A Unified View of Monadic and Applicative Non-determinism

Exequiel Rivas, Mauro Jaskelioff
CIFASIS-CONICET
Universidad Nacional de Rosario, Argentina

Tom Schrijvers
KU Leuven, Belgium

Abstract

It is well-known that monads are monoids in the category of endofunctors, and in fact so are applicative functors. Unfortunately, monoids do not have enough structure to account for computational effects with non-determinism operators.

This article recovers a unified view of computational effects with non-determinism by extending monoids to near-semirings with both additive and multiplicative structure. This enables us to generically define free constructions as well as a novel double Cayley representation that optimises both left-nested sums and left-nested products.

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

1. Introduction

Both monads [21] and applicative functors [20] have been successful in structuring and modularising programs, and there is a considerable amount of research on techniques and properties for programming with either of them. Through a unified view it is possible to leverage the knowledge of one

Email addresses: `rivas@cifasis-conicet.gov.ar` (Exequiel Rivas), `jaskelioff@cifasis-conicet.gov.ar` (Mauro Jaskelioff), `tom.schrijvers@cs.kuleuven.be` (Tom Schrijvers) structure and apply it to programs for the other. Rivas and Jaskelioff [25] presented such a unified view of both constructions, as monoids in monoidal categories, and used it to translate the well-known Cayley representation for monads (aka codensity transformation) and obtain a novel Cayley representation for applicatives.

Many computations deal with non-determinism. Non-determinism naturally occurs, for example, in logic programming languages and in parser combinators. In order to account for non-determinism, the interface of monads and applicative functors are extended with additional operations, giving rise to richer structures. As a consequence of this extension, the existing correspondence between the two algebraic structures falls short, since looking at them as monoids in monoidal categories only tells part of the story. Hence, it is necessary to extend the abstract model from monoids (which only have a multiplicative structure) to near-semirings (which have both a multiplicative and an additive structure).

In this article, we generalise the abstract model that connects monads and applicative functors from monoids to near-semirings in order to account for non-determinism. This yields a precise connection between their non-determinism counterparts, which we exploit in order to analyse, abstract, and derive constructions.

This article is an extended version of the paper [26] presented at PPDP'15. This version has an improved and updated presentation, and adds new technical contributions (marked with ★ below).

The contributions of the paper are as follows:

- We define the notion of non-determinism monad and non-determinism applicative functor.
- We present a generalised form of near-semirings, and establish that both non-determinism monads and non-determinism applicative functors are instances of this generalised notion.
- We construct the free near-semiring and use the generalisation to obtain free constructions for both non-determinism monads and non-determinism applicative functors.
- We construct the double Cayley representation for generalised near-semirings and specialise it to obtain representations for non-determinism monads and non-determinism applicative functors.

★ We show how to extend any applicative functor to a non-determinism one, by translating the ListT construction for monads. We contrast this solution to the solution obtained by composing an applicative functor with the list applicative functor.

★ We show how to extend a monad to a non-determinism monad in the most general way possible. That is, we construct the free non-determinism monad over a monad. This construction is different from the free non-determinism monad on a functor, since we want to recycle the existing monad structure.

- We demonstrate the use of the constructions on two examples: combinatorial search and interleaving parsers.

The rest of the paper is structured as follows. In the next section we review the connection between monads and applicatives, which serves as inspiration for the rest of the paper. In Section 3, we present the generalisation framework that provides the tools for the remaining sections: generalised near-semirings. Then, in Section 4, we show some examples of non-determinism monads and applicatives. In Section 5, we construct the free near-semiring over an object in different near-semiring categories. After that, in Section 6, we introduce the double Cayley representation for a near-semiring. Then, we present two ways of constructing near-semirings from monoids: first generalising the list monad transformer in Section 7, and then constructing the free near-semiring over a monad in Section 8. In Section 9, we give some applications of the developed theory. Finally, we present the related work and conclusions in Sections 10 and 11.

Haskell Source Code. The Haskell source code presented in this paper, including all elided definitions, is available from the second and third authors' webpages. Several of the Haskell definitions given in this paper can be rather daunting. However, our intent is not for the reader to interpret them operationally, but instead to see them as instantiations of a general construction. After all, the exposure of this generic pattern is the main contribution of this work. Therefore, it is advisable to focus on the interface and properties of the different definitions, rather than the details of their implementation.

2. Connecting Monads and Applicative Functors

In Haskell, the interfaces of monads and applicative functors are specified by type classes. The Monad interface provides a basic structure for computations:

$$\begin{array}{l}\textbf{class}\ \mathsf{Monad}\ m\ \textbf{where}\\ \quad \mathsf{return} :: a \to m\ a\\ \quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

where the operation return injects values, and $\gg\!=$ sequentially composes computations. The Applicative interface, in contrast, is as follows:

$$\begin{array}{l}\textbf{class}\ \mathsf{Applicative}\ m\ \textbf{where}\\ \quad \mathsf{pure} :: a \to m\ a\\ \quad (\circledast) :: m\ (b \to a) \to m\ b \to m\ a\end{array}$$

where the pure operation also injects pure values, and $\circledast$ provides application under an effect.

It is well-known that every Monad is an Applicative, and that there are Applicative instances which are not Monads. It is less well-known that both constructions can be explained in terms of a single unifying structure.

Revealing Similarities. There are some similarities between the two interfaces: Both provide a way of injecting pure values, and both have a form of composition operator. However, the similarities are more profound. Monads are monoids in a monoidal category of endofunctors, and so are applicative functors! (The two monoidal categories are different though.) In order to better expose the similarities, let us consider this alternative (but equivalent) presentation of the monad interface:

$$\begin{array}{l}\textbf{class}\ \mathsf{Functor}\ m \Rightarrow \mathsf{Triple}\ m\ \textbf{where}\\ \quad \mathsf{eta}^{\mathsf{T}} :: a \to m\ a\\ \quad \mathsf{mu}^{\mathsf{T}} :: (m \circ m)\ a \to m\ a\end{array}$$

where $\circ$ is functor composition (i.e. $(f \circ g)\ a = f\ (g\ a)$).

A Triple m is equivalent to a Monad m: we set $\mathsf{eta}^{\mathsf{T}} = \mathsf{return}$, and we can define one composition operator in terms of the other. From mu^{T} we can define $t \gg\!= f = \mathsf{mu}^{\mathsf{T}}\ (\mathsf{fmap}\ f\ t)$, and from $\gg\!=$ we can define $\mathsf{mu}^{\mathsf{T}}\ t = t \gg\!= \mathsf{id}$.

Let us modify the Triple interface and replace functor composition with the following datatype which implements the so-called *Day convolution* [5]:

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

where the type b is existentially quantified.[1]

Replacing functor composition $\circ$ with Day convolution $\star$ we obtain the class $\mathsf{DayTriple}$:

```
class Functor m ⇒ DayTriple m where
  eta^D :: a → m a
  mu^D :: (m ⋆ m) a → m a
```

By expanding the definition of $\star$ in mu^{D}, we discover that $\mathsf{DayTriple}$ is equivalent to the $\mathsf{Applicative}$ interface.

Relating Monad and Applicative Laws. The connection between monads and applicative functors is even deeper as also the laws required by each interface can be related. The laws for both interfaces are:

Monad laws	Applicative laws
$\mathsf{return}\ x \ggg u = u\ x$	$\mathsf{pure}\ f \circledast u = f \langle\$\rangle\ u$
$u \ggg \mathsf{return} = u$	$u \circledast \mathsf{pure}\ x = (\$x) \langle\$\rangle\ u$
$(u \ggg v) \ggg w = u \ggg (\lambda x \to v\ x \ggg w)$	$((\circ) \langle\$\rangle\ u \circledast v) \circledast w = u \circledast (v \circledast w)$

where $(\$) :: (a \to b) \to a \to b$ is function application, and $(\langle\$\rangle) :: \mathsf{Functor}\ f \Rightarrow (a \to b) \to f\ a \to f\ b$ maps a function under a functor (i.e. it is an infix version of fmap). In the case of applicatives, we also assume that the functor laws hold.

While one can see some similarities, the exact relation between the two sets of laws is not obvious. In order to see the connection, some subtleties must be made evident. We postpone this to the next section, which introduces the abstract formal framework that unifies the two structures.

Non-Deterministic Computations. The notion of non-deterministic computation we consider in this article involves two additional operations: failure

[1]The reasoning behind the notation is that the type $(\exists b.\, \mathsf{T}\ b) \to \mathsf{A}$ is equivalent to $\forall b.\, \mathsf{T}\ b \to \mathsf{A}$.

and non-deterministic choice between two computations. In monadic computations, these two additional operations are captured by the MonadPlus type class:

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

For applicative computations, the non-deterministic interface is provided by the Alternative type class:

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

Instances of the Monad type class are expected to obey certain laws. Likewise, a MonadPlus instance is a *non-determinism monad* when the following laws hold:

$$m \text{ `mplus` } \mathsf{mzero} = m \tag{1}$$

$$\mathsf{mzero} \text{ `mplus` } m = m \tag{2}$$

$$m_1 \text{ `mplus` } (m_2 \text{ `mplus` } m_3) = (m_1 \text{ `mplus` } m_2) \text{ `mplus` } m_3 \tag{3}$$

$$\mathsf{mzero} \ggg\!= k = \mathsf{mzero} \tag{4}$$

$$(m_1 \text{ `mplus` } m_2) \ggg\!= k = (m_1 \ggg\!= k) \text{ `mplus` } (m_2 \ggg\!= k) \tag{5}$$

The first three laws say that mplus and mzero form a monoid, and the last two laws specify the interaction with the Monad structure. These last two laws are commonly known as *left zero* and *left distribution*.

One should not expect every MonadPlus instance to be a non-determinism monad, as there are uses of the same interface for other purposes that require different sets of laws.

So what about laws for Alternative? As far as we know, the only three established laws that every Alternative instance f should obey are those of a monoid $f\ a$, with the operation ⟨|⟩ acting as multiplication and empty acting as its unit:

$$m \langle|\rangle \mathsf{empty} = m \tag{6}$$

$$\mathsf{empty} \langle|\rangle m = m \tag{7}$$

$$m_1 \langle|\rangle (m_2 \langle|\rangle m_3) = (m_1 \langle|\rangle m_2) \langle|\rangle m_3 \tag{8}$$

However, considering the connection between monads and applicatives, the laws (4) and (5) of non-determinism monads suggest two further laws:

$$\mathsf{empty} \circledast x = \mathsf{empty} \tag{9}$$

$$(f \langle|\rangle g) \circledast x = (f \circledast x) \langle|\rangle (g \circledast x) \tag{10}$$

An Alternative instance is a *non-determinism applicative functor* when laws (6–10) hold.

The connection between monads and applicative functors allows us to translate knowledge from one structure to the other. However, in order to make the connection precise, we need a formal framework.

3. Monoidal and Near-Semiring Categories

In this section we introduce the formal framework which allows us to establish the connection between monads and applicative functors, and between non-determinism monads and non-determinism applicative functors. As a refresher, we start with ordinary monoids and near-semirings over sets.

3.1. Background: Monoids and Near-Semirings

A *monoid* $(M, \dot{\times}, \dot{1})$ is a triple consisting of a set M, together with an operation $\dot{\times} : M \times M \to M$ and an element $\dot{1} \in M$ such that the following axioms hold for all a, b, and $c \in M$:

$$a \mathbin{\dot{\times}} \dot{1} = a \tag{11}$$

$$\dot{1} \mathbin{\dot{\times}} a = a \tag{12}$$

$$a \mathbin{\dot{\times}} (b \mathbin{\dot{\times}} c) = (a \mathbin{\dot{\times}} b) \mathbin{\dot{\times}} c \tag{13}$$

The operation $\dot{\times}$ is called the multiplication of the monoid, while the element $\dot{1}$ is called the unit. We usually refer to a monoid $(M, \dot{\times}, \dot{1})$ simply by its carrier set M.

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

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

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

When two monoids align in a particular way, they form a near-semiring, the central structure in this paper. Formally, a *near-semiring* is defined as a quintuple $(M, \dot{\times}, \dot{1}, \dot{+}, \dot{0})$ where both $(M, \dot{\times}, \dot{1})$ and $(M, \dot{+}, \dot{0})$ are monoids for the same set M; moreover, the following laws relate both structures:

$$\dot{0} \mathbin{\dot{\times}} a = \dot{0} \tag{14}$$
$$(a \mathbin{\dot{+}} b) \mathbin{\dot{\times}} c = (a \mathbin{\dot{\times}} c) \mathbin{\dot{+}} (b \mathbin{\dot{\times}} c) \tag{15}$$

Here $\dot{\times}$ is the multiplication of the near-semiring, $\dot{+}$ is the addition, $\dot{1}$ is the unit, and $\dot{0}$ is the zero.

Remark. In the literature, sometimes the unit is not required: $(M, \dot{\times})$ is only expected to be a semigroup. In this article we consider only near-semirings with unit, and call them simply near-semirings. Also, because only distribution from the right is required, this structure is sometimes called a *right near-semiring.*

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

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

Instances of Nearsemiring must satisfy the near-semiring axioms. Instead of function names, we use infix operators for denoting multiplication and addition.

3.2. Background: Monoidal Categories

Monoidal categories generalise the notion of monoids from sets A to categories $\mathcal{C}$. We remind the reader of the following categorical concepts:

- A *bifunctor* is a functor from a product category. For example, the Cartesian product is a bifunctor $- \times - : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set}$ with action on arrows $(f \times g)$ taking an object (x, y) of the product category to the set $(f\,x, g\,y)$.

- A *natural transformation* $\tau : F \to G$ is a family of morphisms between functors F and G, indexed by objects, such that for all objects A and B, and every morphism $f : A \to B$ the equation $G f \circ \tau_A = \tau_B \circ f$ holds. In order to avoid clutter, we sometimes leave the component implicit and write $\tau : F\,A \to G\,A$ instead of $\tau_A : F\,A \to G\,A$.

- A *natural isomorphism* is a natural transformation for which each component is an isomorphism.

In order to prepare for the generalisation, we first express both operations $\dot{\times}$ and $\dot{1}$ of ordinary monoids as unary morphisms by modifying the unit to take a trivial argument from the singleton set $\{*\}$

$$\begin{aligned} m &: A \times A \to A \\ e &: \{*\} \to A \end{aligned}$$

We generalise from a set A to a category $\mathcal{C}$ by making two replacements:

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

This yields the morphisms:

$$\begin{aligned} m &: M \otimes M \to M \\ e &: I \to M \end{aligned}$$

We expect $\otimes$ and I to work like $\times$ and $\{*\}$, in the sense that $\times$ is associative and $\{*\}$ is a unit with respect to it (up to isomorphism). For that, we require the following natural isomorphisms expressing that $\otimes$ is associative, and that I is a left and right unit with respect to $\otimes$:

$$\begin{aligned} \alpha_{A,B,C} &: A \otimes (B \otimes C) \cong (A \otimes B) \otimes C \\ \lambda_A &: I \otimes A \cong A \\ \rho_A &: A \otimes I \cong A \end{aligned}$$

These natural isomorphisms should interact coherently [19]. The sextuple $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$ is known as a *monoidal category*.

Monoids. A *monoid in a monoidal category* $(\mathcal{C}, \otimes, I, \alpha, \lambda, \rho)$ is an object M, together with morphisms $m : M \otimes M \to M$ and $e : I \to M$, for which the following laws hold:

$$m \circ (e \otimes \mathsf{id}) = \lambda \tag{16}$$
$$m \circ (\mathsf{id} \otimes e) = \rho \tag{17}$$
$$m \circ (\mathsf{id} \otimes m) = m \circ (m \otimes \mathsf{id}) \circ \alpha \tag{18}$$

The laws for monoids in monoidal categories are the corresponding generalisations of equations (11), (12), and (13). The category Set is a monoidal category with the Cartesian product as its tensor and the terminal object (a singleton set) as unit object; monoids in this monoidal category reduce to ordinary monoids.

Monads are Monoids in the Category of Endofunctors. This article mainly works in the category Endo of endofunctors on a category $\mathcal{C}$, which consists of endofunctors as objects and natural transformations as morphisms.

We can give this category monoidal structure by choosing the tensor $\otimes$ to be $\circ$, the composition of functors. Formally, composition of functors F and G is $(F \circ G)(X) = F(G(X))$. Functor composition is a bifunctor: let $f : F \to H$ and $g : G \to K$ be natural transformations, then the action on morphisms $f \circ g : (F \circ G) \to (H \circ K)$ is defined by $f \circ g = f \circ Fg$ (which by naturality is the same as $Hg \circ f$). The object I is the identity functor $\mathsf{Id}(X) = X$. 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 monads. The two associated natural transformations are:

$$e : \mathsf{Id} \to M$$
$$m : M \circ M \to M$$

which correspond to the Triple type class introduced in Section 2.

The three monoid laws (16), (17), and (18) are the usual monad laws found in category theory textbooks and, when expressed in terms of $\gg\!=$, are equivalent to the laws for monads described in Section 2.

The Cartesian structure is not the only monoidal structure on Set, disjoint union being another. Analogously, functor composition is not the only monoidal structure on the category of endofunctors. As a consequence, monads are not the only monoids in the category of endofunctors. Another important class are *applicative functors*, introduced by McBride and Paterson [20] as a way to capture certain effectful computations that do not fit well in the monadic framework.

Applicative Functors are Monoids in the Category of Endofunctors. Applicative functors are based on a category of endofunctors, but with different tensor than monads: the *Day convolution* [5]. There are different presentations for the Day convolution in Haskell [25]. The presentation we chose in Section 2 with the datatype $\star$ directly results in the $\mathsf{Applicative}$ type class.

$$\begin{array}{l}
\mathbf{data}\ (\star)\ f\ g\ a = \forall b.\,\mathsf{Day}\ (f\ (b \to a))\ (g\ b)\\
\mathbf{instance}\ (\mathsf{Functor}\ f, \mathsf{Functor}\ g) \Rightarrow \mathsf{Functor}\ (f \star g)\ \mathbf{where}\\
\quad \mathsf{fmap}\ h\ (\mathsf{Day}\ \mathit{ff}\ gx) = \mathsf{Day}\ ((\lambda f \to h \circ f)\ \langle\$\rangle\ \mathit{ff})\ gx
\end{array}$$

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 natural isomorphisms λ, ρ, and α for this monoidal category are as follows.

$$\begin{array}{l}
\lambda :: \mathsf{Functor}\ f \Rightarrow (\mathsf{Identity} \star f)\ a \to f\ a\\
\lambda\ (\mathsf{Day}\ (\mathsf{Identity}\ f)\ x) = f\ \langle\$\rangle\ x\\
\rho :: \mathsf{Functor}\ f \Rightarrow (f \star \mathsf{Identity})\ a \to f\ a\\
\rho\ (\mathsf{Day}\ f\ (\mathsf{Identity}\ b)) = (\$b)\ \langle\$\rangle\ f\\
\alpha :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g) \Rightarrow (f \star (g \star h))\ a \to ((f \star g) \star h)\ a\\
\alpha\ (\mathsf{Day}\ f\ (\mathsf{Day}\ g\ z)) = \mathsf{Day}\ (\mathsf{Day}\ ((\circ)\ \langle\$\rangle\ f)\ g)\ z
\end{array}$$

A monoid m in this monoidal category has as operations natural transformations $m \star m \to m$ and $\mathsf{Identity} \to m$. Expanding the definition of Day convolution and identity functor leads to the well-known operations of the $\mathsf{Applicative}$ type class:

$$\begin{array}{l}
\mathbf{class}\ \mathsf{Functor}\ m \Rightarrow \mathsf{Applicative}\ m\ \mathbf{where}\\
\quad \mathsf{pure} :: a \to m\ a\\
\quad (\circledast) :: m\ (a \to b) \to m\ a \to m\ b
\end{array}$$

The laws for applicatives introduced in Section 2 are obtained by instantiating the three laws (16), (17), and (18) to this monoidal category. The differences between the laws for monads and the laws for applicatives that we were not able to explain before, can now be attributed to the fact the monads are monoids in a strict monoidal category, where the morphisms λ, ρ, and α are identities, and thus play no role.

3.3. Near-Semiring Categories

A near-semiring is a combination of a multiplicative monoid and an additive monoid. Consequently, near-semiring categories extend monoidal categories to add support for two monoids. This requires a category $\mathcal{C}$ that is monoidal in two ways. A *near-semiring category* is a category with a multiplicative monoidal structure $(\mathcal{C}, \otimes, I_{\otimes}, \alpha_{\otimes}, \lambda_{\otimes}, \rho_{\otimes})$, an additive monoidal structure $(\mathcal{C}, \oplus, I_{\oplus}, \alpha_{\oplus}, \lambda_{\oplus}, \rho_{\oplus})$, and two additional natural transformations, κ (for cancellation) and δ (for distribution) that connect the multiplicative and additive structure

$$\begin{aligned}
&\kappa : I_{\oplus} \otimes M \to I_{\oplus} \\
&\delta : (M_1 \oplus M_2) \otimes M_3 \to (M_1 \otimes M_3) \oplus (M_2 \otimes M_3)
\end{aligned}$$

As happens with monoidal categories, all these transformations must interact coherently. Yet, note that δ and κ are not required be natural isomorphisms. There is only a lax distributivity of the multiplicative structure over the additive structure.

A *near-semiring in a near-semiring category* consists of an object M and four morphisms:

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

which form the multiplicative monoid and the additive monoid, such that the following interaction laws hold:

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

In particular, we recover the ordinary near-semirings presented in Section 3.1 by setting $\mathcal{C} = \mathsf{Set}$, $\otimes = \oplus = \times$, and $I_{\otimes} = I_{\oplus} = \{*\}$.

Cartesian Structure as Additive Structure. In the remainder of this article we will assume we work with *Cartesian categories*, i.e. categories with (finite) products and terminal object. Moreover, we will fix the additive structure $\oplus$ and $I_\oplus$ to be the bifunctor $\times$ and the terminal object 1 respectively. That is, we will consider the additive structure of near-semiring categories to be *Cartesian.* The following definitions are easily generalised to an arbitrary bifunctor $\oplus$, if desired. As usual with the Cartesian structure, the three natural isomorphisms $\alpha_\oplus$, $\lambda_\oplus$, and $\rho_\oplus$ are named assoc, π_1, and π_2, respectively. Moreover, given a monoidal structure $(\mathcal{C}, \otimes, I)$, a distribution morphism δ can be defined as $\delta = \langle \pi_1 \otimes \mathsf{id}, \pi_2 \otimes \mathsf{id} \rangle$ and a cancellation morphism κ can be defined as $\kappa = !_{1 \otimes M}$ where $!_X : X \to 1$ is the family of unique homomorphisms associated with the terminal object 1. This construction is summarized in the following theorem:

Theorem 3.1. *If* $(\mathcal{C}, \otimes, I_\otimes, \alpha_\otimes, \lambda_\otimes, \rho_\otimes)$ *is a monoidal category where* $\mathcal{C}$ *is Cartesian, then* $\mathcal{C}$ *is a near-semiring category with multiplicative structure* $(\mathcal{C}, \otimes, I_\otimes, \alpha_\otimes, \lambda_\otimes, \rho_\otimes)$*, additive structure* $(\mathcal{C}, \times, 1, \mathsf{assoc}, \pi_1, \pi_2)$ *and the natural transformations* δ *and* κ *as defined above.*

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

Near-semiring Category of Endofunctors. We assume that the endofunctors in Endo are over a Cartesian category $\mathcal{C}$, and therefore Endo is also Cartesian: given two $\mathcal{C}$-endofunctors F and G, their product is defined point-wise as

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

The terminal endofunctor 1 is simply the constant endofunctor K_1 which maps every object to the terminal object 1 of $\mathcal{C}$, and every morphism to id_1. As a corollary of Theorem 3.1, any monoidal category of endofunctors over a Cartesian category is a near-semiring category. In particular, this means that both the monoidal category supporting monads and the monoidal category supporting applicative functors are near-semiring categories.

Non-Determinism Monads are Near-Semirings in the Category of Endofunctors. A monad is a monoid in a monoidal category of endofunctors (with functor composition as bifunctor). By Theorem 3.1 we obtain that this monoidal category is a near-semiring category, and we can consider near-semirings in it. A near-semiring in this category consists of a monad M and two additional natural transformations:

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

In Haskell, we write their types as $a :: \forall b.\, m\ b \to m\ b \to m\ b$ and $z :: \forall b.\, m\ b$, and aptly call them mplus and mzero after $\mathsf{MonadPlus}$'s methods.

The first three laws (1)–(3) for non-determinism monads described in Section 2 directly correspond to the laws for near-semirings that we obtain for this particular category. The remaining two laws for this particular category are superficially different:

$$\mathsf{mu}^{\mathsf{T}}\ \mathsf{mzero} = \mathsf{mzero} \tag{19}$$

$$\mathsf{mu}^{\mathsf{T}}\ (m_1\ \text{`}\mathsf{mplus}\text{`}\ m_2) = (\mathsf{mu}^{\mathsf{T}}\ m_1)\ \text{`}\mathsf{mplus}\text{`}\ \mathsf{mu}^{\mathsf{T}}\ m_2 \tag{20}$$

Yet, given the correspondence between mu^{T} and $(\gg\!=)$ and the naturality of mzero and mplus, it is easy to derive laws (4) and (5) from these ones and vice versa.

Non-determinism Applicatives are Near-Semirings in a Category of Endofunctors. An applicative functor is a monoid in a monoidal category of endofunctors (with Day convolution as bifunctor). As before, by Theorem 3.1 we know this monoidal category is a near-semiring category, and we may study near-semirings in it. A near-semiring in this category consists of an applicative functor, and two additional natural transformations:

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

In Haskell, we write their types as $a :: \forall b. f\ \ b \to f\ \ b \to f\ \ b$ and $z :: \forall b. f\ \ b$, and aptly call them $\langle | \rangle$ and empty after the corresponding methods of the $\mathsf{Alternative}$ type class.

The laws that we obtain for near-semirings in this particular near-semiring category are the ones for $\mathsf{Alternative}$ described in Section 2.

4. Examples of Non-determinism Structures

In this section we present several examples of non-determinism monads and non-determinism applicatives. As discussed before, these are instances of $\mathsf{MonadPlus}$ and $\mathsf{Alternative}$, respectively, for which the corresponding near-semiring laws hold. We also discuss some instances that do not satisfy all the laws.

4.1. Non-Determinism Applicatives From Non-Determinism Monads

It is well known that every monad determines an applicative functor. One just defines $\mathsf{pure} = \mathsf{return}$, and $(x \circledast y) = x \gg\!= \lambda h \to y \gg\!= \lambda a \to \mathsf{return}\ (h\ a)$. Likewise, every $\mathsf{MonadPlus}$ instance determines an $\mathsf{Alternative}$ instance: $\mathsf{empty} = \mathsf{mzero}$, and $(\langle|\rangle) = \mathsf{mplus}$.

If a $\mathsf{MonadPlus}$ is a non-determinism monad, i.e. the near-semiring laws for monads hold, then the near-semirings laws for applicatives hold. The applicative laws hold because every monad determines an applicative functor, the monoid laws for $\langle|\rangle$ and empty hold because they hold for mplus and mzero, and the two remaining laws are proved as follows:

$$\begin{aligned}
\mathsf{empty} \circledast x &= \mathsf{mzero} \gg\!= \lambda h \to x \gg\!= \lambda a \to \mathsf{return}\ (h\ a)\\
&= \mathsf{mzero}\\
&= \mathsf{empty}
\end{aligned}$$

$$\begin{aligned}
(f \langle|\rangle g) \circledast x &= (f\ \text{`}\mathsf{mplus}\text{`}\ g) \gg\!= \lambda h \to x \gg\!= \lambda a \to \mathsf{return}\ (h\ a)\\
&= (f \gg\!= \lambda h \to x \gg\!= \lambda a \to \mathsf{return}\ (h\ a))\\
&\qquad \text{`}\mathsf{mplus}\text{`}\ (g \gg\!= \lambda h \to x \gg\!= \lambda a \to \mathsf{return}\ (h\ a))\\
&= (f \circledast x)\ \text{`}\mathsf{mplus}\text{`}\ (g \circledast x)\\
&= (f \circledast x) \langle|\rangle (g \circledast x)
\end{aligned}$$

4.2. Non-Determinism From Lists

In formal semantics, non-determinism is often represented with a powerset monad, with union of sets as addition and the empty set as zero. However, when writing code, lists provide the most common example of a non-determinism monad.

```
instance Monad [] where
  return x = [x]
  []       ≫= f = []
  (x : xs) ≫= f = f x ++ (xs ≫= f)
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

As described in Section 4.1, this non-determinism monad determines a non-determinism applicative. However, it is not the only non-determinism applicative for lists.

Ziplists. Additionally to the instance derived from monads, lists have another instance of Applicative. This is a typical example of an applicative functor that is not derived from a Monad instance.[2]

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

Perhaps surprisingly, ZipLists have an Alternative instance which obeys the near-semiring laws, making ZipLists a non-determinism applicative.

```
instance Alternative ZipList where
  empty = ZL []
  ZL []       ⟨|⟩ ZL ys       = ZL ys
  ZL xs       ⟨|⟩ ZL []       = ZL xs
  ZL (x : xs) ⟨|⟩ ZL (y : ys) = ZL (x : rs)
    where ZL rs = ZL xs ⟨|⟩ ZL ys
```

The instance has a left bias: at each position in the resulting list, it chooses the corresponding element from the first argument, and only returns an element from the second argument if there are no elements at that position in the first argument.

4.3. The Case of Maybe

The datatype Maybe is a well-known monad. Moreover, the standard library of Haskell provides a MonadPlus instance for it.

[2]Remember that zipWith truncates to the length of the shorter list argument.

```
instance Monad Maybe where
  return x         = Just x
  Nothing >>= f    = Nothing
  Just x  >>= f    = f x
instance MonadPlus Maybe where
  mempty = Nothing
  mplus Nothing  y  = y
  mplus (Just v) y  = Just v
```

However, one of the five near-semiring axioms fails to hold. Indeed, left distribution does not hold for Maybe, as can be verified by instantiating $m_1 =$ Just Nothing and $m_2 =$ Just (Just False) in equation (5).

The monad Maybe belongs to an interesting class of instances of MonadPlus that satisfy the *left catch* law[3] rather than left distribution:

$$\mathsf{return}\ a\ \text{`}\mathsf{mplus}\text{`}\ b = \mathsf{return}\ a \tag{21}$$

A difference between the left distribution and left catch laws is that the first relates mplus with join, while the latter relates mplus with return. Left catch is related to an algebraic structure called *dioids* [7, 24], but in this article we only study near-semiring structures.

Maybe *is a Non-Determinism Applicative.* Although Maybe is not a non-determinism monad, it is a non-determinism applicative:

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

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

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

This example shows an applicative derived from a monad, which extends to a non-determinism applicative but, interestingly, it does not extend to a non-determinism monad.

4.4. Parsers

Parsers usually encompass a form of non-determinism to account for the multiple ways in which a given string can be parsed.

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

The m parameter accounts for the non-determinism effect. Thus, a Parser m a takes an input string and effectfully returns an answer consisting of a pair of a parsed a, obtained by consuming some prefix of the input, and the rest of the input which was not consumed.

A Parser m is a Monad whenever m is a Monad, and is a MonadPlus whenever m is a MonadPlus.

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

Moreover, a Parser m is a non-determinism monad whenever m is a non-determinism monad. In particular, if we instantiate m to lists, we obtain Hutton & Meijer's monadic parsers [12]. As lists are a non-determinism monad, we conclude that Hutton & Meijer's parsers are a non-determinism monad.

Non-Example: Maybe Parsers [29]. If we consider instead Parser Maybe, we do not obtain a non-determinism monad. This is expected as Maybe is not a non-determinism monad. Surprisingly, although Maybe is a non-determinism applicative, Parser Maybe is not a non-determinism applicative. In order to show that this is the case, consider the following parser char for characters, and the parsers p_1 and p_2:

```
char :: Char → Parser Maybe ()
char d = Parser (λs → case s of
                              ""      → Nothing
                              (c : s') → if c ≡ d then Just ((), s')
                                                  else Nothing)
p₁ = (char 'a' ⟨|⟩ pure ()) ∗ char 'a'
p₂ = (char 'a' ∗ char 'a') ⟨|⟩ (pure () ∗ char 'a')
```

where $*$ is a variant of $\circledast$ that ignores the computed values.

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

According to the distributive law, p_1 and p_2 should be equivalent. However, they are in fact distinct.

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

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

4.5. Composition of Alternative and Applicative

Composition of applicative functors is an applicative functor. Therefore, if f and g are applicative their composition $f \circ g$ is also applicative.

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

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

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

Note that g is only required to be an $\mathsf{Applicative}$. Moreover, if f is a non-determinism applicative and g an applicative, then $f \circ g$ is also a non-determinism applicative. For example, one can extend an arbitrary applicative functor f by composing it with the list functor and obtain a non-determinism applicative $([\,] \circ f)$. Sections 7 and 8 provide other ways to extend an applicative to an alternative.

The opposite case, where g is a non-determinism applicative, but f is not, does not necessarily yields a non-determinism applicative. In particular, for the following broken instance

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

we have that $\mathsf{empty}\ \langle | \rangle\ y = \mathsf{const\ empty}\ \langle \$ \rangle\ y$ which is generally distinct from empty, and therefore law (7) does not hold.

5. Free Constructions

Free constructions are interesting from a programming point of view since they yield a concrete representation of the programs that can be written when only the interface of the algebraic structure is known.

In this chapter we analyse how to construct the free near-semiring over a given object of a near-semiring category. In order to foster the intuition behind the construction, we start explaining the simpler cases of free monoids and free near-semirings over a set. That is, we first study the particular case of the near-semiring category Set. Then, we study the general case and obtain a general formula for the free near-semiring. This allows us to instantiate the general formula to the near-semiring categories yielding monads and applicatives, and obtain expressions for the free non-determinism monad and the free non-determinism applicative.

5.1. Free Monoids

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

Definition 5.1. *A* monoid homomorphism *from a monoid* $(M, \otimes_M, \mathsf{e}_M)$ *to a monoid* $(N, \otimes_N, \mathsf{e}_N)$ *is a function* $f : M \to N$ *such that* $f(\mathsf{e}_M) = \mathsf{e}_N$ *and* $f(m \otimes_M m') = f(m) \otimes_N f(m')$.

Now we can define the notion of free monoid.

Definition 5.2. *The* free monoid *over a set X is a monoid $(X^*, \otimes_*, \mathsf{e}_*)$ together with a function $\mathsf{inj} : X \to X^*$ such that for every monoid $(M, \otimes_M, \mathsf{e}_M)$ and function $h : X \to M$, there exists a unique monoid homomorphism $\mathsf{univ}(h) : X^* \to M$ such that $\mathsf{univ}(h) \circ \mathsf{inj} = h$.*

A concrete representation for the free monoid over a set X are lists of elements of that set; concatenation is its multiplication and the empty list is its unit. The function inj constructs a singleton list and $\mathsf{univ}(h)$ reduces the list after mapping h over it.

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

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

$$\mathsf{univ}(\mathsf{id}_M) : M^* \to M$$

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

5.2. Free Near-Semirings

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

Definition 5.3. *A* near-semiring homomorphism *from a given near-semiring $(M, \otimes_M, \mathsf{e}_M, \oplus_M, \mathsf{z}_M)$ to a near-semiring $(N, \otimes_N, \mathsf{e}_N, \oplus_N, \mathsf{z}_N)$ is a function $f : M \to N$ such that:*

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

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

Definition 5.4. *The free near-semiring over a set A is a near-semiring A^* together with a map* $\mathsf{inj} : A \to A^*$ *satisfying that for every near-semiring* $(N, \oplus, \mathsf{e}, \otimes, \mathsf{z})$ *and function* $h : A \to N$*, there exists a unique near-semiring homomorphism* $\mathsf{univ}(h) : A^* \to N$ *such that* $\mathsf{univ}(h) \circ \mathsf{inj} = h$*.*

Diagrammatically, we have the following commuting diagram:

$$\begin{tikzcd} A \arrow[r, "\mathsf{inj}"] \arrow[dr, "h"'] & A^* \arrow[d, dashed, "\mathsf{univ}(h)"] \\ & \mathsf{N} \end{tikzcd}$$

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

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

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

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

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

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

5.3. Free Near-semiring on a Near-semiring Category

In order to construct the generalised free near-semiring, we require the underlying category to have coproducts $- + - : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, and also that both $- \times -$ and $- \otimes -$ are *closed.*

Closed Bifunctors. A bifunctor $- \odot -$ is closed if there exists a bifunctor $- \overset{\odot}{\Rightarrow} - : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ together with an isomorphism:

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

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

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

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

$$\begin{aligned} \Delta_\odot &: (A + B) \odot C \to A \odot C + B \odot C \\ \Delta_\odot &= \lceil \, [\, \lfloor \mathsf{inl} \rfloor_\odot , \lfloor \mathsf{inr} \rfloor_\odot \,] \, \rceil_\odot \end{aligned}$$

where $[\_, \_]$ is case analysis on a coproduct. The distributive law is built by using the isomorphism to get rid of the bifunctor in the domain, acting on the result, and then using the isomorphism again to reinstate the bifunctor. This is a common use of closure, and more generally, of adjunctions.

In particular, for the case of $\otimes$ being a closed bifunctor, we have the following distributions:

$$\begin{aligned} \Delta_\otimes &: (A + B) \otimes C \to A \otimes C + B \otimes C \\ \kappa &: 1 \otimes A \to 1 \\ \delta &: (A \times B) \otimes C \to (A \otimes C) \times (B \otimes C) \end{aligned}$$

That is, in a setting where the additive structure is Cartesian and where the multiplicative bifunctor is closed, such a bifunctor distributes with respect to the coproduct, unit and product.

Lists are defined as $\mathsf{List}(A) = \mu X.\, 1 + A \times X$, where the initial algebra is $[\mathsf{nil}, \mathsf{cons}]$, and the universal morphism for the initial algebra is given by $\mathsf{foldr} : (1 + A \times X \to X) \to \mathsf{List}(A) \to X$. Lists are essentially sums of products, and therefore we obtain a distributivity of $\otimes$ over lists:

$$
\begin{aligned}
&\mathsf{distList} : \mathsf{List}(A) \otimes X \to \mathsf{List}(A \otimes X)\\
&\mathsf{distList} = \left\lceil \mathsf{foldr}(\lfloor k \rfloor_{\otimes}) \right\rceil_{\otimes}\\
&\qquad\qquad \text{where} \quad k \;\; : (1 + A \times (X \overset{\otimes}{\Rightarrow} \mathsf{List}(A \otimes X))) \otimes X \to \mathsf{List}(A \otimes X)\\
&\qquad\qquad\qquad\quad\;\; k \;\; = (\mathsf{nil} \circ \kappa + \mathsf{cons} \circ (\mathsf{id} \times \mathsf{ev}_{\otimes}) \circ \delta) \circ \Delta_{\otimes}
\end{aligned}
$$

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

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

The near-semiring operations for the generalised free near-semiring are similar in essence to those defined for the ordinary free near-semiring, but slightly more complex due to their abstract nature.

The initial algebra is given by

$$\begin{tikzcd}
\mathsf{List}(I + A \otimes A^*) \arrow[r, bend left, "\mathsf{in}"] \arrow[r, phantom, "\cong"] & A^* \arrow[l, bend left, "\mathsf{out}"]
\end{tikzcd}$$

The additive structure is given by:

$$
\begin{aligned}
&a : A^* \times A^* \to A^*\\
&a = \mathsf{in} \circ \mathsf{append} \circ (\mathsf{out} \times \mathsf{out})\\
&z : 1 \to A^*\\
&z = \mathsf{in} \circ \mathsf{nil}
\end{aligned}
$$

where $\mathsf{append} : \mathsf{List}(X) \times \mathsf{List}(X) \to \mathsf{List}(X)$ appends one list to another.

The multiplicative structure is more involved. The unit is simple enough:

$$
\begin{aligned}
&e : I_{\otimes} \to A^*\\
&e = \mathsf{in} \circ \mathsf{wrap} \circ \mathsf{inl}
\end{aligned}
$$

where $\mathsf{wrap} : X \to \mathsf{List}(X)$ creates a singleton list.

The multiplication is defined as follows:

$$\begin{aligned}
&m : A^* \otimes A^* \to A^* \\
&m = \mathsf{in} \circ \mathsf{flatten} \circ \mathsf{List}(f) \circ \mathsf{distList} \circ (\mathsf{out} \otimes \mathsf{id}) \\
&\quad \text{where} \;\; f \;\; : (I + A \otimes A^*) \otimes A^* \to \mathsf{List}(I + A \otimes A^*) \\
&\qquad\qquad\;\, f \;\; = [\mathsf{out} \circ \lambda, \mathsf{wrap} \circ \mathsf{inr} \circ (\mathsf{id} \otimes m) \circ \alpha^{-1}] \circ \Delta_{\otimes}
\end{aligned}$$

and $\mathsf{flatten} : \mathsf{List}(\mathsf{List}(X)) \to \mathsf{List}(X)$ flattens a lists of lists.

The near-semiring we have just defined is indeed the free one. The universal property for the free near-semiring states that given a near-semiring N and morphism $h : A \to N$, there is a unique univ that makes the universal diagram commute:

$$\begin{tikzcd}
A \arrow[r, "\mathsf{inj}"] \arrow[dr, "h"'] & A^* \arrow[d, dashed, "\mathsf{univ}(h)"] \\
& N
\end{tikzcd}$$

Let $\mathsf{inj} = \mathsf{in} \circ \mathsf{wrap} \circ \mathsf{inr} \circ (\mathsf{id} \otimes e) \circ \rho_{\otimes}^{-1}$. The induced universal morphism $\mathsf{univ}(h) : A^* \to N$ is defined by structural recursion:

$$\mathsf{univ}(h) = A^* \cong \mathsf{List}(I + A \otimes A^*) \xrightarrow{\mathsf{List}([e, m \circ (h \otimes \mathsf{univ}(h))])} \mathsf{List}(N) \xrightarrow{\mathsf{foldr}([z,a])} N$$

These definitions are highly abstract; the following subsections provide concrete instances of this general construction.

5.4. The Free Non-Determinism Monad

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

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

$$\begin{aligned}
&\textbf{data}\ \mathsf{Free}_{\circ}\, f\ x = \mathsf{Free}_{\circ}\ \{\mathsf{unFree}_{\circ} :: [\mathsf{FFree}_{\circ}\, f\ x]\} \\
&\textbf{data}\ \mathsf{FFree}_{\circ}\, f\ x = \mathsf{Pure}_{\circ}\ x \mid \mathsf{Con}_{\circ}\ (f\ (\mathsf{Free}_{\circ}\, f\ x))
\end{aligned}$$

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

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

The function inj that embeds values of the original functor in the free structure and the universal morphism univ h that uniquely maps the free structure onto another near-semiring are obtained by direct instantiation of the corresponding general constructions in the particular near-semiring category we are currently considering.

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

5.5. The Free Non-determinism Applicative Functor

For obtaining the free non-determinism applicative functor we now work in the near-semiring category of endofunctors with Day convolution as multiplicative structure. Based on the generic recipe we obtain the following datatype:

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

The supporting definitions of the operations are as follows:

```
instance Functor f ⇒ Applicative (Free⋆ f) where
  pure x = Free⋆ [Pure⋆ x]
  Free⋆ xs ⊛ v = Free⋆ (concatMap g xs)
    where g (Pure⋆ f)   = unFree⋆ (fmap f v)
          g (Con⋆ f c) = [Con⋆ (fmap uncurry f) (pure (,) ⊛ c ⊛ v)]
```

```
instance Functor f ⇒ Alternative (Free⋆ f) where
   empty = Free⋆ []
   Free⋆ xs ⟨|⟩ Free⋆ ys = Free⋆ (xs ++ ys)
```

The injection and the family of unique homomorphisms are:

```
inj :: Functor f ⇒ f a → Free⋆ f a
inj x = Free⋆ [Con⋆ (fmap (λz () → z) x) (pure ())]
univ :: Alternative g ⇒ (∀x. f x → g x) → Free⋆ f x → g x
univ h (Free⋆ l) = foldr (⟨|⟩) empty (map univ_T l)
                   where univ_T (Pure⋆ x)  = pure x
                         univ_T (Con⋆ q c) = h q ⊛ univ h c
```

6. Cayley Representations

6.1. Cayley Representation of a monoid

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

$$\begin{tikzcd}
M^* \arrow[r, "\mathsf{rep}^*"] \arrow[d, "\mathsf{univ}(\mathsf{id}_M)"'] & R(M)^* \arrow[d, "\mathsf{univ}(\mathsf{id}_{R(M)})"] \\
M & R(M) \arrow[l, "\mathsf{abs}"]
\end{tikzcd}$$

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

The change of representation that Cayley provides can result in more efficient implementations [11].

The *monoid of endomorphisms* over a set X is $(X \to X, \circ, \mathsf{id})$, where $\circ$ is function composition and id is the identity function. Every monoid has an embedding into the monoid of endomorphisms over its carrier set, a result usually known as Cayley's theorem for monoids [25].

Theorem 6.1 (Cayley for (Set) monoids). *Every monoid $(M, \otimes, \mathsf{e})$ embeds into the monoid of endomorphisms over the set M, namely $(M \to M, \circ, \mathsf{id})$.*

The embedding is given by the monoid morphism $\mathsf{rep} : M \to (M \to M)$ *and function* $\mathsf{abs} : (M \to M) \to M$

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

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

A simple consequence of this theorem is the following

Corollary 6.2. *The monoid of endomorphisms* $(M \to M, \circ, \mathsf{id})$ *is a representation of the monoid* $(M, \otimes, \mathsf{e})$*, with* rep *and* abs *as in the theorem above.*

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

$$\begin{tikzcd} M^* \arrow[r, "\mathsf{rep}^*"] \arrow[d, "\mathsf{univ}(\mathsf{id}_M)"'] & (M \to M)^* \arrow[d, "\mathsf{univ}(\mathsf{id}_{M \to M})"] \\ M \arrow[r, "\mathsf{rep}"'] & M \to M \end{tikzcd}$$

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

6.2. Cayley Representation of Near-semirings

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

$$\begin{tikzcd} N^* \arrow[r, "\mathsf{rep}^*"] \arrow[d, "\mathsf{univ}(\mathsf{id}_N)"'] & R(N)^* \arrow[d, "\mathsf{univ}(\mathsf{id}_{R(N)})"] \\ N & R(N) \arrow[l, "\mathsf{abs}"] \end{tikzcd}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and the representation and abstraction functions are:

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

Unfortunately, the type $(N \overset{\bullet}{\rightarrow} N) \rightarrow (N \overset{\bullet}{\rightarrow} N)$ is not usually available in programming languages due to the side conditions on the type constructor $\overset{\bullet}{\rightarrow}$. One has to compromise and use the type $DC(N) = (N \rightarrow N) \rightarrow (N \rightarrow N)$, for which we obtain the semiring of endomorphisms over endomorphisms $(DC(N), \circ, \mathsf{id}, \oplus', \mathsf{z}')$ as in the following pseudo-Haskell implementation:[4]

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

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

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

[4]Actual Haskell code requires DC to be defined as a newtype and involves additional clutter due to explicit newtype wrapping and unwrapping.

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

Theorem 6.3. *Let* $(N, \otimes, \mathsf{e}, \oplus, \mathsf{z})$ *be a near-semiring. Then the near-semiring* $(DC(N), \circ, \mathsf{id}, \oplus', \mathsf{z}')$ *is a representation of* N*. That is, the following diagram commutes.*

$$\begin{tikzcd}
N^* \arrow[r, "\mathsf{rep}^*"] \arrow[d, "\mathsf{univ}(\mathsf{id}_N)"'] & DC(N)^* \arrow[d, "\mathsf{univ}(\mathsf{id}_{DC(N)})"] \\
N & DC(N) \arrow[l, "\mathsf{abs}"]
\end{tikzcd}$$

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

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

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

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

6.3. Double Cayley Representation in a Near-semiring Category

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

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

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

The definition of the generalised double Cayley representation is given in Figure 1. When specialised to the case of the near-semiring category Set with Cartesian products as multiplicative and additive structure, we recover the double Cayley representation of Section 6.2.

Near-semiring operations:

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

Representation and abstraction functions:

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

Figure 1: Double Cayley Representation

6.4. Double Cayley Representation for Monads

We now instantiate the generalised definition of the double Cayley representation for monads. First, though, we need to define the closures for the two bifunctors involved.

Closures. The multiplicative bifunctor is the composition of functors. Its closure is given by the so-called right Kan extension:

$$\textbf{newtype}\ (\overset{\circ}{\Rightarrow})\ f\ g\ x = \mathsf{Ran}\ \{\mathsf{unRan} :: \forall y.\, (x \to f\ y) \to g\ y\}$$

The additive bifunctor is the Cartesian product of functors and its closure is as follows:

$$\textbf{newtype}\ (\overset{\times}{\Rightarrow})\ f\ g\ x = \mathsf{Exp}\ \{\mathsf{unExp} :: \forall y.\, (x \to y) \to (f\ y \to g\ y)\}$$

The corresponding isomorphisms for the closures are given in Figure 2 and Figure 3, respectively.

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

$$\textbf{newtype}\ \mathsf{DC}\, f\ x = \mathsf{DC}\ \{\mathsf{unDC} :: ((f \overset{\times}{\Rightarrow} f) \overset{\circ}{\Rightarrow} (f \overset{\times}{\Rightarrow} f))\ x\}$$

$$\begin{array}{l}
\textbf{instance}\ \mathsf{Functor}\ (g \overset{\circ}{\Rightarrow} h)\ \textbf{where} \\
\quad \mathsf{fmap}\ f\ m = \mathsf{Ran}\ (\lambda k \to \mathsf{unRan}\ m\ (k \circ f)) \\
\lfloor\cdot\rfloor_{\circ} :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g, \mathsf{Functor}\ h) \Rightarrow \\
\qquad (\forall x. f\ (g\ x) \to h\ x) \to (\forall x. f\ x \to (g \overset{\circ}{\Rightarrow} h)\ x) \\
\lfloor f\rfloor_{\circ} = \lambda gx \to \mathsf{Ran}\ (\lambda k \to f\ (\mathsf{fmap}\ k\ gx)) \\
\lceil\cdot\rceil_{\circ} :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g, \mathsf{Functor}\ h) \Rightarrow \\
\qquad (\forall x. f\ x \to (g \overset{\circ}{\Rightarrow} h)\ x) \to (\forall x. f\ (g\ x) \to h\ x) \\
\lceil f\rceil_{\circ} = \lambda fgx \to \mathsf{unRan}\ (f\ fgx)\ \mathsf{id}
\end{array}$$

Figure 2: Closure of functor composition

$$\begin{array}{l}
\textbf{instance}\ \mathsf{Functor}\ (f \overset{\times}{\Rightarrow} g)\ \textbf{where} \\
\quad \mathsf{fmap}\ f\ m = \mathsf{Exp}\ (\lambda k \to \mathsf{unExp}\ m\ (k \circ f)) \\
\lfloor\cdot\rfloor_{\times} :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g, \mathsf{Functor}\ h) \Rightarrow \\
\qquad (\forall x. (f\ x, g\ x) \to h\ x) \to (\forall x. f\ x \to (g \overset{\times}{\Rightarrow} h)\ x) \\
\lfloor f\rfloor_{\times} = \lambda fx \to \mathsf{Exp}\ (\lambda t \to \lambda gy \to f\ (\mathsf{fmap}\ t\ fx, gy)) \\
\lceil\cdot\rceil_{\times} :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g, \mathsf{Functor}\ h) \Rightarrow \\
\qquad (\forall x. f\ x \to (g \overset{\times}{\Rightarrow} h)\ x) \to (\forall x. (f\ x, g\ x) \to h\ x) \\
\lceil f\rceil_{\times} = \lambda fgx \to \mathsf{unExp}\ (f\ (\mathsf{fst}\ fgx))\ \mathsf{id}\ (\mathsf{snd}\ fgx)
\end{array}$$

Figure 3: Closure of product of functors

$$\begin{array}{l}
\mathsf{proj}\ (\mathsf{DC}\ (\mathsf{Ran}\ a)) = a \\
\textbf{instance}\ \mathsf{Monad}\ (\mathsf{DC}\ f)\ \textbf{where} \\
\quad \mathsf{return}\ x = \mathsf{DC}\ (\mathsf{Ran}\ (\lambda f \to f\ x)) \\
\quad \mathsf{DC}\ (\mathsf{Ran}\ m) \ggg\!= f = \mathsf{DC}\ (\mathsf{Ran}\ (\lambda g \to m\ (\lambda a \to \mathsf{proj}\ (f\ a)\ g))) \\
\textbf{instance}\ \mathsf{MonadPlus}\ (\mathsf{DC}\ f)\ \textbf{where} \\
\quad \mathsf{mzero} = \mathsf{DC}\ (\mathsf{Ran}\ (\lambda k \to \mathsf{Exp}\ (\lambda c\ x \to x))) \\
\quad \mathsf{mplus}\ m\ n = \mathsf{DC}\ (\mathsf{Ran}\ (\lambda sk \to \\
\qquad \mathsf{Exp}\ (\lambda f\ fk \to \mathsf{unExp}\ (\mathsf{proj}\ m\ sk)\ f\ (\mathsf{unExp}\ (\mathsf{proj}\ n\ sk)\ f\ fk))))
\end{array}$$

Figure 4: Operations of the double Cayley Monad

Figure 4 defines the associated near-semiring operations.

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

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

Example. Consider the anyof function which non-deterministically chooses an element from the given list.

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

When m is the list monad, anyof essentially reverses the given list. Due to the left-nested recursion, it has a quadratic time complexity.

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

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

Here, abs forces the type m in anyof to be the double Cayley representation over lists. The complexity of the new implementation is linear in the length of the input list.

6.5. The Double Cayley Representation for Applicatives

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

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

$$\begin{array}{l}
\textbf{instance}\ \mathsf{Functor}\ g \Rightarrow \mathsf{Functor}\ (f \overset{\star}{\Rightarrow} g)\ \textbf{where} \\
\quad \mathsf{fmap}\ f\ (\mathsf{ED}\ g) = \mathsf{ED}\ (\lambda y \to \mathsf{fmap}\ (\lambda(a,b) \to (f\ a, b))\ (g\ y)) \\
\lfloor\cdot\rfloor_{\star} :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g, \mathsf{Functor}\ h) \Rightarrow \\
\qquad\quad (\forall x.\,(f \star g)\ x \to h\ x) \to (\forall x.\, f\ x \to (g \overset{\star}{\Rightarrow} h)\ x) \\
\lfloor f\rfloor_{\star} = \lambda \mathit{fx} \to \mathsf{ED}\ (\lambda \mathit{gy} \to f\ (\mathsf{Day}\ (\mathsf{fmap}\ (,)\ \mathit{fx})\ \mathit{gy})) \\
\lceil\cdot\rceil_{\star} :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g, \mathsf{Functor}\ h) \Rightarrow \\
\qquad\quad (\forall x.\, f\ x \to (g \overset{\star}{\Rightarrow} h)\ x) \to (\forall x.\,(f \star g)\ x \to h\ x) \\
\lceil f\rceil_{\star} = \lambda(\mathsf{Day}\ \mathit{ff}\ \ \mathit{gy}) \to \mathsf{fmap}\ (\mathsf{uncurry}\ (\$))\ (\mathsf{unED}\ (f\ \mathit{ff})\ \mathit{gy})
\end{array}$$

Figure 5: Closure of the Day convolution

The corresponding isomorphism is shown in Figure 5.

Having established the closure of the Day convolution, we define the double Cayley representation:

$$\textbf{newtype}\ \mathsf{DC}\ f\ x = \mathsf{DC}\ \{\mathsf{unDC} :: ((f \overset{\times}{\Rightarrow} f) \overset{\star}{\Rightarrow} (f \overset{\times}{\Rightarrow} f))\ x\}$$

See Figure 6 for the supporting code.

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

$$\begin{array}{l}
\mathsf{rep} :: \mathsf{Alternative}\ f \Rightarrow f\ a \to \mathsf{DC}\ f\ a \\
\mathsf{rep}\ x = \mathsf{DC}\ (\mathsf{ED}\ (\lambda g \to \mathsf{Exp}\ (\lambda f\ \mathit{fx} \to \\
\qquad\qquad\qquad\qquad \mathsf{unExp}\ g\ (\mathsf{flip}\ (\mathsf{curry}\ f))\ \mathsf{empty} \circledast x\ \langle|\rangle\ \mathit{fx}))) \\
\mathsf{abs} :: \mathsf{Alternative}\ f \Rightarrow \mathsf{DC}\ f\ a \to f\ a \\
\mathsf{abs}\ (\mathsf{DC}\ (\mathsf{ED}\ f)) = \mathsf{unExp}\ (f\ (\mathsf{Exp}\ (\lambda g\ i \to \mathsf{pure}\ (g\ ())\ \langle|\rangle\ i)))\ \mathsf{fst}\ \mathsf{empty}
\end{array}$$

7. Constructing Near-semirings from Monoids

If an object already has a monoid structure, we may want to reuse it and add only the missing near-semiring structure. This is not what the free near-semiring construction of Section 5 does, as it starts from any object and entirely ignores any associated algebraic structure.

However, there is a well-known approach for monads that accomplishes the task. This section investigates and generalises that approach.

```
instance Functor f ⇒ Functor (DC f) where
  fmap f (DC z) = DC (ED (λfx →
    fmap (λ(x, y) → (f x, y)) (unED z fx)))
instance Functor f ⇒ Applicative (DC f) where
  pure v = DC (ED (λf → fmap (λy → (v, y)) f))
  (DC (ED h)) ⊛ (DC (ED v)) = fmap (uncurry ($))
    (DC (ED (λf → fmap (λ(b, (a, y)) → ((a, b), y)) (v (h f)))))
instance Functor f ⇒ Alternative (DC f) where
  empty = DC (ED (const (Exp (λh x → x))))
  DC f ⟨|⟩ DC g = DC (ED (λh →
    Exp (λj i → unExp (unED f h) j (unExp (unED g h) j i))))
```

Figure 6: Supporting code for the double Cayley Alternative

7.1. The ListT *Monad Transformer*

The ListT monad transformer [14] is a construction[5] that extends any monad M to a MonadPlus instance. It can be defined in Haskell as the following datatype [23, 15, 22]:[6]

```
newtype ListT m a = ListT {unListT :: m (Maybe (a, ListT m a))}
```

If m is a monad, then so is ListT m.

```
instance Monad m ⇒ Monad (ListT m) where
  return x = ListT (return (Just (x, mzero)))
  p ≫= k  = ListT (do r ← unListT p
                      case r of
                        Nothing     → return Nothing
                        Just (x, p′) → unListT (mplus (k x) (p′ ≫= k)))
```

Moreover, we can lift any computation from m to ListT m

```
instance Trans ListT where
  lift p = ListT (p ≫= λx → return (Just (x, mzero)))
```

[5]Not to be confused with the type $M \circ [\,]$, which is not a proper monad transformer.
[6]Hinze [8] and Kiselyov et al. [16] present CPS-based variants of this definition.

in a way that preserves the monad structure, i.e., lift is a monad morphism and hence satisfies:

$$\begin{aligned} \mathsf{lift}\ (\mathsf{return}\ x) &= \mathsf{return}\ x \\ \mathsf{lift}\ (p \gg\!= k) &= \mathsf{lift}\ p \gg\!= \mathsf{lift} \circ k \end{aligned}$$

Finally, the whole point of ListT is to extend the monad m with non-determinism.

```
instance Monad m ⇒ MonadPlus (ListT m) where
  mzero      = ListT (return Nothing)
  mplus p q = ListT (do r ← unListT p
                        case r of
                          Nothing     → unListT q
                          Just (x, p′) → return (Just (x, mplus p′ q)))
```

The transformer comes equipped with a runListT function that sequentialises all the computations in the underlying monad and returns the list of all solutions.

```
runListT :: Monad m ⇒ ListT m a → m [a]
runListT p = do r ← unListT p
               case r of
                 Nothing → return []
                 Just (x, p′) → runListT p′ ≫= return ∘ (x:)
```

Generalising ListT. It is not difficult to see that ListT can be written in terms of the near-semiring building blocks:

$$\mathsf{ListT} M = \mu X.\, M \circ (1 + (\mathsf{Id} \times X)) \tag{22}$$

This formula can be used for generalising the list monoid transformer to other near-semiring instances. We next consider the instance for applicative functors.

7.2. The ListA *Applicative Transformer*

Just like we extended monads above to MonadPlus instances, we can extend applicative functors to Alternative instances by replacing functor composition for Day convolution in formula (22): $\mathsf{ListA}\ \mathsf{F} = \mu X.\, \mathsf{F} \star (1 + \mathsf{Id} \times \mathsf{X})$.

This yields an *applicative transformer* ListA, which in Haskell can be implemented as:

```
data ListA f a = ∀b. f (b → a) ⊞ Maybe (b, ListA f b)
```

This type turns any applicative functor f into a new applicative functor ListA f,

```
instance Applicative f ⇒ Applicative (ListA f) where
  pure x  = pure (const x) ⊞ Just ((), empty)
  (op_x ⊞ Nothing)      ⊛ ys               = fmap uncurry op_x ⊞ Nothing
  (op_x ⊞ Just (x, xs)) ⊛ ys@(op_y ⊞ r_y) = op ⊞ r
    where
      op = ((λf g → either (f x ∘ g) (uncurry f)) ⟨$⟩ op_x) ⊛ op_y
      r  = fmap (λ(y, ys′) → (Left y, fmap Left ys′ ⟨|⟩
                                   ((curry Right ⟨$⟩ xs) ⊛ ys))) r_y
```

in such a way that the underlying Applicative structure is preserved. Indeed, if we formulate a new type class ATrans for applicative transformers,

```
class ATrans t where
  alift :: Applicative f ⇒ f a → t f a
```

such that alift is an Applicative homomorphism, i.e.,

$$\begin{aligned} \mathsf{alift}\ (\mathsf{pure}\ x) &= \mathsf{pure}\ x \\ \mathsf{alift}\ (f \circledast x) &= \mathsf{alift}\ f \circledast \mathsf{alift}\ x \end{aligned}$$

then ListA is an instance as follows:

```
instance ATrans ListA where
  alift p = fmap const p ⊞ Just ((), empty)
```

Of course, ListA f is also a non-determinism applicative functor.

```
instance Applicative f ⇒ Alternative (ListA f) where
  empty = pure id ⊞ Nothing
  (op ⊞ Nothing)      ⟨|⟩ (op′ ⊞ r) = ((λx y → y) ⟨$⟩ op ⊛ op′) ⊞ r
  (op ⊞ Just (x, xs)) ⟨|⟩ ys         =
    fmap (flip either id) op ⊞ Just (Left x, fmap Left xs ⟨|⟩ fmap Right ys)
```

Just like for monads, we can provide a runListA function to interpret the ListA f structure into f and collect all the solutions in a list.

```
runListA :: Applicative f ⇒ ListA f a → f [a]
runListA (op ⊞ Nothing)     = const [] ⟨$⟩ op
runListA (op ⊞ Just (x, xs)) =
  (λf xs → map f (x : xs)) ⟨$⟩ op ⊛ runListA xs
```

Note that just like with ListT, ListA does not allow us to skip any preceding effects before obtaining the ith solution.[7]

Comparison with the List Alternative. As we saw earlier in Section 4.5, composition with the list functor is another way in which we can add non-determinism to an applicative functor. Moreover, we can also inject $f\ a$ values into $([] \circ f)\ a$ in a way that preserves the applicative structure.

```
instance ATrans ((∘) []) where
  alift p = Comp [p]
```

In addition, we can also run the extended computation to collect all the results in the underlying applicative functor f.

```
runL :: Applicative f ⇒ ([] ∘ f) a → f [a]
runL (Comp ps) = foldr (λx xs → ((:) ⟨$⟩ x) ⊛ xs) (pure []) ps
```

In a sense this representation offers more flexibility than ListA as we can skip some solutions without incurring their effects.

```
dropL :: Applicative f ⇒ Int → ([] ∘ f) a → f [a]
dropL n (Comp ps) = runL (Comp (drop n ps))
```

However this flexibility comes at the cost of sometimes repeating earlier effects. The following example illustrates this behaviour.

```
> runListA (alift (Const [1]) ⊛ (alift (Const [2]) ⟨|⟩ alift (Const [3])))
Const [1, 2, 3]
```

[7]However, unlike ListT, ListA curiously allows us to skip preceding effects when selecting the ith effect.

```
> runL      (alift (Const [1]) ⊛ (alift (Const [2]) ⟨|⟩ alift (Const [3])))
Const [1, 2, 1, 3]
```

The latter result is explained by the fact that $\circledast$ distributes over $\langle | \rangle$ on both sides for $[\,] \circ f$. Another difference is that empty not only is a left zero, but also is a right zero of $\circledast$; this means that some effects are discarded by $[\,] \circ f$.

```
> runListA (alift (Const [1]) ⊛ empty)
Const [1]
> runL      (alift (Const [1]) ⊛ empty)
Const []
```

In summary, neither $[\,] \circ f$ nor $\mathsf{ListA}\ f$ is more general than the other, and the choice of one or the other depends on which behaviour we would like to obtain. This naturally leads to the question: what is the most general (i.e., free) construction that transforms a monoid into a near-semiring while preserving the monoidal structure? We provide an answer in the next section.

8. Near-semirings from Unital Monoids

This section explores another way to construct a near-semiring from a monoid: the *free* near-semiring on a monoid. Instead of working in full generality, we only present the specific construction for monads, as the inspiration for this work arose from prior work on the coproduct of monads.

Coproduct of Monads. The *coproduct* of two monads is the most general monad that supports the operations of both monads. Unfortunately, this coproduct cannot be defined constructively for all monads, and, even when it can be, this construction might be difficult to implement in a programming language. Yet, Uustalu and Ghani [6] have shown how to construct and implement the coproduct of two monads, if those monads are in the restricted class of *ideal* monads. An ideal monad is a monad with a distinguished unit, which Uustalu and Ghani formalise as a monad (M, μ, η) for which M can be decomposed as $\mathsf{Id} + M_0$ for some functor M_0, and $\mu = [\mathsf{id}, \mathsf{inr} \circ \mu_0]$ for some natural transformation $\mu_0 : M_0 \circ (\mathsf{Id} + M_0) \to M_0$. In this article, we use a generalisation, which we call *unital* monads, that allows cancellation of effects.

8.1. Unital Monads

Conceptually, a unital monad is a monad in which it is possible to determine whether a given element is in the image of return. We characterise a unital monad $(\mathsf{Id}+M_0, \mathsf{inl}, \mu)$ in terms of its M_0 endofunctor and a morphism $\mu_0 : M_0 \circ (\mathsf{Id}+M_0) \to \mathsf{Id}+M_0$, which makes the following diagram commute.

$$\begin{tikzcd}
M_0 \circ M_0 \circ (\mathsf{Id}+M_0) \arrow[r, "\mathsf{id}\circ\mathsf{inr}\circ\mathsf{id}"] \arrow[dd, "\mathsf{id}\circ\mu_0"'] & M_0 \circ (\mathsf{Id}+M_0) \circ (\mathsf{Id}+M_0) \arrow[d, "\mu_0\circ\mathsf{id}"] \\
 & (\mathsf{Id}+M_0) \circ (\mathsf{Id}+M0) \arrow[d, "\mu"] \\
M_0 \circ (\mathsf{Id}+M_0) \arrow[r, "\mu_0"'] & \mathsf{Id}+M_0 \\
M_0 \circ \mathsf{Id} \arrow[u, "\mathsf{id}\circ\mathsf{inl}"] \arrow[r, equal] & M_0 \arrow[u, "\mathsf{inr}"']
\end{tikzcd}$$

where $\mu = [\mathsf{id}, \mu_0]$ is the multiplication of the monad induced by M_0.

In Haskell, we capture the type $\mathsf{Id}+M_0$ in the UM datatype

```
data UM m₀ a = Pure a | Impure (m₀ a)
```

and the μ_0 morphism in the Unital type class

```
class Functor m₀ ⇒ Unital m₀ where
  join₀ :: m₀ (UM m₀ a) → UM m₀ a
```

These induce a Monad instance.

```
instance Unital m₀ ⇒ Monad (UM m₀) where
  return x          = Pure x
  Pure x      ≫= k = k x
  Impure op   ≫= k = join₀ (fmap k op)
```

Example: The List Monad. The list monad is an example of a unital monad. This is not immediate in its well-known form $[a]$, but it is exposed in the isomorphic UM List02 a representation where List02 captures non-singleton lists, i.e., the non-unit values.

```
data List02 a = Cons0 | Cons2 a a [a]
```

The isomorphism between UM List02 a and $[a]$ is witnessed by the following two functions.

```
fromL :: [a] → UM List02 a
fromL []           = Impure Cons0
fromL [x]          = Pure x
fromL (x : y : xs) = Impure (Cons2 x y xs)
toL :: UM List02 a → [a]
toL (Pure x)                 = [x]
toL (Impure Cons0)           = []
toL (Impure (Cons2 x y xs)) = x : y : xs
```

The Unital instance for List02 is based on the concat operation for lists. We implement it by using the isomorphism presented.

```
instance Unital List02 where
  join₀ Cons0 = fromL []
  join₀ (Cons2 x y xs) = fromL (concat (toL x : toL y : map toL xs))
```

The proof that join_0 satisfies the necessary laws is routine.

Coproducts of Unital Monads. It is not difficult to see that our notion of unital monad is a generalisation of the notion of ideal monad introduced by Uustalu and Ghani [6], and that their coproduct construction can be used for unital monads. In particular, if $R = \mathsf{Id} + R_0$ and $S = \mathsf{Id} + S_0$ are two unital monads, then the carrier of their coproduct is

$$T = \mathsf{Id} + T_1 + T_2 \tag{23}$$

where T_1 and T_2 are defined as the least fixed point of the following equations

$$T_1 \;\cong\; R_0 \circ (\mathsf{Id} + T_2) \qquad\qquad T_2 \;\cong\; S_0 \circ (\mathsf{Id} + T_1)$$

8.2. The Free Near-Semiring

Let us construct a near-semiring over a monad M. In order to equip M with the MonadPlus operations mplus and mzero, we simply build its coproduct with a monad that provides these operations. Since we want to construct the smallest such near-semiring, we use the list monad, which, as the free monoid, has no extra operations. Moreover, because we use the coproduct construction, M has to be unital.

If we take $R_0 = m_0$ and $S_0 =$ List02, formula (23) explains how to implement the coproduct of the two unital monads. Its carrier functor is defined as follows:

```
data UMForest m0 a = InId a | InT1 (T1 m0 a) | InT2 (T2 m0 a)
data T1 m0 a = T1 (m0 (Either a (T2 m0 a)))
data T2 m0 a = T2 (List02 (Either a (T1 m0 a)))
```

Because it is constucted as a coproduct of monads, UMForest is a monad; its implementation can be found in Figure 7. In what follows, we focus on the MonadPlus instance, which provides the new operations to the monad UM m_0.

The coproduct injection functions let us embed values from two monads into their coproduct:

```
liftLeft :: Unital m0 ⇒ UM m0 a → UMForest m0 a
liftLeft (Pure x)      = InId x
liftLeft (Impure m0) = InT1 (T1 (fmap Left m0))

liftRight :: Unital m0 ⇒ UM List02 a → UMForest m0 a
liftRight (Pure x)      = InId x
liftRight (Impure m0) = InT2 (T2 (fmap Left m0))
```

With the aid of these functions, we can lift operations from the underlying monads to the coproduct. In particular, the MonadPlus instance is implemented by lifting the mzero and mplus operations from the list monad:

```
instance Unital m0 ⇒ MonadPlus (UMForest m0) where
  mzero          = join (liftRight mzero)
  x `mplus` y = join (liftRight (return x `mplus` return y))
```

The UMForest construction is more general than ListT. While the latter sequentialises all alternatives: we need to perform the effects of the i first alternatives to reach the ith result, this is not the case for UMForest, which allows us to skip orthogonal branches and their effects. In fact, UMForest is the most general construction:

Theorem 8.1 (Free Near-Semiring over a Monad)**.** *If* UM m_0 *is a unital monad, then* UMForest m_0 *is the free near-semiring over* UM m_0*.*

```
instance Unital m₀ ⇒ Monad (UMForest m₀) where
  return x = InId x
  InId v        >>= f = f v
  InT₁ (T₁ v) >>= f = reB₁ (join₀ (fmap unB₁ (fmap p₁ v)))
    where p₁ (Left a)    = f a
          p₁ (Right w) = InT₂ w >>= f
          unB₁ (InId a)          = Pure (Left a)
          unB₁ (InT₂ v)          = Pure (Right v)
          unB₁ (InT₁ (T₁ v)) = Impure v
          reB₁ (Pure (Left a))   = InId a
          reB₁ (Pure (Right v)) = InT₂ v
          reB₁ (Impure v)        = InT₁ (T₁ v)
  InT₂ (T₂ v) >>= f = reB₂ (join₀ (fmap unB₂ (fmap p₂ v)))
    where p₂ (Left a)    = f a
          p₂ (Right w) = InT₁ w >>= f
          unB₂ (InId a)          = Pure (Left a)
          unB₂ (InT₁ v)          = Pure (Right v)
          unB₂ (InT₂ (T₂ v)) = Impure v
          reB₂ (Pure (Left a))   = InId a
          reB₂ (Pure (Right v)) = InT₁ v
          reB₂ (Impure v)        = InT₂ (T₂ v)
```

Figure 7: Coproduct of a unital monad and the list monad

9. Applications

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

9.1. Advanced Combinatorial Search

Bunches. Spivey proposed an algebraic structure, called a *bunch*, for expressing combinatorial search strategies such as depth-first (DFS) and breadth-first (BFS) search [28]. Bunches are in fact a non-determinism monad with one additional operation:

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

In addition to the axioms of a non-determinism monad, bunches also satisfy the following axiom relating $(\gg\!=)$ and wrap:

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

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

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

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

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

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

The wrap operator for this type is expressed as follows.

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

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

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

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

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

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

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

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

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

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

By means of $\mathsf{search} \circ \mathsf{dbs}\ n$ we can combine this heuristic with any search strategy.

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

```
bench :: Int → Int
bench n = solutionCount (forest n)
forest :: Int → Forest ()
forest 0 = return ()
forest n = (forest (n − 1) >>= wrap′) ‘mplus‘ wrap′ ()
solutionCount :: Forest () → Int
solutionCount f = sum (map go f)
  where go (Leaf _) = 1
        go (Fork f) = solutionCount f
```

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

We ran the benchmark for different problem sizes using Forest both directly and indirectly through the double Cayley representation. All runs took place in the Criterion benchmarking harness using GHC 7.8.2 on a 3 GHz Intel Core i3 processor, 16 GB RAM and Ubuntu 14.04. All values are in milliseconds.

Size	Forest	DC Forest
64	2	1
128	38	7
256	480	35
512	5,491	150
1,024	51,590	572
2,048	T/O	2,580

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

9.2. Interleaving Alternative *Parsers*

Swierstra and Dijkstra [31] show how to extend Alternative parser combinators with a new combinator ($\langle||\rangle$) to *interleave* two given parsers. For example, suppose we have a parser digits for digit sequences and another parser letters for letter sequences. Then the interleaved parser (digits $\langle||\rangle$ letters) accepts strings like `"a1b45cda19"` and produces the result (`"abcda"`, 14519).

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

$$(\mathsf{pure}\ \texttt{'a'}\ \langle|\rangle\ \mathsf{pure}\ \texttt{'b'})\ \langle||\rangle\ \mathsf{pure}\ 1$$

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

Our free non-determinism applicative functor provides a cleaner slate for interleaved parsers. As a consequence, our solution neither duplicates nor drops results. Moreover, we obtain a solution that is not parser-specific, but works for any Alternative instance.

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

$$\begin{array}{lll}
(\langle||\rangle) :: \mathsf{Functor}\ f \Rightarrow \mathsf{Free}_\star\ f\ \ a \to \mathsf{Free}_\star\ f\ \ b \to \mathsf{Free}_\star\ f\ (a,b) \\
ga\ \langle||\rangle\ gb = & & \mathsf{Free}_\star\ [fa\ \text{`}\mathsf{fwdby}\text{`}\ gb \mid fa \leftarrow fas] \\
 & \langle|\rangle\ (\mathsf{swap}\ \langle\$\rangle & \mathsf{Free}_\star\ [fb\ \text{`}\mathsf{fwdby}\text{`}\ ga \mid fb \leftarrow fbs]) \\
 & \langle|\rangle & \mathsf{Free}_\star\ [\mathsf{Pure}_\star\ (a,b)\ \mid a \leftarrow as, b \leftarrow bs] \\
\quad \textbf{where}\ \mathsf{swap}\ (a,b) = (b,a) \\
 & (as, fas) \quad = \mathsf{split}\ ga \\
 & (bs, fbs) \quad = \mathsf{split}\ gb
\end{array}$$

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

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

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

$$\begin{aligned}&\mathsf{split} :: \mathsf{Free}_\star\ f\ a \to ([a], [\mathsf{FFree}_\star\ f\ a])\\ &\mathsf{split}\ (\mathsf{Free}_\star\ l) = ([a \mid \mathsf{Pure}_\star\ a \leftarrow l], [f \mid f@(\mathsf{Con}_\star\ p\ r) \leftarrow l])\end{aligned}$$

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

$$\begin{aligned}&\mathsf{fwdby} :: \mathsf{Functor}\ f \Rightarrow \mathsf{FFree}_\star\ f\ a \to \mathsf{Free}_\star\ f\ b \to \mathsf{FFree}_\star\ f\ (a, b)\\ &(\mathsf{Con}_\star\ pa\ ra)\ \text{`}\mathsf{fwdby}\text{`}\ \mathit{fbs} = \mathsf{Con}_\star\ (\mathsf{fmap\ first}\ pa)\ (ra\ \langle||\rangle\ \mathit{fbs})\\ &\qquad\qquad\qquad\qquad\qquad\mathbf{where}\ \mathsf{first}\ f\ (c, b) = (f\ c, b)\end{aligned}$$

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

10. Related Work

10.1. Codensity Monad and Cayley Representation

Cayley representations appear under different guises in the literature. Hughes uses it to optimise list concatenation [11] and Voigtländer [33] uses the codensity monad transformer to optimise monadic computations. Rivas and Jaskelioff [25] show that these two optimisations are instances of the Cayley representation for monoids in a generalised setting, and extend it to applicative functors. Our work extends this representation to include the additional operators present in non-deterministic computations by moving from generalised monoids to generalised near-semirings.

10.2. Representation of Near-semirings

Statman [30] provides a connection between lambda calculus and the algebra of near-semirings. He introduces a generalisation of Hoogewijs' representation [9] on which we base our double Cayley representation. Krishna and Chatterjee [17] study the representation of near-semirings in categories, but they only consider Cartesian structures and thus exclude monads and applicative functors.

10.3. Backtracking Monad Transformers

Hinze [8] is the first to derive a (continuation-passing) implementation of the backtracking monad transformer, which is equivalent to the ListT transformer we have mentioned.

Pirog [22] characterises backtracking monad transformers by extending the Eilenberg-Moore algebras of a monad with a monoid structure. This is akin to our taking the monad coproduct with lists.

Uustalu [32] considers different notions of non-determinism by extending the algebras of a monad with different algebraic structures. In the case where one extends them with monoids one obtains the notion of near-semiring category that we use in this article. Uustalu does not study the constructions that we provide in this article except for initiality, which is a particular case of our free construction.

Van der Ploeg and Kiselyov [23] provide a technique for improving theoretical running times of different constructions that handle reflection, including the backtracking monad transformer. While theoretically good, their implementation has big constant factors, and our optimisation achieves better running times (as long as reflection is only needed at the end of the computation).

Jaskelioff and Rivas [15] present a simple technique for obtaining efficient implementations of non-determinism monads and non-determinism applicative functors. The technique provides an alternative to the Double-Cayley representation and has the advantage of allowing reflection, but the near-semiring laws only hold observationally.

10.4. Free Alternatives

Capriotti and Kaposi [4] study the free Applicative construction, but do not present an approach for the free Alternative.

Kmett's `free`[8] package does contain a definition of the free Alternative construction that is, implicitly, based on the right-biased definition of the Day convolution:

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

Postan, Rivas, and Jaskelioff [24] study a variant of non-determinism based on dioids, where left catch is required to hold, instead of the left

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

distribution of near-semirings. They construct the free alternative for this particular notion of non-determinism.

10.5. Coproducts of Monad

We have used coproduct of monads to construct the free near-semiring over a monad. Lüth and Ghani [18] show how to form the coproduct of two *layered* monads (i.e., monads for which η has a partial left inverse), although their data-type representation is not exact. Uustalu and Ghani [6] improve this result with an exact formula for the restricted class of *ideal* monads. Recently, Adámek et al. [1] have used this formula to form the coproduct of *consistent* monads (Set monads for which η_X is injective). However, their result depends on set theoretical arguments, which restricts its applicability to Set functors. Our unital monads are more general than ideal monads, and work for any category. However, when working in Set, they are a strict subclass of consistent monads.

10.6. Applications

Search Heuristics. Schrijvers et al. [27] construct a free monad transformer for the non-deterministic choice operator in order to expose the search tree structure and apply pruning heuristics. After pruning, the resulting search tree is reflected back into the underlying non-determinism monad. Their work differs from ours in that they do not enforce any of the non-determinism axioms, in fact, they deliberately wish to observe the original syntactic structure.

Interleaved Parsers. Swierstra and Dijkstra [31] have proposed their interleaving combinator as a generalisation of earlier combinators for permutations [2] and merged lists. Brown [3] provides a transformer for interleaving Alternatives that are also Monads. His approach provides both more features (e.g., early termination) and fewer (e.g., the transformed Alternative is only Applicative).

11. Conclusions

This paper has introduced a generalised notion of near-semirings, and defined the free near-semiring and the novel double Cayley construction generically. By imposing a near-semiring structure on the instances of MonadPlus and Alternative, we have then obtained these useful constructions by instantiation of the general definition.

We have taken advantage of the unified view and translated the well-known ListT monad transformer, to obtain an applicative functor transformer, and shown that this transformer is useful even though applicative functors are closed under composition and hence one can obtain another transformer by composing with the list applicative functor. Moreover, we have characterised the most general way in which one can extend a unital monad to a non-determinism monad, by constructing the free non-determinism monad over a monad.

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

Not all MonadPlus and Alternative instances proposed in the literature or found "in the wild" are near-semirings. It would be interesting to investigate what algebraic structures underpin them. In particular, MonadPlus instances satisfying the left-catch axiom could be related to *dioids*, and their categorical generalisation [7].

This article shows some of the advantages of a unified view of monads and applicative functors as monoids, and of non-determinism monads and non-determinism applicative functors as near-semirings. Arrows [10] can also be seen as monoids [13, 25], so it is natural to ask if one can also obtain a non-determinism arrow by adding some equations to ArrowPlus. This seems to be the case, but obtaining the corresponding free construction and double-Cayley representation requires care as the closure of the corresponding tensor is not obvious.

Acknowledgements

This work has been partially funded by the Flemish Fund for Scientific Research (FWO) and the KU Leuven Latin America Fund.

References

[1] Adámek, J., Milius, S., Bowler, N., Levy, P. B., 2012. Coproducts of Monads on Set. In: Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science. LICS '12. IEEE Computer Society, Washington, DC, USA, pp. 45–54.

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

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

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

[5] Day, B., Feb. 1973. Note on monoidal localisation. Bulletin of the Australian Mathematical Society 8, 1–16.

[6] Ghani, N., Uustalu, T., Oct. 2004. Coproducts of Ideal Monads. Theoret. Informatics Appl. 38 (4), 321–342.

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

[8] Hinze, R., Sep. 2000. Deriving Backtracking Monad Transformers. SIGPLAN Not. 35 (9), 186–197.

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

[10] Hughes, J., 5 2000. Generalising monads to arrows. Science of Computer Programming 37 (1-3), 67–111.

[11] Hughes, R. J. M., Mar. 1986. A novel representation of lists and its application to the function "reverse". Inf. Process. Lett. 22 (3), 141–144.

[12] Hutton, G., Meijer, E., Jul. 1998. Monadic parsing in haskell. J. Funct. Program. 8 (4), 437–444.

[13] Jacobs, B., Heunen, C., Hasuo, I., 2009. Categorical semantics for arrows. J. Funct. Program. 19 (3-4), 403–438.

[14] Jaskelioff, M., 2009. Lifting of operations in modular monadic semantics. Ph.D. thesis, University of Nottingham.

[15] Jaskelioff, M., Rivas, E., 2015. Functional pearl: A smart view on datatypes. In: Fisher, K., Reppy, J. H. (Eds.), Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP '15. pp. 355–361.

[16] Kiselyov, O., Shan, C.-c., Friedman, D. P., Sabry, A., 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming. ICFP '05. ACM, New York, NY, USA, pp. 192–203.

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

[18] Lüth, C., Ghani, N., 2002. Composing monads using coproducts. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. ICFP '02. ACM, New York, NY, USA, pp. 133–144.

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

[20] McBride, C., Paterson, R., Jan. 2008. Applicative programming with effects. J. Funct. Program. 18 (1), 1–13.

[21] Moggi, E., Jul. 1991. Notions of computation and monads. Inf. Comput. 93 (1), 55–92.

[22] Piróg, M., 2016. Eilenberg-Moore Monoids and Backtracking Monad Transformers. In: Atkey, R., Krishnaswami, N. (Eds.), Proceedings 6th Workshop on Mathematically Structured Functional Programming, Eindhoven, Netherlands, 8th April 2016. Vol. 207 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 23–56.

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

[24] Postan, E., Rivas, E., Jaskelioff, M., 2017. Dioids for computational effects. In: Proceedings of Simposio Latinoamericano de Teoría Computacional, Conferencia Latinoamericana de Informática (XLIII CLEI), Córdoba, Argentina, September 2017.

[25] Rivas, E., Jaskelioff, M., 2017. Notions of Computation as Monoids. J. Funct. Program. Accepted for publication.

[26] Rivas, E., Jaskelioff, M., Schrijvers, T., 2015. From monoids to near-semirings: The essence of MonadPlus and Alternative. In: Falaschi, M., Albert, E. (Eds.), Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming. PPDP'15. ACM, pp. 196–207.

[27] Schrijvers, T., Wu, N., Desouter, B., Demoen, B., 2014. Heuristics Entwined with Handlers Combined: From Functional Specification to Logic Programming Implementation. In: Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming. PPDP '14. ACM, New York, NY, USA, pp. 259–270.

[28] Spivey, J. M., Jul. 2009. Algebras for combinatorial search. J. Funct. Program. 19 (3-4), 469–487.

[29] Spivey, M., Nov. 2012. When Maybe is not good enough. J. Funct. Program. 22 (6), 747–756.

[30] Statman, R., 2014. Near Semi-rings and Lambda Calculus. In: Dowek, G. (Ed.), Rewriting and Typed Lambda Calculi. Vol. 8560 of LNCS. Springer International Publishing, pp. 410–424.

[31] Swierstra, D., Dijkstra, A., 2013. Parse Your Options. In: Achten, P., Koopman, P. (Eds.), The Beauty of Functional Code. Vol. 8106 of LNCS. Springer Berlin Heidelberg, pp. 234–249.

[32] Uustalu, T., Aug. 2016. A divertimento on MonadPlus and nondeterminism. J. of Log. and Algebr. Methods in Program 85 (5, Part 2), 1086–1094, articles dedicated to Prof. J. N. Oliveira on the occasion of his 60th birthday.

[33] Voigtländer, J., 2008. Asymptotic improvement of computations over free monads. In: Proceedings of the 9th International Conference

on Mathematics of Program Construction. MPC ’08. Springer-Verlag, Berlin, Heidelberg, pp. 388–403.