

Dioids for Computational Effects

Ezequiel Postan

*DCC-ECEN
Facultad de Cs. Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Rosario, Argentina*

Exequiel Rivas and Mauro Jaskelioff

*DCC-ECEN
Facultad de Cs. Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
and
CIFASIS-CONICET
Rosario, Argentina*

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.

Keywords: dioid, monad, Haskell, computational effect

1 Introduction

Algebraic structures have been central to the modelling of computational effects. For example, monads [12,13,21,17] have been used to model many computational effects such as global state, exceptions, environments, input/output, and continuations. More recently, applicative functors [11] are becoming popular in diverse applications such as modelling parsers [18], characterising traversals [4,7], and in combination with monads to obtain concurrent queries [10].

While monads and applicative functors are two different algebraic structures, they have a unified framework. Rivas and Jaskelioff [14] have shown that both of them can be seen as instances of a same unifying concept: monoids in a monoidal

category [9]. 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 [6] is shown to be essentially the same as a newer one for monads [20], and led to the discovery of a new one for applicative functors [14] 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 [19], but in functional programming the most common are deep backtracking and shallow backtracking [16]. 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 [15]. If, on the other hand, one wants to model shallow backtracking, then one arrives at the algebraic structure of dioids [5].

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 2 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 3, 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 4, 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 5, we conclude.

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

2 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.

2.1 Monoids in Programming

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

class Monoid m where

$u :: () \rightarrow m$
 $(\otimes) :: m \times m \rightarrow m$

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

$$a \otimes (u ()) = a \quad (1)$$

$$(u ()) \otimes a = a \quad (2)$$

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

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

$u () = 0$
 $x \otimes y = x + y$

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

$\text{Endo} :: (a \rightarrow a) \rightarrow \text{Endo } a$

instance Monoid (Endo a) where

$u () = \text{Endo id}$
 $(\text{Endo } f) \otimes (\text{Endo } g) = \text{Endo } (f \circ g)$

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 $f :: M_1 \rightarrow M_2$ such that the monoid instances are preserved:

$$f (u ()) = u ()$$

$$f (a \otimes b) = f a \otimes f b$$

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 the **Monoid** instance **Integer** to the **Monoid** instance **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.

2.2 Categorification

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

Definition 2.1 A *monoidal category* is a sextuple $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$, where:

- \mathcal{C} is a category;
- $\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ 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:

$$\begin{aligned}\alpha &: A \otimes (B \otimes C) \cong (A \otimes B) \otimes C \\ \lambda &: I \otimes A \cong A \\ \rho &: A \otimes I \cong A\end{aligned}$$

All these natural transformations should obey coherence laws [9].

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

Definition 2.2 A *monoid in a monoidal category* is an object M , together with operations $e : I \rightarrow M$ and $m : M \otimes M \rightarrow M$, for which the following laws hold:

$$m \circ (e \otimes \text{id}) = \lambda \tag{4}$$

$$m \circ (\text{id} \otimes e) = \rho \tag{5}$$

$$m \circ (\text{id} \otimes m) = m \circ (m \otimes \text{id}) \circ \alpha \tag{6}$$

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.

2.3 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

- (i) given a type, say `Integer`, they construct a type of list of `Integers`;
- (ii) 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 ∘ g) = fmap f ∘ 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 \Rightarrow 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 [1].

```
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)
```

2.4 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** $\rightarrow m$, and $m \circ m \rightarrow m$ 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 $m \Rightarrow$ Triple m **where**

$\eta :: a \rightarrow m\ a$

$\mu :: m\ (m\ a) \rightarrow m\ a$

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

$$\begin{aligned}\mu \circ \eta &= \text{id} \\ \mu \circ \text{fmap } \eta &= \text{id} \\ \mu \circ \text{fmap } \mu &= \mu \circ \mu\end{aligned}$$

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 $:: a \rightarrow m\ a$

$(\text{>}) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

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 $\eta = \text{return}$, that μ can be defined as $\mu\ x = (x\ \text{> id})$, and that (>) can be defined as $(x\ \text{> } k) = \mu\ (\text{fmap } k\ x)$. 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 $\text{fmap } f\ v = (v\ \text{> return } \circ f)$.

Remark 2.3 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:

$\text{curry} :: (a \times b \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$

$\text{curry } f\ a\ b = f\ (a, b)$

$\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a \times b \rightarrow c)$

$\text{uncurry } f\ (a, b) = f\ a\ b$

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 [11] as a way to capture certain effectful computations that do not fit well in the monadic framework.

2.5 Applicative functors

Applicative functors are based on a category of endofunctors, but with different tensor than monads: the *Day convolution* [3]. The Day convolution may be implemented as follows:

data $(\star) f g a$ **where**

```
Day :: f (b → a) × (g b) → (f ∗ g) a
```

instance (Functor f , Functor g) \Rightarrow Functor $(f \star g)$ **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).

```
λ :: Functor f ⇒ (Identity ∗ f) a → f a
λ (Day (Identity f) x) = fmap f x

ρ :: Functor f ⇒ (f ∗ Identity) a → f a
ρ (Day f (Identity b)) = fmap (λh → h b) f

α :: (Functor f, Functor g) ⇒
  (f ∗ (g ∗ h)) a → ((f ∗ g) ∗ h) a
α (Day f (Day g z)) = Day (Day (fmap (∘) f) g) z
```

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

class Functor $f \Rightarrow$ Applicative f **where**

$\text{pure} :: a \rightarrow f\ a$

$(\otimes) :: f\ (b \rightarrow a) \times f\ b \rightarrow f\ a$

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

$$\begin{aligned}\text{pure } f \otimes u &= \text{fmap } f\ u \\ u \otimes \text{pure } x &= \text{fmap } (\lambda h \rightarrow h\ x)\ u \\ (\text{fmap } (\circ)\ u \otimes v) \otimes w &= u \otimes (v \otimes w)\end{aligned}$$

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 2.4 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 Monad $m \Rightarrow$ Applicative m **where**

$\text{pure } x = \text{return } x$

$u \otimes v = u \succcurlyeq \lambda f \rightarrow v \succcurlyeq \lambda x \rightarrow \text{return } (f\ x)$

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 $K\ x\ a$ **where**

$\text{MK} :: x \rightarrow K\ x\ a$

instance Functor $(K\ x)$ **where**

$\text{fmap } f\ (\text{MK } x) = \text{MK } x$

instance Monoid $x \Rightarrow$ Applicative $(K\ x)$ **where**

$\text{pure } a = \text{MK } u$

$(\text{MK } x) \otimes (\text{MK } y) = \text{MK } (x \otimes y)$

3 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 \mathbf{z} . Whereas the multiplicative monoid gives a model of sequencing, the \oplus operation models a non-deterministic choice, and models \mathbf{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.

3.1 Set dioids

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

class Dioid d **where**

$z \quad :: () \rightarrow d$

$u \quad :: () \rightarrow d$

$(\oplus) :: d \times d \rightarrow d$

$(\otimes) :: d \times d \rightarrow d$

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

$$a \otimes (u ()) = a \quad (7)$$

$$(u ()) \otimes a = a \quad (8)$$

$$a \otimes (b \otimes c) = (a \otimes b) \otimes c \quad (9)$$

$$a \oplus (z ()) = a \quad (10)$$

$$(z ()) \oplus a = a \quad (11)$$

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c \quad (12)$$

$$(z ()) \otimes a = z () \quad (13)$$

$$(u ()) \oplus a = u () \quad (14)$$

The laws 7 to 12 express that d is a monoid with respect to (\otimes, u) and (\oplus, 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 BinFun $a = a \times a \rightarrow a$

instance Dioid (BinFun a) **where**

$z \quad = \lambda(a, b) \rightarrow a$

$u \quad = \lambda(a, b) \rightarrow b$

$f \oplus g = \lambda(a, b) \rightarrow f (g (a, b), b)$

$f \otimes g = \lambda(a, b) \rightarrow f (a, g (a, b))$

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 [15] 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 *diod homomorphism* is a function $f :: D_1 \rightarrow D_2$ such that the dioid instances are preserved:

$$\begin{aligned} f(u()) &= u() \\ f(z()) &= z() \\ f(a \otimes b) &= f a \otimes f b \\ f(a \oplus b) &= f a \oplus f b \end{aligned}$$

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 .

3.2 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 3.1 A *diod 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;
- $\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ and $\oplus : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ are bifunctors;
- I and Z are objects of \mathcal{C} ;
- α_\otimes , λ_\otimes , ρ_\otimes , α_\oplus , λ_\oplus , and ρ_\oplus are natural isomorphisms:

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

- κ_\otimes and κ_\oplus are natural transformations:

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

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 3.2 A *diod in a dioid category* is an object D , together with operations $z : Z \rightarrow D$, $e : I \rightarrow D$, $s : D \oplus D \rightarrow D$ and $m : D \otimes D \rightarrow D$ for which the following

laws hold:

$$m \circ (e \otimes \text{id}) = \lambda_{\otimes} \quad (15)$$

$$m \circ (\text{id} \otimes e) = \rho_{\otimes} \quad (16)$$

$$m \circ (\text{id} \otimes m) = m \circ (m \otimes \text{id}) \circ \alpha_{\otimes} \quad (17)$$

$$s \circ (z \oplus \text{id}) = \lambda_{\oplus} \quad (18)$$

$$s \circ (\text{id} \oplus z) = \rho_{\oplus} \quad (19)$$

$$s \circ (\text{id} \oplus s) = s \circ (s \oplus \text{id}) \circ \alpha_{\oplus} \quad (20)$$

$$m \circ (z \otimes \text{id}) = z \circ \kappa_{\otimes} \quad (21)$$

$$s \circ (e \oplus \text{id}) = e \circ \kappa_{\oplus} \quad (22)$$

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 $\kappa_{\otimes} = \kappa_{\oplus} = \pi_1 : \{*\} \times A \rightarrow \{*\}$ is not a natural isomorphism, but only a natural transformation.

Lemma 3.3 *In general, from a monoidal category $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$ with Cartesian structure (that is, with 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:*

$$\begin{aligned} \alpha_{\times} &= \langle \langle \pi_1, \pi_1 \circ \pi_2 \rangle, \pi_2 \circ \pi_2 \rangle : A \times (B \times C) \rightarrow (A \times B) \times C \\ \kappa_{\otimes} &= !_1 \otimes A : 1 \otimes A \rightarrow 1 \end{aligned}$$

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 3.4 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 $f : D_1 \rightarrow D_2$ such that the following equations hold:

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

3.3 Cartesian structure for functors

We can use Lemma 3.3 to extend the monoidal categories of endofunctors discussed in Sections 2.4 and 2.5 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 K_1 *a where*

$$K_1 :: () \rightarrow K_1$$

instance Functor K_1 **where**

$\text{fmap } f (K_1 ()) = K_1 ()$

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 $(f \times g) a$ **where**

$\text{Pair} :: f a \times g a \rightarrow (f \times g) a$

instance (Functor f , Functor g) \Rightarrow Functor $(f \times g)$ **where**

$\text{fmap } h (\text{Pair } (fa, ga)) = \text{Pair } (\text{fmap } h fa, \text{fmap } h ga)$

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.

3.4 Non-determinism Monads

By Lemma 3.3 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 `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 $K_1 \rightarrow d$, $\text{Identity} \rightarrow d$, $d \times d \rightarrow d$ and $d \circ d \rightarrow d$ respectively. If we expand the definitions, we can present this information in a type class:

class DioidM d **where**

$\zeta :: () \rightarrow d a$

$\eta :: a \rightarrow d a$

$\sigma :: d a \times d a \rightarrow d a$

$\mu :: d (d a) \rightarrow d a$

subject to the laws:

$$\begin{aligned} \mu \circ \eta &= \text{id} \\ \mu \circ \text{fmap } \eta &= \text{id} \\ \mu \circ \text{fmap } \mu &= \mu \circ \mu \\ \sigma \circ \text{pair id zero} &= \text{fst} \\ \sigma \circ \text{pair zero id} &= \text{snd} \\ \sigma \circ \text{pair id } \sigma &= \sigma \circ \text{pair } \sigma \text{ id } \circ \alpha \\ \mu \circ \zeta &= \zeta \\ \sigma \circ \text{pair } \eta \text{ id} &= \eta \circ \text{fst} \end{aligned}$$

where $\text{pair } f g (x, y) = (f x, g y)$ 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 $m \Rightarrow$ MonadPlus m **where**

mzero :: $m\ a$

mplus :: $m\ a \rightarrow m\ a \rightarrow m\ a$

subject to the following laws

mplus **mzero** $u = u$

mplus u **mzero** $= u$

mplus u (**mplus** $v\ w$) $=$ **mplus** (**mplus** $u\ v$) w

mzero $\succcurlyeq f =$ **mzero**

mplus (**return** x) $u =$ **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 **mplus** (**return** x) $u =$ **return** x . (In fact, lists are the canonical example of deep-backtracking non-determinism.)

3.5 Non-determinism Applicative Functors

We turn again to Lemma 3.3 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 (\otimes) of type $K_1 \dot{\rightarrow} d$, $\text{Identity} \dot{\rightarrow} d$, $d \times d \dot{\rightarrow} d$ and $d \star d \dot{\rightarrow} d$ respectively. If we expand the definitions, we can present this information in a type class:

class DioidF d **where**

empty :: $() \rightarrow d\ a$

pure :: $a \rightarrow d\ a$

$(\langle \rangle)$:: $d\ a \times d\ a \rightarrow d\ a$

(\otimes) :: $d\ (b \rightarrow a) \times d\ b \rightarrow d\ a$

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 $f \Rightarrow$ Alternative f **where**

empty :: $f\ a$

($\langle| \rangle$) :: $f\ a \rightarrow f\ a \rightarrow f\ a$

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

empty $\langle| \rangle\ u = u$

$u\ \langle| \rangle\ \text{empty} = u$

$u\ \langle| \rangle\ (v\ \langle| \rangle\ w) = (u\ \langle| \rangle\ v)\ \langle| \rangle\ w$

empty $\oplus u = \text{empty}$

pure $x\ \langle| \rangle\ u = \text{pure } x$

We can extend Remark 2.4 to the type classes **MonadPlus** and **Alternative**, and obtain the following result.

instance MonadPlus $m \Rightarrow$ Alternative m **where**

empty = mzero

$u\ \langle| \rangle\ v = \text{mplus } u\ v$

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 $d \Rightarrow$ Alternative $(K\ d)$ **where**

empty = MK z

$(\text{MK } d_1)\ \langle| \rangle\ (\text{MK } d_2) = \text{MK } (d_1 \oplus d_2)$

4 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 [2]. In our setting, we employ free structures as a device to work with those programs that only use the structure under analysis.

4.1 Free monoids

The notion of free ordinary monoid is captured by a universal property. Formally, we say that the type **FreeMon_a** is the free monoid over a when:

- **FreeMon_a** is an instance of **Monoid**;
- there is a function $\text{ins} :: a \rightarrow \text{FreeMon}_a$;
- for any **Monoid** instance m and function $f :: a \rightarrow m$, there exists a unique monoid homomorphism $\text{univ } f :: \text{FreeMon}_a \rightarrow m$ such that $\text{univ } f \circ \text{ins} = f$.

While mathematically precise, this definition is not constructive: it does not provide a procedure to construct such 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, $a \otimes (u () \otimes (b \otimes c))$ and $(a \otimes b) \otimes (c \otimes u ())$ 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

$\text{Nil} :: \text{FreeMon } a$

$\text{Cons} :: a \times \text{FreeMon } a \rightarrow \text{FreeMon } a$

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

instance Monoid (FreeMon a) where

$u () = \text{Nil}$

$\text{Nil} \otimes bs = bs$

$(\text{Cons } a \ as) \otimes bs = \text{Cons } a \ (as \otimes bs)$

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

$\text{ins} :: a \rightarrow \text{FreeMon } a$

$\text{ins } a = \text{Cons } a \ \text{Nil}$

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

$\text{univ} :: \text{Monoid } m \Rightarrow (a \rightarrow m) \rightarrow \text{FreeMon } a \rightarrow m$

$\text{univ } f \ \text{Nil} = u ()$

$\text{univ } f \ (\text{Cons } a \ as) = f \ a \otimes \text{univ } f \ as$

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:

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

where \cup represents the disjoint union of sets.

Generalising to monoidal categories the equation becomes

$$A^* \cong I + A \otimes A^* \quad (23)$$

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

$$\begin{array}{ccc} X & \xrightarrow{\text{ins}} & F \\ & \searrow f & \downarrow \text{univ}(f) \\ & & M \end{array}$$

The morphism ins is called the *insertion* morphism, and $\text{univ } f$ 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

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

that leads to the following data type:

data Free f a **where**

Nil_o :: $a \rightarrow \text{Free } f \ a$

Cons_o :: $f \ (\text{Free } f \ a) \rightarrow \text{Free } f \ a$

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

instance Functor $f \Rightarrow \text{Monad } (\text{Free } f)$ **where**

return x = Nil_o x

Nil_o $x \succcurlyeq f = f \ x$

Cons_o $v \succcurlyeq f = \text{Cons}_o \ (\text{fmap } (\succcurlyeq f) \ v)$

The insertion morphism and lifting are as follows:

ins :: Functor $f \Rightarrow f \rightarrow \text{Free } f$

ins $v = \text{Cons}_o \ (\text{fmap Nil}_o \ v)$

univ :: (Functor f , Monad m) $\Rightarrow (f \rightarrow m) \rightarrow (\text{Free } f \rightarrow m)$

$\text{univ } f \text{ (Nil}_o x) = \text{return } x$
 $\text{univ } f \text{ (Cons}_o v) = f \text{ (fmap (univ } f) v) \succ \text{id}$

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:

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

which leads to the following data type

data FreeA f a **where**
 $\text{Nil}_\star :: a \rightarrow \text{FreeA } f \text{ } a$
 $\text{Cons}_\star :: f \text{ (} b \rightarrow a \text{)} \times \text{FreeA } f \text{ } b \rightarrow \text{FreeA } f \text{ } a$

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

instance Functor $f \Rightarrow \text{Applicative (FreeA } f) \text{ where}$
 $\text{pure } x = \text{Nil}_\star x$
 $\text{Nil}_\star h \otimes x = \text{fmap } h \text{ } x$
 $\text{Cons}_\star (h, x) \otimes y = \text{Cons}_\star (\text{fmap uncurry } h) (\text{fmap } (,) \text{ } x \otimes y)$

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

$\text{ins} :: \text{Functor } f \Rightarrow f \dot{\rightarrow} \text{FreeA } f$
 $\text{ins } v = \text{Cons}_\star (\text{fmap const } v) (\text{Nil}_\star ())$
 $\text{univ} :: (\text{Functor } f, \text{Applicative } g) \Rightarrow (f \dot{\rightarrow} g) \rightarrow (\text{FreeA } f \dot{\rightarrow} g)$
 $\text{univ } f \text{ (Nil}_\star x) = \text{pure } x$
 $\text{univ } f \text{ (Cons}_\star v \text{ } r) = f \text{ } v \otimes \text{univ } f \text{ } r$

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 [8]. The case of free monads and free applicative functors is analysed in detail by Rivas and Jaskelioff [14].

4.2 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 free ordinary dioids and then generalising them, 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 4.2 Let X be an object of a dioid category, the *free dioid* over X is a dioid (F, z_F, e_F, s_F, m_F) together with a morphism $\text{ins} : X \rightarrow F$ such that for any dioid (D, z_D, e_D, s_D, m_D) and morphism $f : X \rightarrow D$ there exists a unique dioid homomorphism $\text{univ}(f) : F \rightarrow D$ such that $\text{univ}(f) \circ \text{ins} = f$.

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:

$$X^* = 0 \sqcup 1 \sqcup T \quad (24)$$

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

$$S = X \sqcup (M \times_{\otimes} 0) \sqcup (M \times_{\otimes} T) \quad (26)$$

$$M = X \sqcup (S \times_{\oplus} 1) \sqcup (S \times_{\oplus} T) \quad (27)$$

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:

data Free a where

Zero :: Free a

Unit :: Free a

Term :: Term $a \rightarrow$ Free a

data Term a where

LiftT :: $a \rightarrow$ Term a

SumT¹ :: Sum $a \rightarrow$ Term a

SumT² :: Sum $a \times$ Term $a \rightarrow$ Term a

MultT¹ :: Mult $a \rightarrow$ Term a

MultT² :: Mult $a \times$ Term $a \rightarrow$ Term a

data Sum a where

LiftS :: $a \rightarrow$ Sum a

MultS¹ :: Mult $a \rightarrow$ Sum a

MultS² :: Mult $a \times$ Term $a \rightarrow$ Sum a

data Mult a where

LiftM :: $a \rightarrow$ Mult a

SumM¹ :: Sum $a \rightarrow$ Mult a

SumM² :: Sum $a \times$ Term $a \rightarrow$ Mult a

The dioid operations for 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.

```

( $\otimes_T$ ) :: Term a  $\rightarrow$  Free a  $\rightarrow$  Term a
LiftT x       $\otimes_T$  Zero  = MultT1 (LiftM x)
LiftT x       $\otimes_T$  Unit  = LiftT x
LiftT x       $\otimes_T$  Term y = MultT2 (LiftM x, y)
SumT1 x      $\otimes_T$  Zero  = MultT1 (SumM1 x)
SumT1 x      $\otimes_T$  Unit  = SumT1 x
SumT1 x      $\otimes_T$  Term y = MultT2 (SumM1 x, y)
SumT2 (x, y)  $\otimes_T$  Zero  = MultT1 (SumM2 (x, y))
SumT2 (x, y)  $\otimes_T$  Unit  = SumT2 (x, y)
SumT2 (x, y)  $\otimes_T$  Term z = MultT2 (SumM2 (x, y), z)
MultT1 x      $\otimes_T$  y      = MultT1 x
MultT2 (x, y)  $\otimes_T$  z      = MultT2 (x, y  $\otimes_T$  z)

```

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

```

instance Dioid (Free a) where
  z () = Zero
  u () = Unit
  Zero  $\oplus$  x = x
  Unit  $\oplus$  v = Unit
  Term x  $\oplus$  y = Term (x  $\oplus_T$  y)
  Zero  $\otimes$  v = Zero
  Unit  $\otimes$  x = x
  Term x  $\otimes$  y = Term (x  $\otimes_T$  y)

```

The insertion morphism and lifting are as follows:

```

ins :: a  $\rightarrow$  Free a
ins a = Term (LiftT a)
univ :: Dioid d  $\Rightarrow$  (a  $\rightarrow$  d)  $\rightarrow$  Free a  $\rightarrow$  d
univ f Zero      = z ()
univ f Unit      = u ()
univ f (Term v)  = univT f v
univT :: Dioid d  $\Rightarrow$  (a  $\rightarrow$  d)  $\rightarrow$  Term a  $\rightarrow$  d
univT f (LiftT x)      = f x
univT f (SumT1 s)      = univM f s  $\oplus$  u ()
univT f (SumT2 (s, t)) = univM f s  $\oplus$  univT f t
univT f (MultT1 s)      = univS f s  $\otimes$  z ()
univT f (MultT2 (s, t)) = univS f s  $\otimes$  univT 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 \sqcup 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 \sqcup 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_o f x **where**

Zero_o :: $K_1\ x \rightarrow \text{Free}_o\ f\ x$
 Unit_o :: $\text{Identity}\ x \rightarrow \text{Free}_o\ f\ x$
 Term_o :: $\text{Term}_o\ f\ x \rightarrow \text{Free}_o\ f\ x$

data Term_o f x **where**

LiftT_o :: $f\ x \rightarrow \text{Term}_o\ f\ x$
 SumT_o¹ :: $\text{Sum}_o\ f\ x \times \text{Identity}\ x \rightarrow \text{Term}_o\ f\ x$
 SumT_o² :: $\text{Sum}_o\ f\ x \times \text{Term}_o\ f\ x \rightarrow \text{Term}_o\ f\ x$
 MultT_o¹ :: $\text{Mult}_o\ f\ (K_1\ x) \rightarrow \text{Term}_o\ f\ x$
 MultT_o² :: $\text{Mult}_o\ f\ (\text{Term}_o\ f\ x) \rightarrow \text{Term}_o\ f\ x$

data Sum_o f x **where**

LiftS_o :: $f\ x \rightarrow \text{Sum}_o\ f\ x$
 MultS_o¹ :: $\text{Mult}_o\ f\ (K_1\ x) \rightarrow \text{Sum}_o\ f\ x$
 MultS_o² :: $\text{Mult}_o\ f\ (\text{Term}_o\ f\ x) \rightarrow \text{Sum}_o\ f\ x$

data Mult_o f x **where**

LiftM_o :: $f\ x \rightarrow \text{Mult}_o\ f\ x$
 SumM_o¹ :: $\text{Sum}_o\ f\ x \times \text{Identity}\ x \rightarrow \text{Mult}_o\ f\ x$

$$\text{SumM}_\circ^2 :: \text{Sum}_\circ f x \times \text{Term}_\circ f x \rightarrow \text{Mult}_\circ f x$$

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 (\bowtie) operator on $\text{Free}_\circ f$, 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 $\text{Free}_\star f x$ **where**

$\text{Zero}_\star :: \text{Free}_\star f x$

$\text{Unit}_\star :: x \rightarrow \text{Free}_\star f x$

$\text{Term}_\star :: \text{Term}_\star f x \rightarrow \text{Free}_\star f x$

data $\text{Term}_\star f x$ **where**

$\text{LiftT}_\star :: f x \rightarrow \text{Term}_\star f x$

$\text{SumT}_\star^1 :: \text{Sum}_\star f x \rightarrow x \rightarrow \text{Term}_\star f x$

$\text{SumT}_\star^2 :: \text{Sum}_\star f x \rightarrow \text{Term}_\star f x \rightarrow \text{Term}_\star f x$

$\text{MultT}_\star^1 :: \text{Mult}_\star f (a \rightarrow x) \rightarrow \text{Term}_\star f x$

$\text{MultT}_\star^2 :: \text{Mult}_\star f (a \rightarrow x) \rightarrow \text{Term}_\star f a \rightarrow \text{Term}_\star f x$

data $\text{Sum}_\star f x$ **where**

$\text{LiftS}_\star :: f x \rightarrow \text{Sum}_\star f x$

$\text{MultS}_\star^1 :: \text{Mult}_\star f (a \rightarrow x) \rightarrow \text{Sum}_\star f x$

$\text{MultS}_\star^2 :: \text{Mult}_\star f (a \rightarrow x) \rightarrow \text{Term}_\star f a \rightarrow \text{Sum}_\star f x$

data $\text{Mult}_\star f x$ **where**

$\text{LiftM}_\star :: f x \rightarrow \text{Mult}_\star f x$

$\text{SumM}_\star^1 :: \text{Sum}_\star f x \rightarrow x \rightarrow \text{Mult}_\star f x$

$\text{SumM}_\star^2 :: \text{Sum}_\star f x \rightarrow \text{Term}_\star f x \rightarrow \text{Mult}_\star f x$

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` $f \Rightarrow \text{Applicative } (\text{Free}_\star f)$ **where**

`pure` $x = \text{Unit}_\star x$

$\text{Zero}_\star \otimes v = \text{Zero}_\star$

$\text{Unit}_\star f \otimes x = \text{fmap } f x$

$\text{Term}_\star x \otimes y = \text{Term}_\star (x \otimes_T y)$

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

$(\otimes_T) :: \text{Functor } f \Rightarrow$

$\text{Term}_\star f (a \rightarrow b) \rightarrow \text{Free}_\star f a \rightarrow \text{Term}_\star f b$

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

$\text{eval}_F :: \text{Functor } f \Rightarrow f (a \rightarrow b) \rightarrow a \rightarrow f b$

$\text{eval}_F v a = \text{fmap } (\lambda f \rightarrow f a) v$

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 Free_\star .

$\text{ins}_\star :: \text{Functor } f \Rightarrow f a \rightarrow \text{Free}_\star f a$

$\text{ins}_\star v = \text{Term}_\star (\text{LiftT}_\star v)$

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

5 Conclusion

This paper has introduced a generalised notion of dioids, which was used to study computations with shallow-backtracking non-determinism. By considering the instances of MonadPlus and Alternative type classes as dioids, we have obtained a set of laws that those instances 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] Bainbridge, E., P. J. Freyd, A. Scedrov and P. J. Scott, *Functorial polymorphism*, Theoretical Computer Science **70** (1990), pp. 35 – 64.
URL <http://www.sciencedirect.com/science/article/pii/0304397590901517>
- [2] Burstall, R. M. and J. A. Goguen, “Algebras, Theories and Freeness: An Introduction for Computer Scientists,” Springer Netherlands, Dordrecht, 1982 pp. 329–349.
URL http://dx.doi.org/10.1007/978-94-009-7893-5_11
- [3] Day, B., *Note on monoidal localisation*, Bulletin of the Australian Mathematical Society **8** (1973), pp. 1–16.
URL http://journals.cambridge.org/article_S0004972700045433
- [4] Gibbons, J. and B. c. d. s. Oliveira, *The essence of the iterator pattern*, J. Funct. Program. **19** (2009), pp. 377–402.
URL <http://dx.doi.org/10.1017/S0956796809007291>
- [5] Grandis, M., *Cubical monads and their symmetries*, Rend. Instit. Mat. Univ. Trieste **25** (1993), pp. 223–264.
URL <http://hdl.handle.net/10077/4693>
- [6] Hughes, J., *A novel representation of lists and its application to the function “reverse”*, Information Processing Letters **22** (1986), pp. 141–144.
URL [http://dx.doi.org/10.1016/0020-0190\(86\)90059-1](http://dx.doi.org/10.1016/0020-0190(86)90059-1)
- [7] Jaskelioff, M. and O. Rypacek, *An investigation of the laws of traversals*, in: J. Chapman and P. B. Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, Tallinn, Estonia, 25 March 2012, Electronic Proceedings in Theoretical Computer Science **76** (2012), pp. 40–49.
URL <https://arxiv.org/abs/1202.2919v1>
- [8] Kelly, G. M., *A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on*, Bulletin of the Australian Mathematical Society **22** (1980), pp. 1–83.
URL <https://doi.org/10.1017/S0004972700006353>
- [9] Mac Lane, S., “Categories for the Working Mathematician,” Number 5 in Graduate Texts in Mathematics, Springer-Verlag, 1971, second edition, 1998.
- [10] Marlow, S., 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*, ICFP ’14 (2014), pp. 325–337.
URL <http://doi.acm.org/10.1145/2628136.2628144>
- [11] McBride, C. and R. Paterson, *Applicative programming with effects*, Journal of Functional Programming **18** (2008), pp. 1–13.
URL <http://dx.doi.org/10.1017/S0956796807006326>
- [12] Moggi, E., *Computational lambda-calculus and monads*, in: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (1989), pp. 14–23.
URL <http://dl.acm.org/citation.cfm?id=77350.77353>
- [13] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.
URL [http://dx.doi.org/10.1016/0890-5401\(91\)90052-4](http://dx.doi.org/10.1016/0890-5401(91)90052-4)
- [14] Rivas, E. and M. Jaskelioff, *Notions of computation as monoids*, CoRR **abs/1406.4823** (2014).
URL <http://arxiv.org/abs/1406.4823>
- [15] Rivas, E., M. Jaskelioff and T. Schrijvers, *From monoids to nearsemirings: The essence of monadplus and alternative*, in: M. Falaschi and E. Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP’15 (2015), pp. 196–207.
URL <http://doi.acm.org/10.1145/2790449.2790514>
- [16] Spivey, J. M., *When maybe is not good enough.*, J. Funct. Program. **22** (2012), pp. 747–756.
URL <http://dblp.uni-trier.de/db/journals/jfp/jfp22.html#Spivey12>
- [17] Spivey, M., *A functional theory of exceptions*, Sci. Comput. Program. **14** (1990), pp. 25–42.
URL [https://doi.org/10.1016/0167-6423\(90\)90056-J](https://doi.org/10.1016/0167-6423(90)90056-J)
- [18] Swierstra, S. D. and L. Duponcheel, *Deterministic, error-correcting combinator parsers*, in: J. Launchbury, E. Meijer and T. Sheard, editors, *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text*, Lecture Notes in Computer Science **1129** (1996), pp. 184–207.
URL http://dx.doi.org/10.1007/3-540-61628-4_7

- [19] Uustalu, T., *A divertimento on monadplus and nondeterminism*, J. Log. Algebr. Meth. Program. **85** (2016), pp. 1086–1094.
URL <http://dx.doi.org/10.1016/j.jlamp.2016.06.004>
- [20] Voigtländer, J., *Asymptotic improvement of computations over free monads*, in: *Proceedings of the 9th International Conference on Mathematics of Program Construction*, MPC '08 (2008), pp. 388–403.
URL http://dx.doi.org/10.1007/978-3-540-70594-9_20
- [21] Wadler, P., *Comprehending monads*, in: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90 (1990), pp. 61–78.
URL <http://doi.acm.org/10.1145/91556.91592>