbf{where}\\
&\quad \mathsf{LiftS} :: a \to \mathsf{Sum}\ a\\
&\quad \mathsf{MultS}^1 :: \mathsf{Mult}\ a \to \mathsf{Sum}\ a\\
&\quad \mathsf{MultS}^2 :: \mathsf{Mult}\ a \times \mathsf{Term}\ a \to \mathsf{Sum}\ a\\
&\textbf{data}\ \mathsf{Mult}\ a\ \textbf{where}\\
&\quad \mathsf{LiftM} :: a \to \mathsf{Mult}\ a\\
&\quad \mathsf{SumM}^1 :: \mathsf{Sum}\ a \to \mathsf{Mult}\ a\\
&\quad \mathsf{SumM}^2 :: \mathsf{Sum}\ a \times \mathsf{Term}\ a \to \mathsf{Mult}\ a
\end{aligned}
$$

The dioid operations for $\mathsf{Free}\ a$ are not difficult to write, although their length can be intimidating. Two auxiliary functions $(\oplus_T)$ and $(\otimes_T)$ are provided, as they help to structure the multiplication and addition. We give the implementation only for $(\otimes_T)$, as the implementation for $(\oplus_T)$ is dual.

$$
\begin{aligned}
&(\otimes_T) :: \mathsf{Term}\ a \to \mathsf{Free}\ a \to \mathsf{Term}\ a\\
&\mathsf{LiftT}\ x \qquad \otimes_T \mathsf{Zero} \qquad = \mathsf{MultT}^1\ (\mathsf{LiftM}\ x)\\
&\mathsf{LiftT}\ x \qquad \otimes_T \mathsf{Unit} \qquad = \mathsf{LiftT}\ x\\
&\mathsf{LiftT}\ x \qquad \otimes_T \mathsf{Term}\ y = \mathsf{MultT}^2\ (\mathsf{LiftM}\ x, y)\\
&\mathsf{SumT}^1\ x \qquad \otimes_T \mathsf{Zero} \qquad = \mathsf{MultT}^1\ (\mathsf{SumM}^1\ x)\\
&\mathsf{SumT}^1\ x \qquad \otimes_T \mathsf{Unit} \qquad = \mathsf{SumT}^1\ x\\
&\mathsf{SumT}^1\ x \qquad \otimes_T \mathsf{Term}\ y = \mathsf{MultT}^2\ (\mathsf{SumM}^1\ x, y)\\
&\mathsf{SumT}^2\ (x, y) \otimes_T \mathsf{Zero} \qquad = \mathsf{MultT}^1\ (\mathsf{SumM}^2\ (x, y))\\
&\mathsf{SumT}^2\ (x, y) \otimes_T \mathsf{Unit} \qquad = \mathsf{SumT}^2\ (x, y)\\
&\mathsf{SumT}^2\ (x, y) \otimes_T \mathsf{Term}\ z = \mathsf{MultT}^2\ (\mathsf{SumM}^2\ (x, y), z)\\
&\mathsf{MultT}^1\ x \qquad \otimes_T y \qquad = \mathsf{MultT}^1\ x\\
&\mathsf{MultT}^2\ (x, y) \otimes_T z \qquad = \mathsf{MultT}^2\ (x, y \otimes_T z)
\end{aligned}
$$

Using those functions, the Dioid instance for Free is the following:

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Dioid}\ (\mathsf{Free}\ a)\ \textbf{where}\\
&\quad \mathsf{z}\ () = \mathsf{Zero}\\
&\quad \mathsf{u}\ () = \mathsf{Unit}\\
&\quad \mathsf{Zero} \quad \oplus x = x\\
&\quad \mathsf{Unit} \quad \oplus v = \mathsf{Unit}\\
&\quad \mathsf{Term}\ x \oplus y = \mathsf{Term}\ (x \oplus_T y)\\
&\quad \mathsf{Zero} \quad \otimes v = \mathsf{Zero}\\
&\quad \mathsf{Unit} \quad \otimes x = x\\
&\quad \mathsf{Term}\ x \otimes y = \mathsf{Term}\ (x \otimes_T y)
\end{aligned}
$$

The insertion morphism and lifting are as follows:

```
ins :: a → Free a
ins a = Term (LiftT a)

univ :: Dioid d ⇒ (a → d) → Free a → d
univ f Zero       = z ()
univ f Unit       = u ()
univ f (Term v) = univ_T f v

univ_T :: Dioid d ⇒ (a → d) → Term a → d
univ_T f (LiftT x)          = f x
univ_T f (SumT¹ s)          = univ_M f s ⊕ u ()
univ_T f (SumT² (s, t))    = univ_M f s ⊕ univ_T f t
univ_T f (MultT¹ s)         = univ_S f s ⊗ z ()
univ_T f (MultT² (s, t))   = univ_S f s ⊗ univ_T f t
```

where univ_M and univ_S are auxiliary functions that work as expected.

The definition of the free dioid and its operations is not complicated but is rather subtle. Therefore we have formally verified that this is indeed the free dioid using Agda as a proof assistant.

We now turn to the problem of generalising the equations (24–27) to other dioid categories. For generalising the free monoid construction, we replaced disjoint union $\uplus$ for coproduct $+$, Cartesian product $\times$ for monoidal tensor $\otimes$, and the singleton set $\{*\}$ for the unit object I of a monoidal category. In the case of dioids, the system of recursive equations presented only involve, in addition to disjoint union, Cartesian product $\times$ and the singleton set $\{*\}$. Nevertheless, a dioid category has more structure: two objects Z and I, and two bifunctors $\oplus$ and $\otimes$. That is the reason we introduced the renamings $\times_\otimes$, $\times_\oplus$, 0, and 1: for keeping track of which constructors correspond to the additive structure and which to the multiplicative structure.

Replacing $\uplus$ for $+$, $\times_\otimes$ for $\otimes$, 1 for I, $\times_\oplus$ for $\oplus$, and 0 for Z in the equations, we obtain the following system of equations for the tentative free dioid over an object X:

$$
\begin{aligned}
X^* &= Z + I + T \\
T &= X + (S \oplus I) + (S \oplus T) + (M \otimes Z) + (M \otimes T) \\
S &= X + (M \otimes Z) + (M \otimes T) \\
M &= X + (S \oplus I) + (S \oplus T)
\end{aligned}
$$

When considering the category of endofunctors with composition as multiplication and Cartesian product as addition, we obtain data type constructors representing these formulas:

```
data Free∘ f x where
  Zero∘  :: K₁ x → Free∘ f x
  Unit∘  :: Identity x → Free∘ f x
  Term∘ :: Term∘ f x → Free∘ f x
data Term∘ f x where
  LiftT∘   :: f x → Term∘ f x
  SumT¹∘  :: Sum∘ f x × Identity x → Term∘ f x
  SumT²∘  :: Sum∘ f x × Term∘ f x → Term∘ f x
  MultT¹∘ :: Mult∘ f (K₁ x) → Term∘ f x
  MultT²∘ :: Mult∘ f (Term∘ f x) → Term∘ f x
data Sum∘ f x where
  LiftS∘   :: f x → Sum∘ f x
  MultS¹∘ :: Mult∘ f (K₁ x) → Sum∘ f x
  MultS²∘ :: Mult∘ f (Term∘ f x) → Sum∘ f x
data Mult∘ f x where
  LiftM∘   :: f x → Mult∘ f x
  SumM¹∘ :: Sum∘ f x × Identity x → Mult∘ f x
  SumM²∘ :: Sum∘ f x × Term∘ f x → Mult∘ f x
```

While at first sight this data type seemed to work, we ran into problems when we tried to define the MonadPlus instance. Specifically, we got stuck when writing the $(\ggg)$ operator on $\mathsf{Free}_\circ\, f$, as we needed to distribute coproducts on the right of $\circ$. Our conjecture is that the recursive equations presented only work for tensors which distribute coproducts on both parameters. Fortunately, Day convolution satisfies such requirement, and we successfully applied the formulas to obtain the free Alternative. The resulting data type constructor is similar to that of monads, but we inlined the data type constructors K_1 and Identity to avoid some clutter:

```
data Free⋆ f x where
  Zero⋆  :: Free⋆ f x
  Unit⋆  :: x → Free⋆ f x
  Term⋆ :: Term⋆ f x → Free⋆ f x
data Term⋆ f x where
  LiftT⋆   :: f x → Term⋆ f x
  SumT¹⋆  :: Sum⋆ f x → x → Term⋆ f x
  SumT²⋆  :: Sum⋆ f x → Term⋆ f x → Term⋆ f x
  MultT¹⋆ :: Mult⋆ f (a → x) → Term⋆ f x
  MultT²⋆ :: Mult⋆ f (a → x) → Term⋆ f a → Term⋆ f x
data Sum⋆ f x where
  LiftS⋆   :: f x → Sum⋆ f x
  MultS¹⋆ :: Mult⋆ f (a → x) → Sum⋆ f x
  MultS²⋆ :: Mult⋆ f (a → x) → Term⋆ f a → Sum⋆ f x
data Mult⋆ f x where
  LiftM⋆   :: f x → Mult⋆ f x
  SumM¹⋆ :: Sum⋆ f x → x → Mult⋆ f x
  SumM²⋆ :: Sum⋆ f x → Term⋆ f x → Mult⋆ f x
```

The instances of Functor, Applicative and Alternative are involved, although they follow the same pattern of the operations for free ordinary dioids. We present the Applicative instance, as its multiplication is the most involved operation.

```
instance Functor f ⇒ Applicative (Free⋆ f) where
  pure x = Unit⋆ x
  Zero⋆      ⊛ v = Zero⋆
  Unit⋆ f    ⊛ x = fmap f x
  Term⋆ x ⊛ y = Term⋆ (x ⊛_T y)
```

The function $(\circledast_T)$ is the equivalent to $(\otimes_T)$ for free ordinary dioids, and it is defined as follows:

```
(⊛_T) :: Functor f ⇒
         Term⋆ f (a → b) → Free⋆ f a → Term⋆ f b
```

$$
\begin{aligned}
\mathsf{LiftT}_\star\ x \quad &\circledast_T\ \mathsf{Zero}_\star &&= \mathsf{MultT}^1_\star\ (\mathsf{LiftM}_\star\ x)\\
\mathsf{LiftT}_\star\ x \quad &\circledast_T\ \mathsf{Unit}_\star\ y &&= \mathsf{LiftT}_\star\ (\mathsf{eval_F}\ x\ y)\\
\mathsf{LiftT}_\star\ x \quad &\circledast_T\ \mathsf{Term}_\star\ y &&= \mathsf{MultT}^2_\star\ (\mathsf{LiftM}_\star\ x)\ y\\
\mathsf{SumT}^1_\star\ x\ y &\circledast_T\ \mathsf{Zero}_\star &&= \mathsf{MultT}^1_\star\ (\mathsf{SumM}^1_\star\ x\ y)\\
\mathsf{SumT}^1_\star\ x\ y &\circledast_T\ \mathsf{Unit}_\star\ v &&= \mathsf{SumT}^1_\star\ (\mathsf{eval_F}\ x\ v)\ (y\ v)\\
\mathsf{SumT}^1_\star\ x\ y &\circledast_T\ \mathsf{Term}_\star\ v &&= \mathsf{MultT}^2_\star\ (\mathsf{SumM}^1_\star\ x\ y)\ v\\
\mathsf{SumT}^2_\star\ x\ v &\circledast_T\ \mathsf{Zero}_\star &&= \mathsf{MultT}^1_\star\ (\mathsf{SumM}^2_\star\ x\ v)\\
\mathsf{SumT}^2_\star\ x\ v &\circledast_T\ \mathsf{Unit}_\star\ y &&= \mathsf{SumT}^2_\star\ (\mathsf{eval_F}\ x\ y)\ (\mathsf{eval_F}\ v\ y)\\
\mathsf{SumT}^2_\star\ x\ v &\circledast_T\ \mathsf{Term}_\star\ y &&= \mathsf{MultT}^2_\star\ (\mathsf{SumM}^2_\star\ x\ v)\ y\\
\mathsf{MultT}^1_\star\ x \quad &\circledast_T\ v &&= \mathsf{MultT}^1_\star\ (\mathsf{fmap\ uncurry}\ x)\\
\mathsf{MultT}^2_\star\ x\ v &\circledast_T\ w &&= \mathsf{MultT}^2_\star\ (\mathsf{fmap\ uncurry}\ x)\\
& && \quad (\mathsf{fmap}\ (,)\ v \circledast_T w)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval_F} :: \mathsf{Functor}\ f \Rightarrow f\ (a \to b) \to a \to f\ b\\
&\mathsf{eval_F}\ v\ a = \mathsf{fmap}\ (\lambda f \to f\ a)\ v
\end{aligned}
$$

The lifting function is also involved, although we insist again that it follows the pattern presented for ordinary dioids. We close our discussion for free structures by presenting the insertion function for $\mathsf{Free}_\star$.

$$
\begin{aligned}
&\mathsf{ins}_\star :: \mathsf{Functor}\ f \Rightarrow f\ a \to \mathsf{Free}_\star\ f\ a\\
&\mathsf{ins}_\star\ v = \mathsf{Term}_\star\ (\mathsf{LiftT}_\star\ v)
\end{aligned}
$$

The correctness for these definitions was also formally verified using Agda.

V. Conclusion

This paper has introduced a generalised notion of dioids, which was used to study computations with shallow-backtracking non-determinism. By considering MonadPlus and Alternative type classes as dioids in the category of endofunctors, we have obtained a set of laws that those type classes should obey.

We have shown a concrete description for the free ordinary dioid, and then generalised the construction to dioid categories. By instantiation, this resulted in the construction of the free Alternative, but we were not able to do the same to obtain a free MonadPlus.

As further work, it would be interesting to study the notion of dioid in the dioid category of endoprofunctors, and see whether the general formulation for free dioids presented can be used in this setting to obtain a free arrow with a notion of shallow-backtracking non-determinism.

Acknowledgment

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

References

[1] E. Moggi, "Computational lambda-calculus and monads," in *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Press, 1989, pp. 14–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=77350.77353

[2] ——, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, Jul. 1991. [Online]. Available: http://dx.doi.org/10.1016/0890-5401(91)90052-4

[3] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90. New York, NY, USA: ACM, 1990, pp. 61–78. [Online]. Available: http://doi.acm.org/10.1145/91556.91592

[4] M. Spivey, "A functional theory of exceptions," *Sci. Comput. Program.*, vol. 14, no. 1, pp. 25–42, May 1990. [Online]. Available: https://doi.org/10.1016/0167-6423(90)90056-J

[5] C. McBride and R. Paterson, "Applicative programming with effects," *Journal of Functional Programming*, vol. 18, no. 01, pp. 1–13, 2008. [Online]. Available: http://dx.doi.org/10.1017/S0956796807006326

[6] S. D. Swierstra and L. Duponcheel, "Deterministic, error-correcting combinator parsers," in *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text*, ser. Lecture Notes in Computer Science, J. Launchbury, E. Meijer, and T. Sheard, Eds., vol. 1129. Springer, 1996, pp. 184–207. [Online]. Available: http://dx.doi.org/10.1007/3-540-61628-4_7

[7] J. Gibbons and B. c. d. s. Oliveira, "The essence of the iterator pattern," *J. Funct. Program.*, vol. 19, no. 3-4, pp. 377–402, Jul. 2009. [Online]. Available: http://dx.doi.org/10.1017/S0956796809007291

[8] M. Jaskelioff and O. Rypacek, "An investigation of the laws of traversals," in Proceedings Fourth Workshop on *Mathematically Structured Functional Programming,* Tallinn, Estonia, 25 March 2012, ser. Electronic Proceedings in Theoretical Computer Science, J. Chapman and P. B. Levy, Eds., vol. 76. Open Publishing Association, 2012, pp. 40–49. [Online]. Available: https://arxiv.org/abs/1202.2919v1

[9] S. Marlow, L. Brandy, J. Coens, and J. Purdy, "There is no fork: An abstraction for efficient, concurrent, and concise data access," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14. New York, NY, USA: ACM, 2014, pp. 325–337. [Online]. Available: http://doi.acm.org/10.1145/2628136.2628144

[10] E. Rivas and M. Jaskelioff, "Notions of computation as monoids," *CoRR*, vol. abs/1406.4823, 2014. [Online]. Available: http://arxiv.org/abs/1406.4823

[11] S. Mac Lane, *Categories for the Working Mathematician*, ser. Graduate Texts in Mathematics. Springer-Verlag, 1971, no. 5, second edition, 1998.

[12] J. Hughes, "A novel representation of lists and its application to the function "reverse"," *Information Processing Letters*, vol. 22, no. 3, pp. 141–144, 1986. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(86)90059-1

[13] J. Voigtländer, "Asymptotic improvement of computations over free monads," in *Proceedings of the 9th International Conference on Mathematics of Program Construction*, ser. MPC '08. Springer-Verlag, 2008, pp. 388–403. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70594-9_20

[14] T. Uustalu, "A divertimento on monadplus and nondeterminism," *J. Log. Algebr. Meth. Program.*, vol. 85, no. 5, pp. 1086–1094, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.jlamp.2016.06.004

[15] J. M. Spivey, "When maybe is not good enough." *J. Funct. Program.*, vol. 22, no. 6, pp. 747–756, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/jfp/jfp22.html#Spivey12

[16] E. Rivas, M. Jaskelioff, and T. Schrijvers, "From monoids to nearsemirings: The essence of monadplus and alternative," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP'15, M. Falaschi and E. Albert, Eds. ACM, 2015, pp. 196–207. [Online]. Available: http://doi.acm.org/10.1145/2790449.2790514

[17] M. Grandis, "Cubical monads and their symmetries," *Rend. Instit. Mat. Univ. Tieste*, vol. 25, pp. 223–264, 1993. [Online]. Available: http://hdl.handle.net/10077/4693

[18] E. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott, "Functorial polymorphism," *Theoretical Computer Science*, vol. 70, no. 1, pp. 35 – 64, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397590901517

[19] B. Day, "Note on monoidal localisation," *Bulletin of the Australian Mathematical Society*, vol. 8, pp. 1–16, 2 1973. [Online]. Available: http://journals.cambridge.org/article_S0004972700045433

[20] R. M. Burstall and J. A. Goguen, *Algebras, Theories and Freeness: An Introduction for Computer Scientists*. Dordrecht: Springer Netherlands, 1982, pp. 329–349. [Online]. Available: http://dx.doi.org/10.1007/978-94-009-7893-5_11

[21] G. M. Kelly, "A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on," *Bulletin of the Australian Mathematical Society*, vol. 22, no. 01, pp. 1–83, 1980. [Online]. Available: https://doi.org/10.1017/S0004972700006353