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. 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:

\[
\text{class Monoid } m \ \text{where} \\
\quad u :: () \rightarrow m \\
\quad (\otimes) :: m \times m \rightarrow m
\]
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

\[
\begin{align*}
\tag{1} a \otimes (u()) &= a \\
\tag{2} (u()) \otimes a &= a \\
\tag{3} a \otimes (b \otimes c) &= (a \otimes b) \otimes c
\end{align*}
\]

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 \( \text{Integer} \) is a monoid of addition and zero:

```haskell
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.

```haskell
data Endo a where
  Endo :: (a \rightarrow a) \rightarrow Endo a

instance Monoid (Endo a) where
  u() = Endo id
  (Endo f) \otimes (Endo g) = 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 \( \text{Monoid} \). A monoid homomorphism is a function \( f :: M_1 \rightarrow M_2 \) such that the monoid instances are preserved:

\[
\begin{align*}
f(u()) &= u() \\
f(a \otimes b) &= f a \circ f b
\end{align*}
\]

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

For example, the following is a monoid homomorphism from \( \text{Integer} \) to \( \text{Endo Integer} \).

```haskell
rep :: Integer \rightarrow Endo Integer
rep x = Endo (\lambda y \rightarrow x + y)
```

We want to program with algebraic structures such as \( \text{Monoid} \) above, but we may also want to generalise from types to type 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;
- \( \otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C} \) is a bifunctor;
- \( I \) is an object of \( \mathcal{C} \);
  - The bifunctor \( \otimes \) and \( I \) generalise \( \times \) and \( () \) in the code above.
- \( \alpha, \lambda, \) and \( \rho \) are natural isomorphisms:
  \[
  \begin{align*}
  \alpha &: A \otimes (B \otimes C) \cong (A \otimes B) \otimes C \\
  \lambda &: I \otimes A \cong A \\
  \rho &: A \otimes I \cong A
  \end{align*}
  \]

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 \( e : I \rightarrow M \) and \( m : M \otimes M \rightarrow M \), for which the following laws hold:

\[
\begin{align*}
\tag{4} m \circ (e \otimes id) &= \lambda \\
\tag{5} m \circ (id \otimes e) &= \rho \\
\tag{6} m \circ (id \otimes m) &= m \circ (m \otimes id) \circ \alpha
\end{align*}
\]

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

For example, the category \( \text{Set} \) 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 \( \text{Integer} \), they construct a type of list of \( \text{Integers} \);
2) given a function, say from \( \text{Integers} \) to \( \text{Booleans} \), they can apply the function to every \text{Integer} to obtain a list of \text{Booleans}.

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

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

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

For example, the identity type constructor is a functor.

```haskell
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** \((f \circ g)\) \textbf{where}

\[
\text{Comp} :: f \ (g \ a) \to (f \circ g) \ a
\]

**instance** \((\text{Functor} \ f, \text{Functor} \ g) \Rightarrow \text{Functor} \ (f \circ g)\) \textbf{where}

\[
f \text{map} \ (\text{Comp} \ x) = \text{Comp} \ (f \text{map} \ (f \text{map} \ x))
\]

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 \([13]\).

**type** \((f \rightarrow g) = \forall \, x. \ f \, x \to g \, x\)

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 \(a\) \textbf{where}

Nothing :: Maybe \(a\)

Just \(\,:\,:\, a \to Maybe \, a\)

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

**instance** \(\text{Functor} \, \text{Maybe}\) \textbf{where}

\[
f \text{map} \, \text{Nothing} = \text{Nothing}
\]

\[
f \text{map} \, \text{Just} \, x = \text{Just} \, (f \, x)
\]

\[D. \text{ Monads}\]

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 \(\lambda\), \(\rho\), and \(\alpha\), which complete the monoidal category, are identities.

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

**class** Functor \(m \Rightarrow \text{Triple} \, m\) \textbf{where}

\[
\eta :: a \to m \ a
\]

\[
\mu :: m \ (m \ a) \to m \ a
\]

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

\[
\mu \circ \eta = \text{id}
\]

\[
\mu \circ f \text{map} \, \eta = \text{id}
\]

\[
\mu \circ f \text{map} \, \mu = \mu \circ \mu
\]

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 \(m\) \textbf{where}

\[
\text{return} :: a \to m \ a
\]

\[
(\backslash \!\!\!\!\!\!\!\!\Rightarrow) :: m \ a \to (a \to m \ b) \to m \ b
\]

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 \(\mu\) can be defined as \(\mu \, x = (x \Rightarrow \text{id})\), and that \((\Rightarrow)\) can be defined as \((x \Rightarrow k) = \mu \, (f \text{map} \, k \, x)\). Notice that Monad \(m\) does not require the type constructor \(m\) to be a functor: the \(\text{fmap}\) operation is derivable from the Monad instance by defining \(f \text{map} \, v = (v \Rightarrow \text{return} \circ f)\).

**Remark II.3** (Currying). The function \((\Rightarrow)\) 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:

\[
\text{curry} :: (a \times b \to c) \to (a \to (b \to c))
\]

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

\[
\text{uncurry} :: (a \to b \to c) \to (a \times b \to c)
\]

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

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 \textbf{where}

\[
\text{return} \, x = \text{Just} \, x
\]

\[
\text{Nothing} \Rightarrow f = \text{Nothing}
\]

\[
(\text{Just} \, x) \Rightarrow f = f \, x
\]

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.

\[E. \text{ Applicative functors}\]

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** \((\ast)\) \(f \, g \, a\) \textbf{where}

\[
\text{Day} :: f \, (b \to a) \times (g \, b) \to (f \ast g) \, a
\]

**instance** \((\text{Functor} \, f, \text{Functor} \, g) \Rightarrow \text{Functor} \, (f \ast g)\) \textbf{where}

\[
f \text{map} \, h \, (\text{Day} \, f \, f \, g \, x) = \text{Day} \, (f \text{map} \, (f \text{map} \, (f \text{map} \, h \, f \, f \, g) \, f) \, f) \, g x
\]

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 \(\lambda\), \(\rho\) and \(\alpha\) for this monoidal category are as follows (we give only one direction of each isomorphism).

\[
\lambda :: \text{Functor} \, f \Rightarrow (\text{Identity} \ast f) \, a \to f \, a
\]

\[
\lambda \, (\text{Day} \, (\text{Identity} \, f) \, x) = f \text{map} \, f \, x
\]
\[
\rho :: \text{Functor } f \Rightarrow (f \ast \text{Identity}) \ a \to f a \\
\rho (\text{Day } f \ (\text{Identity } b)) = \text{fmap } (\lambda h \to h \ b) \ f \\
\alpha :: (\text{Functor } f, \text{Functor } g) \Rightarrow (f \ast (g \ast h)) \ a \to ((f \ast g) \ast h) \ a \\
\alpha (\text{Day } f \ (\text{Day } g \ z)) = \text{Day } (\text{Day } (\text{fmap } (\circ) \ f) \ g) \ z
\]

The monoids in this monoidal category are functors \( f \) with operations \( e \) and \( m \) of type \( \text{Identity} \to f \), and \( f \ast f \to f \) 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:

\[\text{class } \text{Functor } f \Rightarrow \text{Applicative } f \text{ where}\]
\[\text{pure} :: a \to f a \\
(\otimes) :: f (b \to a) \times f b \to f a\]

Instantiating the general monoid laws \( \text{[4, 5, and 6] to this monoidal category, we obtain the applicative laws:}\)
\[\text{pure } f \otimes u = \text{fmap } f u \\
u \otimes \text{pure } x = \text{fmap } (\lambda h \to h \ x) \ u \\
(\text{fmap } (\circ) \ u \otimes v) \otimes w = u \otimes (v \otimes w)\]

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.

\[\text{instance } \text{Monad } m \Rightarrow \text{Applicative } m \text{ where}\]
\[\text{pure } x = \text{return } x \\
u \otimes \text{pure } x = \text{fmap } (\lambda h \to h \ x) \ u \\
(\text{fmap } (\circ) \ u \otimes v) \otimes w = u \otimes (v \otimes w)\]

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:

\[\text{data } K x a \text{ where}\]
\[K :: x \to K x a\]

\[\text{instance } \text{Functor } (K x) \text{ where}\]
\[\text{fmap } f (K x) = K (f x)\]

\[\text{instance } \text{Monoid } x \Rightarrow \text{Applicative } (K x) \text{ where}\]
\[\text{pure } a = K \text{u} \\
(K x) \otimes (K y) = K (x \otimes y)\]

### 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 \( \otimes \). Whereas the multiplicative monoid gives a model of sequencing, the \( \oplus \) operation models a non-deterministic choice, and models \( \otimes \) 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:

\[\text{class } \text{Dioid } d \text{ where}\]
\[z :: () \to d \\
u :: () \to d \\
(\oplus) :: d \times d \to d \\
(\otimes) :: d \times d \to d\]

This time, we expect the following laws to be satisfied:
\[a \oplus (u ()) = a \]
\[(u ()) \otimes a = a \]
\[a \oplus (b \otimes c) = (a \oplus b) \otimes c \]
\[a \oplus (z ()) = a \]
\[(z ()) \otimes a = a \]
\[a \oplus (b \oplus c) = (a \oplus b) \oplus c \]
\[(z ()) \oplus a = z () \]
\[(u ()) \oplus a = u () \]

The laws \( \text{[7 to 12]} \) express that \( d \) is a monoid with respect to \( (\otimes, u) \) and \( (\oplus, z) \). Laws \( \text{[13 and 14]} \) 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:

\[\text{type } \text{BinFun } a = a \times a \to a\]

\[\text{instance } \text{Dioid } (\text{BinFun } a) \text{ where}\]
\[z = \lambda (a, b) \to a \\
u = \lambda (a, b) \to b \\
f \oplus g = \lambda (a, b) \to f (g (a, b), b) \\
f \otimes g = \lambda (a, b) \to f (a, g (a, b))\]

In models of deep-backtracking non-determinism, law \( \text{[14]} \) 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 \[\text{[16]} \] 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 \textit{dioid homomorphism} is a function \( f : D_1 \to D_2 \) such that the dioid instances are preserved:
\[f (u ()) = u () \]
\[f (z ()) = z () \]
\[f (a \oplus b) = f (a \otimes f b) \\
f (a \oplus b) = f (a \otimes f b)\]

The type of \( f \) determines that the operations \( u, z, \oplus, \) and \( \otimes \) 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 \((\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)\) 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 \(\mathcal{C}\);
- \(\alpha, \lambda, \rho\) are natural isomorphisms:
  \[
  \alpha : A \otimes (B \otimes C) \cong (A \otimes B) \otimes C
  \]
  \[
  \lambda : I \otimes A \cong A
  \]
  \[
  \rho : A \otimes I \cong A
  \]
  \[
  \alpha : A \otimes (B \otimes C) \cong (A \otimes B) \otimes C
  \]
  \[
  \lambda : I \otimes A \cong A
  \]
  \[
  \rho : A \otimes I \cong A
  \]
- \(\kappa\) and \(\epsilon\) are natural transformations:
  \[
  \kappa : Z \otimes A \to Z
  \]
  \[
  \epsilon : I \oplus A \to I
  \]

Again, we expect these natural transformations to obey some coherence laws, which include those of \((\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)\) and \((\mathcal{C}, \oplus, Z, \lambda, \rho)\) 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:

\[
m \circ (e \otimes id) = \lambda
\]
\[
m \circ (id \otimes e) = \rho
\]
\[
m \circ (id \otimes m) = m \circ (m \otimes id) \circ \alpha
\]
\[
s \circ (z \otimes id) = \lambda
\]
\[
s \circ (id \otimes z) = \lambda
\]
\[
s \circ (id \otimes s) = s \circ (s \otimes id) \circ \alpha
\]
\[
m \circ (z \otimes id) = z \circ \kappa
\]
\[
s \circ (e \otimes id) = e \circ \kappa
\]

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 \(\text{Set}\) with \(Z = I = \{\ast\}\) and \(\oplus = \otimes = \ast\). Notice that \(\kappa = \epsilon = \ast : \{\ast\} \times A \to \{\ast\}\) 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 \((\mathcal{C}, \oplus, I, \alpha, \lambda, \rho, x, 1, \alpha, \pi_1, \kappa, \pi_1)\), where:

\[
\alpha = \{\pi_1, \pi_1 \circ \pi_2, \pi_2 \circ \pi_2\} : A \times (B \times C) \to (A \times B) \times C
\]
\[
\kappa = !_1: I \to A \to 1
\]

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:

\[
f \circ e_1 = e_2
\]
\[
f \circ z_1 = z_2
\]
\[
f \circ m_1 = m_2 \circ (f \otimes f)
\]

**C. Cartesian structure for functors**

We can use Lemma III.3 to extend the monoidal categories of endofunctors discussed in Sections II.3 and II.4 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**

\[
\text{Functor} K_1 \text{ where}
\]
\[
fmap f (K_1 ()) = K_1 ()
\]

Here, the unit type () represents the terminal object. Similarly, every 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**

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

**instance**

\[
\text{Functor} f, \text{Functor} g \Rightarrow \text{Functor} (f \times g) \text{ where}
\]
\[
fmap h (\text{Pair} (fa, ga)) = \text{Pair} (fmap h fa, fmap h ga)
\]

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 \(\ast\), the object \(I\) to be the identity functor \(\text{Identity}\) and the object \(Z\) to be the constant terminal functor \(K_1\).

The dioids in this dioid category are functors \(d\) with operations \(\zeta, \eta, \sigma, \mu\) 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:
subject to the laws:

\[
\begin{align*}
\mu \circ \eta &= \text{id} \\
\mu \circ \text{fmap} \eta &= \text{id} \\
\mu \circ \text{fmap} \mu &= \mu \circ \mu \\
\sigma \circ \text{pair} \text{id} \circ \text{zero} &= \text{fst} \\
\sigma \circ \text{pair} \text{zero} \circ \text{id} &= \text{snd} \\
\sigma \circ \text{pair} \sigma \circ \text{id} \circ \alpha &= \alpha \\
\mu \circ \zeta &= \zeta \\
\sigma \circ \text{pair} \eta \circ \text{id} &= \eta \circ \text{fst}
\end{align*}
\]

where \( \text{pair} \ f \ g \ (x,y) = (f \ x, g \ y) \) and \( \alpha \ (x,(y,z)) = ((x,y),z) \). The operations \( \eta \) and \( \mu \) 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 \text{MonadPlus} \ m \ **where**

\[
mzero :: m a \\
mplus :: m a \rightarrow m a \\
\]

subject to the following laws

\[
\begin{align*}
\text{mplus} \ mzero \ u &= u \\
\text{mplus} \ u \ mzero &= u \\
\text{mplus} \ (\text{mplus} \ u \ v) &= \text{mplus} \ (\text{mplus} \ u \ v) \ w \\
\text{mzero} \triangleright= \text{f} &= \text{mzero} \\
\text{mplus} \ (\text{return} \ x) \ u &= \text{return} \ x
\end{align*}
\]

in addition to those of monads.

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

**instance** MonadPlus Maybe **where**

\[
mzero = \text{Nothing} \\
mplus \text{Nothing} \ v &= \text{Nothing} \\
mplus \ (\text{Just} \ x) \ v &= \text{Just} \ x
\]

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

**E. Non-determinism Applicative Functors**

We turn again to Lemma 11.3 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 \( K_1 \rightarrow d \), \( \text{Identity} \rightarrow d \), \( d \times d \rightarrow d \) and \( d \times d \rightarrow d \) respectively. If we expand the definitions, we can present this information in a type class:

**class** DioidF \( d \ **where**

\[
\begin{align*}
\text{empty} :: () &\rightarrow d a \\
\text{pure} :: a &\rightarrow d a \\
(\langle\rangle) :: d a \times d a &\rightarrow d a \\
(\otimes) :: d \ (b \rightarrow a) \times d \ b &\rightarrow d a
\end{align*}
\]

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 \text{Alternative} \ f **where**

\[
\begin{align*}
\text{empty} :: f a &\rightarrow f a \\
(\langle\rangle) :: f a \rightarrow f a \\
\end{align*}
\]

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

\[
\begin{align*}
\text{empty} \ (\langle\rangle) &= u = u \\
\text{empty} \ (\langle\rangle) &= \text{empty} = u \\
\text{empty} \ (\langle\rangle) (\text{mempty} \ v) &= (\text{u} \ (\langle\rangle) v) \ (\langle\rangle) w \\
\text{empty} \ (\langle\rangle) \text{v} &= \text{mplus} \ u \ v \\
\text{pure} \ (\langle\rangle) \text{u} &= \text{pure} \ x
\end{align*}
\]

We can extend Remark 11.4 to the type classes MonadPlus and Alternative, and obtain the following result.

**instance** MonadPlus m \( \Rightarrow \text{Alternative} \ m **where**

\[
\begin{align*}
\text{empty} &= \text{mzero} \\
\text{mplus} \ (\text{mempty} \ v) &= \text{mplus} \ u \ v \\
\text{mplus} \ (\text{return} \ x) \ u &= \text{return} \ x
\end{align*}
\]

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 \text{Alternative} \ (K \\ d) **where**

\[
\begin{align*}
\text{empty} &= \text{MK} \ z \\
(\text{MK} \\ d_1) \ (\langle\rangle) &= \text{MK} \ (d_1 \otimes d_2)
\end{align*}
\]

**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 \( \text{FreeMon}_a \) is the free monoid over \( a \) when:

- \( \text{FreeMon}_a \) 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 \( \text{FreeMon}_a \) 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, \( a \otimes (u (b @ (c @ u ())) \otimes t) \) 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.

```haskell
data FreeMon a where
  Nil :: FreeMon a
  Cons :: a \times \text{FreeMon} a \rightarrow \text{FreeMon} a
```

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

```haskell
instance Monoid (FreeMon a) where
  u () = Nil
  Nil \otimes bs = bs
  (Cons a as) \otimes bs = Cons a (as \otimes bs)
```

The insertion function represents an \( a \) atom by a singleton list.

```haskell
ins :: a \rightarrow \text{FreeMon} a
ins a = Cons a Nil
```

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

```haskell
\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 \text{ as } = f a \otimes \text{univ } f \text{ 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 \( (C, \otimes, I) \).

**Definition IV.1 (Free monoid).** Let \( X \) be an object, the free monoid over \( X \) is a monoid \((X, e_X, m_X)\) 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:

\[
\begin{array}{c}
X \xrightarrow{\text{ins}} F \\
\downarrow f \\
M
\end{array}
\]

The morphism \( \text{ins} \) 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:

```haskell
data Free f a where
  Nil_a :: a \rightarrow \text{Free } f a
  Cons_a :: f (\text{Free } f a) \rightarrow \text{Free } f a
```

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

```haskell
instance Functor f \Rightarrow Monad (Free f) where
  return x = Nil_a x
  Nil_a x \triangleright f = f x
  Cons_a v \triangleright f = Cons_a (fmap (\triangleright ) v)
```

The insertion morphism and lifting are as follows:

```haskell
\text{ins} :: \text{Functor } f \Rightarrow f \rightarrow \text{Free } f
\text{ins } v = \text{Cons}_a (fmap \text{Nil}_a v)
\text{univ } : (\text{Functor } f, \text{Monad } m) \Rightarrow (f \rightarrow m) \rightarrow (\text{Free } f \rightarrow m)
\text{univ } f (\text{Nil}_a x) = \text{return } x
\text{univ } f (\text{Cons}_a v) = f (\text{fmap } (\triangleright ) v)
```

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:

\[ f^* \cong \text{Identity } + f \circ f^* \]

which leads to the following data type

```haskell
data FreeA f a where
  Nil_a :: a \rightarrow \text{FreeA } f a
  Cons_a :: (b \rightarrow a) \times \text{FreeA } f b \rightarrow \text{FreeA } f a
```
Again, we find the instantiation of the general formula yields the free applicative functor on a functor \( f \), with applicative instance:

\[
\text{instance} \quad \text{Functor} \ f \Rightarrow \text{Applicative} \ (\text{FreeA} \ f) \quad \text{where}
\]

\[
\begin{align*}
\text{pure} \ &x = \text{Nil}, \ x \\
\text{Nil}, \ h &\quad \otimes x = \text{fmap} \ h \ x \\
\text{Cons}, \ (h, x) \otimes y &= \text{Cons}, \ (\text{fmap} \ \text{uncurry} \ h) \ (fmap (_) \ x \otimes y)
\end{align*}
\]

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

\[
\begin{align*}
\text{ins} \ &:: \ \text{Functor} \ f \Rightarrow \ f \rightarrow \text{FreeA} \ f \\
\text{ins} \ v &= \text{Cons}, \ (\text{fmap} \ \text{const} \ v) \ (\text{Nil}, \ ()) \\
\text{univ} \ &:: \ \text{(Functor} \ f, \ \text{Applicative} \ g \Rightarrow \ (f \rightarrow g) \rightarrow \text{FreeA} \ f \rightarrow g \text{)} \\
\text{univ} \ f \ (\text{Nil}, \ x) &= \text{pure} \ x \\
\text{univ} \ f \ (\text{Cons}, \ v \ r) &= f \ v \otimes \text{univ} \ f \ r
\end{align*}
\]

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 [27]. 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 \((C, \oplus, I, \alpha_\oplus, \lambda_\oplus, \rho_\oplus, \varnothing, Z, \alpha_\otimes, \lambda_\otimes, \rho_\otimes, \kappa_\otimes, \kappa_\otimes)\), and then obtain the ordinary notion for the dioid category \( C \) 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 \( \text{ins} : X \rightarrow F \) such that for any dioid \((D, z_D, e_D, s_D, m_D)\) and morphism \( f : X \rightarrow G \) there exists a unique dioid homomorphism \( \text{univ}(f) : F \rightarrow G \) 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:

\[
\begin{align*}
X &= 0 \cup 1 \cup T \\
T &= X \cup (S \times_\oplus 1) \cup (S \times_\otimes T) \cup (M \times_\otimes 0) \cup (M \times_\otimes T) \\
S &= X \cup (M \times_\otimes 0) \cup (M \times_\otimes T) \\
M &= X \cup (S \times_\otimes 1) \cup (S \times_\otimes T)
\end{align*}
\]

where \( 0 = 1 = \{ * \} \) and \( \times_\otimes = \times_\otimes \times \). 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 → Free a

**data** Term a **where**

- LiftT :: a → Term a
- SumT1 :: Sum a → Term a
- SumT2 :: Sum a × Term a → Term a
- MultT1 :: Mult a → Term a
- MultT2 :: Mult a × Term a → Term a

**data** Sum a **where**

- LiftS :: a → Sum a
- MultS1 :: Mult a → Sum a
- MultS2 :: Mult a × Term a → Sum a

**data** Mult a **where**

- LiftM :: a → Mult a
- SumM1 :: Sum a → Mult a
- SumM2 :: Sum a × Term a → 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.

\[
\begin{align*}
\otimes_T &= \text{Term} \ a \rightarrow \text{Free} \ a \rightarrow \text{Term} \ a \\
\text{LiftT} \ x &= \otimes_T \text{Zero} = \text{MultT1} (\text{LiftM} \ x) \\
\text{LiftT} \ x &= \otimes_T \text{Unit} = \text{LiftT} \ x \\
\text{LiftT} \ x &= \otimes_T \text{Term} \ y = \text{MultT2} (\text{LiftM} \ x, y) \\
\text{SumT1} \ x &= \otimes_T \text{Zero} = \text{MultT1} (\text{SumM1} \ x) \\
\text{SumT1} \ x &= \otimes_T \text{Unit} = \text{SumT1} \ x \\
\text{SumT1} \ x &= \otimes_T \text{Term} \ y = \text{MultT2} (\text{SumM1} \ x, y) \\
\text{SumT2} \ (x, y) &= \otimes_T \text{Zero} = \text{MultT1} (\text{SumM2} (x, y)) \\
\text{SumT2} \ (x, y) &= \otimes_T \text{Unit} = \text{SumT2} (x, y) \\
\text{SumT2} \ (x, y) &= \otimes_T \text{Term} \ z = \text{MultT2} (\text{SumM2} (x, y), z) \\
\text{MultT1} \ x &= \otimes_T \text{y} = \text{MultT1} \ x \\
\text{MultT2} \ (x, y) &= \otimes_T \text{z} = \text{MultT2} (x, y \otimes_T z)
\end{align*}
\]

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

**instance** Dioid (Free a) **where**

- \( z () = \text{Zero} \)
- \( u () = \text{Unit} \)
- \( \text{Zero} \otimes x = \text{Zero} \)
- \( \text{Unit} \otimes v = \text{Unit} \)
- \( \text{Term} \ x \otimes y = \text{Term} (x \otimes_T y) \)
- \( \text{Zero} \otimes v = \text{Zero} \)
- \( \text{Unit} \otimes x = x \)
- \( \text{Term} \ x \otimes y = \text{Term} (x \otimes_T y) \)
The insertion morphism and lifting are as follows:

\[
\begin{align*}
\text{ins} & : a \rightarrow \text{Free } a \\
\text{ins } a & = \text{Term } (\text{Lift } T a) \\
\text{univ} & : \text{Dioid } d \Rightarrow (a \rightarrow d) \rightarrow \text{Free } a \rightarrow d \\
\text{univ } f & \text{ Zero } = z () \\
\text{univ } f \text{ Unit } = u () \\
\text{univ } f \text{ (Term } v) & = \text{univ } T f v
\end{align*}
\]

For the tentative free dioid over an object in the equations, we obtain the following system of equations:

\[
\begin{align*}
\text{constructors } & \text{ correspond to the additive structure and which renamings } \\
\text{are } & \text{ auxiliary functions that work as expected.}
\end{align*}
\]

In the case of dioids, the system of recursive equations

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

Replacing \( \cup \) 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 \( X \):

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

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

\[
\begin{align*}
\text{data } \text{Free}_o f x & \text{ where } \\
\text{Zero}_o & : K_1 x \rightarrow \text{Free}_o f x \\
\text{Unit}_o & : \text{Identity } x \rightarrow \text{Free}_o f x \\
\text{Term}_o & : \text{Term}_o f x \rightarrow \text{Free}_o f x
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Term}_o f x & \text{ where } \\
\text{Lift}_{T o} & : f x \rightarrow \text{Term}_o f x \\
\text{Sum}_{T o}^1 & : \text{Sum}_o f x \times \text{Identity } x \rightarrow \text{Term}_o f x \\
\text{Sum}_{T o}^2 & : \text{Sum}_o f x \times \text{Term}_o f x \rightarrow \text{Term}_o f x \\
\text{Mult}_{T o}^2 & : \text{Mult}_o f (K_1 x) \rightarrow \text{Term}_o f x
\end{align*}
\]

Mult\(_T^2\) is the equivalent to \( (\otimes) \) for free ordinary dioids, and it is defined as follows:

\[
\begin{align*}
(\otimes) & : \text{Functor } f \Rightarrow \\
\text{Term}_o f (a \rightarrow b) & \rightarrow \text{Free}_o f a \rightarrow \text{Term}_o f b
\end{align*}
\]
We close our discussion for free structures by presenting the insertion function for
again that it follows the pattern presented for ordinary dioids.

shallow-backtracking non-determinism. By considering
be used in this setting to obtain a free arrow with a notion of
whether the general formulation for free dioids presented can
MonadPlus
backtracking non-determinism. By considering
Alternative
By instantiation, this resulted in the construction of the free
endofunctors, we have obtained a set of laws that those type
classes should obey.

As further work, it would be interesting to study the notion
dioid in the dioid category of endoproducts, 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.

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
endofunctions, 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 endoproducts, 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
http://dl.acm.org/citation.cfm?id=77530.77533
[2] ———, “Notions of computation and monads,” Information and
http://dx.doi.org/10.1016/0890-5401(91)90052-4
ACM Conference on LISP and Functional Programming, ser. LFP ’90.
http://doi.acm.org/10.1145/91556.91592

https://doi.org/10.1016/0167-6423(90)90056-I
[Online]. Available: http://dx.doi.org/10.1017/S0956796807006526
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. Laubach, E. Meijer,
[Online]. Available: http://dx.doi.org/10.1017/S0956796809007291
traversals,” in Proceedings Fourth Workshop on Mathematically
Structured Functional Programming, Tallinn, Estonia, 25 March
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,
http://doi.acm.org/10.1145/2628136.2628144
abs/1406.4823
Texts in Mathematics. Springer-Verlag, 1971, no. 5, second edition,
1998.
the function “reverse”,” Information Processing Letters, vol. 22,
1016/0020-0190(86)90059-1
monoids,” in Proceedings of the 9th International Conference on
Mathematics of Program Construction, ser. MPC ’08. Springer-Verlag,
978-3-540-70594-9_20
http://dblp.uni-trier.de/db/journals/jfp/jfp22.html#Spivey12
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
http://doi.acm.org/10.1145/2790449.2790514
http://hdl.handle.net/10277/4693
[18] E. Bainbridge, P. J. Freyd, A. Scedrov, and J. P. Scott, “Functorial
polymorphism,” Theoretical Computer Science, vol. 70, no. 1, pp. 35
article/pii/0304397590900157
http://journals.cambridge.org/article
http://dx.doi.org/10.1017/S0004972700005433
[20] R. M. Burstall and J. A. Goguen, Algebra, Theories and Freeness:
An Introduction for Computer Scientists. Dordrecht: Springer
1007/978-94-009-7939-5
algebras, free monoids, colimits, associated sheaves, and so on,” Bulletin
of the Australian Mathematical Society, vol. 22, no. 01, pp. 1–83,