Algebras for Weighted Search

DONNACHA OISÍN KIDNEY, Imperial College London, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom

Weighted search is an essential component of many fundamental and useful algorithms. Despite this, it is relatively under explored as a computational effect, receiving not nearly as much attention as either depth- or breadth-first search. This paper explores the algebraic underpinning of weighted search, and demonstrates how to implement it as a monad transformer.

The development first explores breadth-first search, which can be expressed as a polynomial over semirings. These polynomials are generalised to the free semimodule monad to capture a wide range of applications, including probability monads, polynomial monads, and monads for weighted search. Finally, a monad transformer based on the free semimodule monad is introduced. Applying optimisations to this type yields an implementation of pairing heaps, which is then used to implement Dijkstra's algorithm and efficient probabilistic sampling. The construction is formalised in Cubical Agda and implemented in Haskell.

CCS Concepts: • **Theory of computation** → **Shortest paths**; **Backtracking**; *Proof theory*; *Constructive mathematics*; *Type theory*; *Logic and verification*; Algebraic semantics; • **Software and its engineering** → **Functional languages**; *Data types and structures*.

Additional Key Words and Phrases: Haskell, Agda, graph search, monad

ACM Reference Format:
Donnacha Oisín Kidney and Nicolas Wu. 2021. Algebras for Weighted Search. *Proc. ACM Program. Lang.* 5, ICFP, Article 72 (August 2021), 30 pages. https://doi.org/10.1145/3473577

1 INTRODUCTION

Functional programmers often describe a problem with multiple solutions by a nondeterministic program where sequential composition counts how many steps it takes to produce a solution, and parallel composition offers the choice between solutions. Different search strategies on such a program produce different ways of structuring the solution space. This paper is about how search strategies relate to different free structures, and how understanding that structure can lead to efficient search algorithms.

The two most common strategies for exploring the solutions of a nondeterministic program are depth-first and breadth-first. It is well known that a depth-first strategy produces a list of solutions, and that the underlying structure of lists are free monoids where the monoid allows solutions to be listed in order. Less well known is the fact that a breadth-first strategy produces a tree that can be though of as a list of bags, and that the underlying structure here is that of free semirings. Semirings are a structure that involve two monoids, and in this case one is for sequential and the other is for parallel composition of solutions.

When the solutions to a problem can additionally be annotated with a weight—which may, for instance, represent a probability—more sophisticated search patterns become expressible. Weighted searches use this information to organise the solutions in terms of the weights, and a typical

Authors' addresses: Donnacha Oisín Kidney, Imperial College London, London, United Kingdom, o.kidney21@imperial.ac.uk; Nicolas Wu, Imperial College London, London, United Kingdom, n.wu@imperial.ac.uk.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/8-ART72
https://doi.org/10.1145/3473577

example is Dijkstra's algorithm. In this paper we demonstrate that that the structure of weight-first strategies is the free semimodule, this corresponds to a free semiring with an additional scalar multiplication, where this scalar allows weights to be accounted for.

To understand this construction, we use breadth-first search and its representation as a free semiring as a starting point. This construction can be understood by the language of polynomials (over a semiring), and by generalizing the exponents in this semiring from natural numbers to arbitrary monoids, the structure for weighted search emerges. Working at this level of abstraction means that the results can be applied in different contexts: we show how reservoir sampling, shortest subset sum, and Dijkstra's algorithm are all instances of this theory.

One difficulty with these constructions is their efficiency. The naive representation of the free semimodule, for instance, implements Dijkstra's algorithm with exponential time. To improve this we employ the well-known pattern of using Cayley representations of monoids. Additionally, we show how hyperfunctions further improve performance by enabling fusion.

To summarize, the contributions of this paper are:

- a formalization of breadth-first search in terms of the Levels type—which is the free semiring—using polynomials, as well as an optimised breadth-first search using the the Queue type, which is an efficient free applicative using the Cayley representation (Section 2);
- a monad transformer, the LevelsT type, which is an efficient free semiring that is optimised using a Cayley representation and hyperfunctions (Section 3);
- a formalization of weighted search called the Weighted type—which is the free semimodule—where a monus is used to express weighted search (Section 4);
- an efficient implementation of Weighted, called the Heap type (and its associated HeapT transformer), which is achieved by relaxing the commutativity rule to build a heap (Section 5);
- the application of the HeapT to the implementation of reservoir sampling, shortest subset sum, and Dijkstra's algorithm, as well as benchmarks that demonstrate performance (Section 6).

Finally, related work is dicussed (Section 7), before the paper concludes (Section 8). Supporting code for this paper is available at https://doi.org/10.5281/zenodo.4774319 [Kidney and Wu 2021].

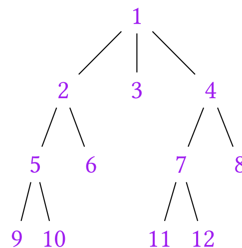
2 BREADTH-FIRST SEARCH IS POLYNOMIAL

One way to represent a nondeterministic search space of solutions is with a tree given by the Tree datatype, where Tip indicates that there are no solutions, and each node of the tree x & xs contains a solution x, together with branches xs that lead to further possible solutions.

```
data Tree a = Tip | a & [Tree a]
```

The two most common ways of enumerating the solutions in such a search space are using depth-first and breadth-first traversals. For instance, consider the following tree:

```
tree = 1 & [2 & [5 & [9  & [ ]
                     , 10 & [ ]]
                , 6 & [ ]]
           , 3 & [ ]
           , 4 & [7 & [11 & [ ]
                      , 12 & [ ]]
                , 8 & [ ]]]
```



The depth-first enumeration of this tree can be described by the *dfe* function:

```
dfe :: Tree a → [a]                          dfe tree ≡ [1, 2, 5, 9, 10, 6, 3, 4, 7, 11, 12, 8]
dfe Tip       = []
dfe (x & xs) = [x] ++ choices dfe xs
```

In the Tip case, there are no solutions, so the empty list is returned, otherwise, the node x & xs immediately returns the solution $[x]$ followed by the choices offered in xs, which are extracted using *choices*:

```
choices :: Alternative f ⇒ (a → f b) → [a] → f b
choices f []       = empty
choices f (x : xs) = f x <|> choices f xs
```

This uses the Alternative class to model nondeterminism: the <|> operation represents nondeterministic choice, and *empty* represents no solutions. This class assumes that <|> forms a monoid with *empty*, which means that <|> is associative with unit *empty*. When called by *dfe*, this specialises to the instance on lists, where *empty* = [], and (<|>) = (++).

The reason for keeping the operations in *choice* abstract is that the <|> operation there plays a very different role to the operation ++ in *dfe*. Both of these operators express a kind of nondeterministic choice, but ++ chooses between a parent and its children, and <|> chooses between siblings. The definition of these operators is what dictates the order of the search. For depth-first search, parents come before their children, and children are ordered left-to-right. If this order is changed to children before parents, for instance, the tree would be enumerated in post order:

```
poe :: Tree a → [a]                          poe tree ≡ [9, 10, 5, 6, 2, 3, 11, 12, 7, 8, 4, 1]
poe Tip       = []
poe (x & xs) = [x] +̇+ choices poe xs
  where lhs +̇+ rhs = rhs ++ lhs
```

The only change here from *dfe* is that *poe* uses +̇+, which is a flipped version of ++.

In this paper children will be treated equally: this requires a data structure that does not order its elements. This is exactly what a *bag* (sometimes also called a multiset) provides, where, for instance, ⦃2, 3, 5, 2⦄ = ⦃2, 2, 3, 5⦄. Here, the <|> operator expresses no preference between its operands.

2.1 Levels

When parents are ordered before their children, but siblings are unordered, the result of an enumeration is a list of bags containing the children at each respective level of the search tree; this result is encapsulated by the Levels type:

```
newtype Levels a = Levels [⦃a⦄]
```

One way to think of the Levels type is as nondeterministic computations where the outcomes are grouped into buckets by the number of "steps" needed to reach an outcome. For example, the value Levels [⦃a⦄, ⦃⦄, ⦃b, c⦄, ⦃d⦄] represents a computation which has 4 outcomes (a, b, c, and d), where a takes 0 steps to reach, b and c both take 2, and d takes 3. Another example is *nats*:

```
nats :: Levels ℕ
nats = Levels [⦃n⦄ | n ← [0 ..]]
```

This results in Levels [⦃0⦄, ⦃1⦄, ⦃2⦄ ..], which is a computation that takes n steps to reach each ⦃n⦄.

Levels is a well-known type which has been defined multiple times [Spivey [2000], Spivey and Seres [2003] and Fischer [2009] have versions of the type differ only slightly from the one presented

here]. The type and its variants are often used in the definition of breadth-first search algorithms; here for instance it is used to define a breadth-first search enumeration:

$$
\begin{array}{l}
bfe :: \mathsf{Tree}\ a \to \mathsf{Levels}\ a \\
bfe\ \mathsf{Tip} \qquad = \mathsf{Levels}\ [\,] \\
bfe\ (x \,\&\, xs) = pure\ x +\!\!\!+ choices\ bfe\ xs \\
\quad \textbf{where}\ lhs +\!\!\!+ rhs = lhs \mathbin{\langle|\rangle} wrap\ rhs
\end{array}
\qquad
\begin{array}{l}
bfe\ tree \equiv \\
\quad \mathsf{Levels}\ [\wr 1 \int, \wr 2,3,4 \int, \wr 5,6,7,8 \int, \wr 9,10,11,12 \int]
\end{array}
$$

The only difference between this function and dfe is the replacement of $+\!\!\!+$ with another ordering. This ordering is described by the operators $pure$, $wrap$, and $\langle|\rangle$, which assign the cost of steps:

$$
\begin{array}{l}
pure :: a \to \mathsf{Levels}\ a \\
pure\ x = \mathsf{Levels}\ [\wr x \int]
\end{array}
\qquad
\begin{array}{l}
wrap :: \mathsf{Levels}\ a \to \mathsf{Levels}\ a \\
wrap\ (\mathsf{Levels}\ xs) = \mathsf{Levels}\ (\wr\ \int : xs)
\end{array}
$$

The result of $pure\ x$ assigns a cost of 0 to the value x by placing it in the first bag in the list of outcomes, and $wrap\ xs$ increments the cost of xs by prepending an empty bag.

The other function used in this definition is $\langle|\rangle$, which comes from the Alternative instance:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Alternative\ Levels}\ \textbf{where} \\
\quad empty = \mathsf{Levels}\ [\,] \\
\quad \mathsf{Levels}\ xs \mathbin{\langle|\rangle} \mathsf{Levels}\ ys = \mathsf{Levels}\ (zipL\ xs\ ys)
\end{array}
\qquad
\begin{array}{l}
zipL\ [\,] \quad ys \quad = ys \\
zipL\ xs \quad [\,] \quad = xs \\
zipL\ (x : xs)\ (y : ys) = (x \cup y) : zipL\ xs\ ys
\end{array}
$$

The $\langle|\rangle$ operator combines bags of the corresponding levels, using $zipL$. The $zipL$ function (sometimes called "long zip" [Gibbons and Jones 1998]) zips two lists by merging bags at the same cost level and keeping the leftovers if one list is longer than the other.

One interesting aspect of the Levels type is its Monad instance. Like lists, Levels can be used to model nondeterministic computations, however its search order is breadth-first, rather than depth-first. The following function, for instance (which searches for Pythagorean triples) would diverge without producing a single answer if standard lists had been used instead of Levels. With Levels, however, all the results are enumerated:

```
1  pyth :: Levels (ℕ, ℕ, ℕ)
2  pyth = do
3    x ← nats
4    y ← nats
5    z ← nats
6    guard (x > 0 ∧ y > 0 ∧ z > 0)
7    guard (x ∗ x + y ∗ y ≡ z ∗ z)
8    pure (x, y, z)
```

```
>>> mapM_ print pyth
(3,4,5)
(4,3,5)
(6,8,10)
(8,6,10)
(5,12,13)
...
```

To understand the Monad instance it helps to remember that Levels represents a collection of nondeterministic computations, grouped into bags that are ordered in a list by the number of steps it took to reach them. In $pyth$, for example, the output is ordered by the number of steps it takes to reach the last statement, on line 8. Since it takes n steps to reach each $\wr n \int$ in $nats$, it therefore takes x steps to reach x on line 3, a further y steps to reach line 4, and a further z steps to reach line 5. When computations are sequenced, the number of steps in the resulting computation is the sum of the number of steps of the constituent computations; in this example, the number of steps to reach the end of line 5 is the sum of x, y and z. The $guard$ functions prune the search space: any outcomes which do not satisfy the two predicates on lines 6 and 7 are discarded. The $guard$ function, like $pure$, adds nothing to the cost, meaning that the final cost on line 8 is the same as the cost on line 5 ($x + y + z$). This explains why `(6,8,10)` comes before `(5,12,13)` in the output.

The Monad instance describes how sequencing works. The $\gg\!=$ operation extends a search with a continuation: $p \gg\!= k$ takes the outcomes of the program p and searches from them using k, while adding appropriate costs.

```
instance Monad Levels where
  Levels [ ]        >>= k = empty
  Levels (xs : xss) >>= k = choices' k xs <|> wrap (Levels xss >>= k)
```

In the case of the empty computation, there are no outcomes to search from, so $\gg\!=$ returns no solutions. In the other case, the computations are divided into those it takes 0 steps to reach (xs), and those it takes at least one step to reach (xss). The solutions in xs are extended by k using $choices'\ k\ xs$, where $choices'$ is a version of $choices$ which works on bags, that fairly combines the result of applying k to each outcome in xs by using $\langle|\rangle$. This is combined with recursively applying $\gg\!= k$ to the outcomes in xss, which each take one extra step to reach, expressed here with $wrap$.

2.2 Polynomials in $\mathbb{N}$

Up until now, Levels has been explained as a nondeterministic computation where outcomes are grouped by the steps it takes to reach them. While that mental model is useful and intuitive, it turns out that there is another conceptual model of Levels that yields much more theoretical insight, especially from an algebraic perspective. This model is that of Levels as a *polynomial*.

Informally speaking, a polynomial is an expression consisting of variables, constants, addition, multiplication, and exponentiation by a natural number. It is possible to represent a polynomials of a single variable (in normalised form) with a list of coefficients, where the ith entry in the list is the coefficient of the term x^i.

$$(2x^2 + 1)(x^2 + 2) = 2x^4 + 4x^2 + x^2 + 2 = 2x^0 + 0x^1 + 5x^2 + 0x^3 + 2x^4 \simeq [2, 0, 5, 0, 2]$$

This normalised representation is where the link between Levels and polynomials arises.

The link is easiest to see when Levels is specialised to Levels (). There, Levels is a list of $\wr()\int$. The type $\wr()\int$ is a bag full of just one value, repeated many times; there is a simple isomorphism between it and the natural numbers:

$$\wr\int \mapsto 0, \wr()\int \mapsto 1, \wr(),()\int \mapsto 2, \wr(),(),()\int \mapsto 3, \ldots$$

As such, the type Levels () is isomorphic to $[\mathbb{N}]$: the concrete representation of polynomials[1].

That fact alone is not so surprising: such isomorphisms are common, and often uninteresting ($\mathbb{N}$ itself is isomorphic to $[\mathbb{N}]$, after all). What is interesting is that the operations on a polynomial—addition, multiplication, exponentiation by a natural number—are all defined on the Levels type directly, and in fact some have been used in this paper already. Addition corresponds to $\langle|\rangle$:

$$(2x^2 + 1) + (x^3 + 2) \simeq \text{Levels}\ [1, 0, 2] \langle|\rangle \text{Levels}\ [2, 0, 0, 1] = \text{Levels}\ [3, 0, 2, 1] \simeq 3 + 2x^2 + x^3$$

Multiplication corresponds to the monadic sequencing operator $\gg$, where $xs \gg ys = xs \gg\!= \lambda_ \to ys$:

$$\begin{aligned}(2x^2 + 1) \times (x^2 + 2) &\simeq \text{Levels}\ [1, 0, 2] \gg \text{Levels}\ [2, 0, 1] \\ &= \text{Levels}\ [2, 0, 1] \langle|\rangle\ wrap\ (wrap\ (\text{Levels}\ [2, 0, 1] \langle|\rangle \text{Levels}\ [2, 0, 1])) \\ &= \text{Levels}\ [2, 0, 1] \langle|\rangle\ wrap\ (wrap\ (\text{Levels}\ [4, 0, 2])) \\ &= \text{Levels}\ [2, 0, 1] \langle|\rangle \text{Levels}\ [0, 0, 4, 0, 2] \\ &= \text{Levels}\ [2, 0, 5, 0, 2] \simeq 2x^4 + 5x^2 + 2\end{aligned}$$

[1] Strictly speaking, for the isomorphism with polynomials to be valid the Levels type has to ignore trailing empty bags (because polynomials ignore trailing empty zeroes). Our implementation does indeed do this.

2.3 Polynomials in Alternative

Unfortunately, the definition of multiplication for Levels () does not generalise properly to Levels a, since the type of $(\gg) :: \text{Levels } a \rightarrow \text{Levels } b \rightarrow \text{Levels } b$ necessitates that all search outcomes of the first operand are discarded. Programs like *pyth* (Section 2.1), however, require access to intermediate results like x, y, and z (lines 3–5). A more general definition of multiplication is needed.

The first step of that generalisation is realising the coefficients in polynomials need not be $\mathbb{N}$: they can be any *semiring*, which abstracts + and × to monoids with appropriate structure.

Definition 1 (Semiring). A semiring is a combination of a monoid $(R, \times, 1)$ (called the multiplicative monoid) and a *commutative* monoid $(R, +, 0)$ (called the additive monoid), related by the following laws:

$$0 \times x = 0 \quad \text{(Left Annihilation)} \qquad x \times (y + z) = (x \times y) + (x \times z) \quad \text{(Left Distribution)}$$

$$x \times 0 = 0 \quad \text{(Right Annihilation)} \qquad (x + y) \times z = (x \times z) + (y \times z) \quad \text{(Right Distribution)}$$

However, this definition of multiplication is still not general enough, since its operands are required to have the same type. The type of multiplication which requires both operands to be used in the output is $(\times) :: \text{Levels } a \rightarrow \text{Levels } b \rightarrow \text{Levels } (a, b)$. This operator is known to be an alternative formulation of $(\circledast) :: \text{Levels } (a \rightarrow b) \rightarrow \text{Levels } a \rightarrow \text{Levels } b$ from the Applicative type class [McBride and Paterson 2008; Paterson 2012].

There is already a significant amount of work on understanding the Applicative, Alternative, classes in terms of monoids and (near) semirings: Rivas and Jaskelioff [2014] gave a concrete framework for understanding Applicative in terms of monoidal categories, and Rivas et al. [2018] extended the work to include the nondeterminism classes, which were given the algebraic interpretation of near-semirings. Our work differs from theirs in that we deal primarily with semirings (not near-semirings), but the theory of monoidal categories and the categorification of near-semirings still applies. The key observation of this theory is that semirings can be generalised from pairs of monoids to pairs of monoidal categories: when these categories are Set and lax monoidal functors (the category for Applicative functors), the relevant semiring is that of Alternative and Applicative. For our purposes, this means it is sufficient to provide Applicative and Alternative instances for a type for it to be a semiring.

We have already given an Alternative instance for Levels, and so all that remains is an appropriate Applicative instance. While such an instance can be derived from the Monad instance automatically, it is instructive to see the direct implementation of ⊛:

```
(⊛) :: Levels (a → b) → Levels a → Levels b
Levels [ ]       ⊛ _         = Levels [ ]
Levels (x : xs) ⊛ Levels ys = Levels (map (x⊛) ys) <|> wrap (Levels xs ⊛ Levels ys)
```

Notice that this is a computational interpretation of the right distribution law in Definition 1. Furthermore, this implementation of ⊛ is structurally identical to the implementation of multiplication given by McIlroy [1999] when multiplying streams.

Having now implemented instances of both Alternative and Applicative for Levels, the theory of Rivas et al. [2018] establishes that Levels is a polynomial in the Alternative and Applicative semiring in the Set and lax monoidal functors categories.

2.4 Polynomials in Applicative

The Levels type allows us to stage and rearrange the outcomes of programs with nondeterminism using Alternative, Applicative, and *wrap*[2]; we may also want to stage and rearrange the outcomes of programs with other effects. In a sense, Levels took the Alternative-Applicative semiring and

facilitated the reordering of Alternative expressions with the *wrap* combinator. A type to stage and reorder *effects*, then, can be constructed from a polynomial with Alternative swapped out for some monoid in a monoidal category representing effects. Applicatives themselves represent effects, as it happens, so the polynomial for rearranging and staging effects will be the polynomial over the Applicative-Applicative semiring.

The Levels type takes its structure from the free monoid (i.e. it is a list, albeit with some extra structure): similarly, the polynomial of the Applicative-Applicative semiring takes its structure from the free applicative [Capriotti and Kaposi 2014]:

```
data Ap f a where
  Pure :: a → Ap f a
  Lift :: (a → b → c) →
            f a → Ap f b → Ap f c

lift :: f a → Ap f a
lift x = Lift const x (Pure ())

lower :: Applicative f ⇒ Ap f a → f a
lower (Pure x)      = pure x
lower (Lift f x xs) = liftA2 f x (lower xs)
```

The Ap type is the free applicative: it is a list of effectful computations, terminated by a pure value, and interspersed with functions to combine the results of each computation. Any $f\ a$ can be *lift*ed into the free applicative; conversely, if f is itself Applicative, *lower* defines conversion in the other direction. It uses the *liftA2* function, which combines effectful computations with a binary function:

```
liftA2 :: Applicative f ⇒ (a → b → c) → f a → f b → f c
liftA2 f xs ys = pure f <*> xs <*> ys
```

As Ap is indeed the free applicative, *lower* forms an applicative homomorphism.

Similar to the Monoid instance for the free monoid, the Applicative instance for the free applicative performs concatenation:

```
instance Applicative (Ap f) where
  pure :: a → Ap f a
  pure = Pure
  (<*>) :: Ap f (a → b) → Ap f a → Ap f b
  Pure x      <*> Pure y      = Pure (x y)
  Pure x      <*> Lift g y ys = Lift (λy ys → x (g y ys)) y ys
  Lift f x xs <*> ys          = Lift (λx (y, z) → f x y z) x (liftA2 (,) xs ys)
```

In the Pure case, there are no effects in the left-hand-side operand; as such, <*> returns the right-hand-side operand with the pure value modified by x. In the Lift case, the effect of x remains, and the tails of the lists are concatenated; the combining function f is modified to handle the new structure of the tail of the list.

However, this is not the only possible Applicative instance on Ap. In particular, if f is itself Applicative there is a definition of <*> which mimics the definition of + on polynomials:

```
Pure x      <*> Pure y      = Pure (x y)
Pure x      <*> Lift g y ys = Lift (λy ys → x (g y ys)) y ys
Lift f x xs <*> Pure y      = Lift (λx xs → f x xs y) x xs
Lift f x xs <*> Lift g y ys =
  Lift (λ(x, y) (xs, ys) → f x xs (g y ys)) (liftA2 (,) x y) (liftA2 (,) xs ys)
```

[2]These operators are the Applicative cousins of a "bunch" from Spivey [2009]

Where the previous definition of $\circledast$ was similar to $+\!\!+$ on lists, this is similar to long zip (*zipL*), or $\langle|\rangle$ on Levels. In the case where either operand is Pure, this definition of $\circledast$ is equivalent to the previous concatenation-based definition. When both operands are Lift, however, the effects at the head of the list are combined immediately using their applicative instance.

While it is also likely possible to define $\times$ on this type, a law-abiding implementation would be degenerative: the distributivity laws on $\circledast$ and $\langle|\rangle$ means that $\circledast$ would have to ignore the type parameters of one of its arguments.

Instead, what we can use this type for is the scheduling of effects. Like Levels, it has a *wrap* function, which pushes its contents one level lower.

```
wrap :: Applicative f ⇒ Ap f a → Ap f a
wrap = Lift (λ_ x → x) (pure ())
```

Rather than altering the order of nondeterministic results, this alters the order of effect execution:

```
stages :: Ap IO [String]
stages = sequenceA
  [wrap (wrap (wrap (out "a")))
  , wrap (out "b")
  , wrap (wrap (out "c"))
  , out "d"
  , wrap (out "e")]
```

```
>>> lower stages
out: d
out: b
out: e
out: c
out: a
["a","b","c","d","e"]
```

```
out s = lift (do putStrLn ("out: " ++ s); pure s)
```

The function $sequenceA :: \text{Applicative}\ f \Rightarrow [f\ a] \rightarrow f\ [a]$ takes a list of effectful computations, runs each effect, and returns the list of the results of each computation. In this example it evaluates each of the Ap IO computations in turn: as can be seen in the output on the right, each letter is printed in order of how many *wrap*s it was nested under, starting with the fewest, "d", and ending with "a", which was nested under 3 *wrap*s. Notice also that despite the reordering of effects the pure value itself is unaffected: the strings in the returned list are in precisely the same order as they were given.

An especially interesting use of this type is for breadth-first traversal of trees [Easterly 2019]:

```
bft :: Applicative f ⇒ (a → f b) → Tree a → f (Tree b)
bft f = lower ∘ g where g Tip      = pure Tip
                        g (x & xs) = liftA2 (&) (lift (f x)) (wrap (traverse g xs))
```

The *bft* function can be compared to *traverse*: both of these functions have the same type signature (when specialised to trees), and both apply an applicative action to every value in the container without changing the shape of the container. In fact, *bft* is a valid implementation of *traverse*, although the default implementation of *traverse* will evaluate effects in depth-first order, while *bft* will evaluate them in breadth-first order.

The *bft* function can help solve the tricky problem of breadth-first numbering [Okasaki 2000]:

```
renumber :: Tree a → Tree Int
renumber = flip evalState 0 ∘ bft num
  where num _ = do modify (+1); get
```

```
evalState :: State s a → s → a
modify    :: (s → s) → State s ()
get       :: State s s
```

The *renumber* function numbers the nodes of a tree in breadth-first order. First, the *bft* function applies the effectful action *num* to every value in the tree. This action produces an effect in the state monad [Jones 1995]: the state in question here is an integer counter, which is first incremented

using *modify* and then the updated value is retrieved and returned using *get*. Finally, *evalState* runs the whole computation with an initial state of 0.

2.5 Cayley Transforms

While elegant and theoretically interesting, this implementation of *bft* is slow. At every node in the tree, *bft* is recursively called on the children of the node, and then the constructed effects are combined using ⟪*⟫. Unfortunately, since ⟪*⟫ is $O(n)$, this makes the overall complexity of *bft* $O(n^2)$.

In fact this same problem plagues the implementation of *bfe* in Section 2.1: there, too, the children of a node are traversed once in a recursive call, and then combined using a zip-like function (‹|› in the case of *bfe*). The solution in that case is to perform the zipping in the recursive call itself, fusing the two steps together.

$$
\begin{array}{ll}
bfe :: \mathsf{Tree}\ a \to \mathsf{Levels}\ a & f\ \mathsf{Tip}\qquad\quad qs \qquad\ = qs \\
bfe\ t = \mathsf{Levels}\ (f\ t\ [\,]) & f\ (x\ \&\ xs)\ (q : qs) = \wr x \wr \cup q : foldr\ f\ qs\ xs \\
 & f\ (x\ \&\ xs)\ [\,] \qquad = \wr x \wr \quad\ : foldr\ f\ [\,]\ xs
\end{array}
$$

This implementation of *bfe* fuses away the call to *choices*: instead of mapping over the list of children and then using ‹|› to zip them together, for every child *f* performs the zipping and recursion in the same step. A similar version of this function can be found in Gibbons and Jones [1998].

Adapting this optimisation to work with the *bft* function is difficult: a mechanical translation of the above code would not work, for instance, as the types don't match up. Instead, first the above optimisation must rephrased in more general terms. After this generalisation the optimisation can be re-specialised to the Ap type, yielding an $O(n)$ implementation of *bft*.

The general optimisation technique in particular that *bfe* uses is the Cayley transform.

Definition 2 (Cayley's Theorem). Cayley's theorem for monoids [Cayley 1854; Jacobson 1985] states that every monoid embeds into a monoid of transformations. Given a monoid $(M, \bullet, \epsilon)$, we can construct the transformation monoid $(M \to M, \circ, \mathrm{id})$, called the Cayley representation. We have the following two homomorphisms for converting between a monoid and its Cayley representation:

$$
\begin{array}{ll}
abs\ \ :: M \to (M \to M) & \qquad rep\ \ :: (M \to M) \to M \\
abs\ x = \lambda y \to x \bullet y & \qquad rep\ x = x\ \epsilon
\end{array}
$$

Cayley's theorem is of interest to computer scientists because it can be used to reduce the cost of repeated applications of the monoidal binary operator. Because the Cayley representation's monoidal binary operator (function composition) has $O(1)$ complexity, the Cayley representation can provide a more efficient representation of some monoidal type for applications which make heavy use of the monoid operations. Its most well-known use is difference lists [Clark and Tärnlund 1977; Hughes 1986], where it optimises ++ on the list monoid. In the case of *bfe*, the monoid being optimised is the (Levels, ‹|›, *empty*) monoid.

So how does this apply to the *bft* function? Rivas and Jaskelioff [2014] generalised the Cayley transform to monoidal categories, noticing that another well-known optimisation (the codensity monad transformer, Hutton et al. 2010; Voigtländer 2008) was in fact an instance of the Cayley transform. They also demonstrated how to apply the Cayley transform to the monoid of applicatives, which allows for the optimisation of *bft*.

On Applicatives, the Cayley representation has the following type:

$$\mathbf{newtype}\ \mathsf{Cayley}\ f\ a = \mathsf{Cayley}\ \{ runC :: \forall b.\ f\ b \to f\ (a, b) \}$$

This type is obtained by partially applying *liftA2* (,) to some Applicative value; similarly to how the Cayley monoid on set is obtained by partially applying the monoidal binary operator to a value.

Conversion back to the underlying Applicative is done by applying the function to the neutral element for the monoid (which is *pure* for Applicatives).

```
abs :: Applicative f ⇒ f a → Cayley f a
abs x = Cayley (liftA2 (,) x)

rep :: Applicative f ⇒ Cayley f a → f a
rep x = fmap fst (runC x (pure ()))
```

Finally, the actual monoidal operators themselves in this category are precisely the Applicative methods. They are implemented on Cayley like so:

```
instance Functor f ⇒ Applicative (Cayley f) where
  pure x  = Cayley (fmap (x, ))
  fs <*> xs = Cayley (fmap (λ(f, (x, xs)) → (f x, xs)) ∘ runC fs ∘ runC xs)
```

Notice that <*> here itself is basically a form of function composition.

We now define a queue of applicative effects; it is the Cayley transform of the Ap type.

```
type Queue f = Cayley (Ap f)
```

This type is the "efficient" version of the Ap type, allowing a definition of an $O(n)$ *bft*.

The only remaining operator that needs to be defined is *wrap*.

```
wrap :: Applicative f ⇒ Queue f a → Queue f a
wrap xs = Cayley (f xs) where f :: Applicative f ⇒ Queue f a → Ap f b → Ap f (a, b)
                              f xs (Pure y)      = Lift (const id) (pure ()) (runC xs (Pure y))
                              f xs (Lift g y ys) = Lift (fmap ∘ g) y (runC xs ys)
```

This function serves the same purpose as *wrap* on Levels, or indeed the *wrap* defined previously on Ap. Here there is some more plumbing required, however.

Finally, the following is the efficient implementation of *bft*:

```
bft :: Applicative f ⇒ (a → f b) → Tree a → f (Tree b)
bft f = lower ∘ rep ∘ h where h Tip      = pure Tip
                              h (x & xs) = liftA2 (&) (abs (lift (f x))) (wrap (traverse h xs))
```

Other than the conversion to and from the Queue type, this implementation is structurally identical to *bft* using Ap.

We will also include the following slightly more efficient implementation of *bft*, although it is not necessary to understand for the rest of the paper:

```
bft :: Applicative f ⇒ (a → f b) → Tree a → f (Tree b)
bft f Tip      = pure Tip
bft f (x & xs) = liftA2 (&) (f x) (bftF f xs)

bftF :: Applicative f ⇒ (a → f b) → [Tree a] → f [Tree b]
bftF t = fmap head ∘ foldr (<*>) (pure [ ]) ∘ foldr f [pure ([ ]:)] where
  f Tip qs = qs
  f (x & xs)  ~(q : qs) = liftA2 (⋈) (t x) q : foldr f (p qs) xs

  p [ ]      = [pure ([ ]:)]
  p (x : xs) = fmap (([ ]:)∘) x : xs

  (x ⋈ k) ~(xs : ks) = ((x & xs) : y) : ys where ~(y : ys) = k ks
```

This function is only marginally more efficient than the previous one; basically we have inlined a number of the Applicative methods where we can, which allows for a slight reduction in the number of overall operations. More important, however, is the fact that we noticed we could remove the

existentials and GADTs from the solution after this inlining. This allowed us to use lazy irrefutable pattern-matches (those matches that are prefixed with ~, which are not permitted on GADTs), which means this function is significantly lazier than the previous. In fact it can handle infinite trees, something the Ap-based solution cannot. There may be different formulations of the free applicative which have this laziness property.

3 THE LEVELS TRANSFORMER

Monads such as the Levels type are a convenient abstraction that enable specific effects to be expressed in a controlled manner within a program. The problem with monads, however, is that they do not generally compose with one another. The solution to this is to provide a *monad transformer*, which takes a monad m, and augments it to a monad $t\ m$ with additional effects.

To this end, this section introduces the LevelsT monad transformer, which adds the effect of breadth-first backtracking with Levels to an arbitrary monad (Section 3.1). The main difficulty is producing an efficient version of this transformer, and this is achieved by combining the Cayley representation with hyperfunctions to enable the fusion of intermediate datastructures (Section 3.2).

3.1 The Levels Transformer

The Levels monad is a list of bags, and so it should be no surprise that the LevelsT transformer is based on the list transformer [Jaskelioff and Moggi 2010], but where the elements are bags.

```
newtype LevelsT m a = LevelsT {runLevelsT :: m (Maybe (Bag a, LevelsT m a))}
```

This transforms a monad m by adding a list (of bags) whose constructors are nested by m. The list [⟅1⟆, ⟅2⟆, ⟅3⟆] could be encoded in LevelsT m (Just (⟅1⟆, m (Just (⟅2⟆, m (Just (⟅3⟆, m Nothing)))))), although this representation intersperses every bag in the list with some effect m. The cons constructor (:) has been replaced by Just, and the nil constructor ([ ]) has been replaced by Nothing.

The implementations of ≫= and <|> mimic the implementations of the same functions on Levels, written in the monadic style.

```
instance Monad m ⇒
  Monad (LevelsT m) where
  LevelsT xs ≫= k = LevelsT (xs ≫= go) where
    go Nothing        = pure Nothing
    go (Just (x, xs)) = runLevelsT
      (choices′ k x <|> wrap (xs ≫= k))
    wrap xs = LevelsT (pure (Just (⟅⟆, xs)))

LevelsT xs <|> LevelsT ys =
    LevelsT (liftA2 go xs ys)
  where
    go Nothing ys = ys
    go xs Nothing = xs
    go (Just (x, xs)) (Just (y, ys)) =
      Just (x ∪ y, xs <|> ys)
```

The LevelsT type is quite similar to LogicT in Kiselyov et al. [2005], though with the crucial difference that LogicT does not group outcomes into bags. There, they seek to define "fair" analogues to ≫= and <|> for nondeterministic programs; on lists, these analogues have the following definitions:

```
interleave :: [a] → [a] → [a]
interleave [ ]      ys = ys
interleave (x : xs) ys = x : interleave ys xs

(≫-) :: [a] → (a → [b]) → [b]
xs ≫- k = foldr (interleave ∘ k) [ ] xs
```

The *interleave* function is fairer than ++, as it produces a list with items drawn from each of its operands in an alternating fashion (*interleave* $[a, b, c]$ $[x, y, z]$ = $[a, x, b, y, c, z]$), where ++ first produces all the elements in the first operand and only then inspects the second. The ≫- function is similarly fairer than ≫=: as it uses *interleave*, its search pattern follows a kind of diagonalisation, in a pseudo-breadth-first fashion. Despite that, these functions are still not as fair as the implementations

of $\ggg$ and $\langle|\rangle$ on Levels, since there is still left bias. This bias also forces undesirable strictness: the *pyth* example would diverge if implemented with $\succ\!\!-$, for instance. Finally, $\succ\!\!-$ does not form a monad; *interleave* does not form a monoid; and their combination does not form a (near) semiring. The LevelsT type remedies these problems, while retaining the ability to define "unfair" (what we would call depth-first) variants of $\ggg$ and $\langle|\rangle$.

3.2 Hyperfunctions and Polynomial Fusion

There is a Cayley transformed version of the LevelsT transformer which is worth exploring:

$$\textbf{newtype}\ \mathsf{LevelsT}\ m\ a = \mathsf{LevelsT}\ \{ \mathit{runLevelsT} :: \forall r.\ (\wr a \wr \to m\ r \to m\ r) \to m\ r \to m\ r \}$$

This is the same as the continuation-based LogicT in Kiselyov et al. [2005] (except that we keep bags of elements). The type is a Cayley representation of the $+\!\!+$ monoid on lists. Equivalently, it's the type of the church-encoded list transformer. It fuses away the depth-first variants of $\langle|\rangle$ and $\ggg$. In Kiselyov et al. [2005], the breadth-first operations are implemented, but the resulting time complexity is $O(n^2)$.

The poor time complexity stems from the implementation of $\langle|\rangle$: on church-encoded lists (which LogicT and LevelsT are, structurally speaking), zip-like functions such as $\langle|\rangle$ or *zip* are quite tricky to implement efficiently. This trickiness has in fact caused problems before: in Haskell, for instance, it prevented the use of list fusion on the *zip* function. List fusion is an important optimisation in Haskell that can dramatically improve the performance of certain functions on lists (and especially the composition of multiple functions on lists). The class of functions that are amenable to fusion is effectively those functions that can be written using the *foldr* function; equivalently, the fusable list functions are those functions that can be defined on church-encoded lists. So functions like *map*, $+\!\!+$, and *filter* are fusable, but *tail*[3] and *foldr1* are not. For some time it was thought that *zip* fell into the latter category [Gill et al. 1993], but Launchbury et al. [2000] showed that it does not, and in doing so paved the way for an efficient implementation of $\langle|\rangle$ on LevelsT.

In order to explain the implementation of $\langle|\rangle$, it helps to first look at the implementation of *zip* that uses *foldr*. First, consider the following standard definition of *zip*:

```
zip :: [a] → [b] → [(a, b)]
zip []       ys       = []
zip xs       []       = []
zip (x : xs) (y : ys) = (x, y) : zip xs ys
```

In order to rewrite this function as a fold, it is helpful to first rewrite it so that it pattern-matches on each list one at a time, in separate functions:

```
zip :: [a] → [b] → [(a, b)]     xz [] _          = []         yz [] _ _          = []
zip xs ys = xz xs ys            xz (x : xs) ys = yz ys xs x   yz (y : ys) xs x = (x, y) : xz xs ys
```

This version of *zip* is equivalent to the first, it simply has been split up so that the processing of each list is made clearer. It also makes it clearer that the *xz* and *yz* functions themselves are folds, and can be rewritten as such:

```
zip :: [a] → [b] → [(a, b)]     xz = foldr f b where          yz = foldr f b where
zip xs ys = (xz xs) (yz ys)       b _ = []                      b _ _ = []
                                  f x xk yk = yk xk x           f y yk xk x = (x, y) : xk yk
```

[3]While it is of course possible to define *tail* on church-encoded lists, such a definition is $O(n)$, and so not suitable for fusion, the whole purpose of which is to improve performance.

However at this point there is a problem: this program is not well-typed. The error specifically is `cannot construct the infinite type: t0 ~ a -> (t0 -> [(a, b)]) -> [(a, b)]`. A useful trick for circumstances such as these is to simply take the type error and define a type which satisfies it.

```
newtype Zip a b = Zip {runZip :: a → (Zip a b → [(a, b)]) → [(a, b)]}
```

Indeed, this newtype (when inserted in the appropriate places) can make the fold-based *zip* above well-typed. Instead of showing that version, however, we will use a slight variant on the Zip type that is more general: the type of *hyperfunctions* [Launchbury et al. 2000].

```
newtype a ↬ b = Hyp {invoke :: (b ↬ a) → b}
```

This type captures the same recursion pattern as Zip, and can be used to write a well-typed, fold-based implementation of *zip*:

```
zip :: [a] → [b] → [(a, b)]
zip xs ys = invoke (xz xs) (yz ys)

xz :: [a] → (a → [(a, b)]) ↬ [(a, b)]
xz = foldr f b where
  f x xk = Hyp (λyk → invoke yk xk x)
  b = Hyp (λ_ → [ ])

yz :: [b] → [(a, b)] ↬ (a → [(a, b)])
yz = foldr f b where
  f y yk = Hyp (λxk x → (x, y) : invoke xk yk)
  b = Hyp (λ_ _ → [ ])
```

To use hyperfunctions to implement <|> on LevelsT the type needs to be altered slightly:

```
newtype a ↬_m b = HypM {invokeM :: m ((a ↬_m b → a) → b)}
```

Since the <|> function will be interleaving monadic effects, the hyperfunction needs to have a way to wrap the recursive calls in monadic effects. With this new type, the definition of <|> can be derived from *zip* in a relatively straightforward way, with the only real difference being the threading of a monadic effect through the function.

```
(<|>) :: LevelsT m a → LevelsT m a → LevelsT m a
xs <|> ys = LevelsT (λc n → xz xs ≫= (yz c n ys≫=))

xz xs = runLevelsT xs f b
  where
    f x xk = pure (λyk → yk (HypM xk) x)
    b = pure (λyk → yk (HypM b) ⟅⟆)

yz c n ys = runLevelsT ys f (pure b)
  where
    f y yk =
      pure (λxk x →
        c (x ∪ y)
          (invokeM xk ≫= (yk≫=)))
    b xk ⟅⟆ = n
    b xk x  = c x (invokeM xk ≫= ($b))
```

4 WEIGHTED SEARCH IS A SEMIMODULE

Breadth-first search is useful, but it only describes a narrow range of the useful algorithms for search. In effect, it allows us to run a search where the order is dictated by the number of discrete steps each computation is from the root. The problem is that some algorithms, such as Dijkstra's algorithm, order the search by some arbitrary cost.

This section expands on the algebraic explanation of breadth-first search to include a monad for cost-weighted search. We will show that breadth-first search is in fact a special case of this new

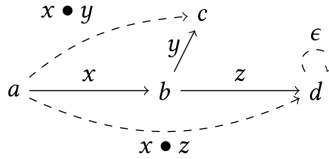
algebra, and we will define the free construction on the algebra and show how it relates to other well-known constructions like the probability monad.

The semimodule is an algebra which contains commutativity laws: traditionally, the inclusion of such laws makes free objects tricky (if not impossible) to define in Martin-Löf type theory [Altenkirch et al. 2011]. To deal with this, we will present our constructions and code in this section in Cubical Agda [Vezzosi et al. 2019], which is an extension to Agda which adds, among other things, the ability to define quotiented data types. This is crucial to our construction of the free semimodule, and enables much of the formalisation in this section.

4.1 An Algebra of Costs

The first thing to describe precisely is the notion of "cost". This is the quantity that later search algorithms will minimise; it underpins a lot of the later understanding in this paper so it makes sense to stop briefly and consider how to characterise it algebraically.

As a starting point, we argue that any reasonable type which can function as a cost must be totally ordered, and monoidal. The requirement for a total order comes from the fact that Dijkstra's algorithm (as well as all the other search algorithms that will be implemented in this paper) rely on the ability to compare costs. The monoidal constraint comes from the need to be able to describe the cost of constructed paths: if there is a path $a \longrightarrow b$ with cost x, and a path from $b \longrightarrow c$ with cost y, there must be a constructed path $a \longrightarrow b \longrightarrow c$, and its cost should be $x \bullet y$; the sum of the costs of the constituent paths. Another constructed path is the trivial path: it should cost nothing (ϵ) to go from d to d. These two operations, along with some common-sense laws, tell us that any cost should be a monoid.



Next there is the question of how the total order and monoid operations interact. The following two laws should hold, for instance:

$$\forall x.\ \epsilon \leqslant x \qquad (1) \qquad\qquad \forall x, y, z.\ y \leqslant z \implies x \bullet y \leqslant x \bullet z \qquad (2)$$

1 says that ϵ is the smallest cost. 2 says that if $b \longrightarrow c$ is a cheaper path than $b \longrightarrow d$, then the constructed path $a \longrightarrow b \longrightarrow c$ should be cheaper than $a \longrightarrow b \longrightarrow d$.

While it is possible to construct a granular hierarchy of different notions of cost (commutative vs non-commutative costs, cancellative costs, antisymmetric vs non-antisymmetric costs, etc.), we have opted for the simpler approach of picking one set of relatively strong laws to describe the "cost algebra". Some of these laws may well exclude interesting types, but they do not exclude the most important examples, and they allow some important optimisations which are used in Section 5.

The first simplification is to specify precisely *which* order the cost must be defined with:

```
x ≤ y = ∃[ k ] (y ≡ x • k)
```

This is Agda syntax: it means "$x \leqslant y$ when there exists a k such that $x \bullet k = y$". Constructively, the proof is a pair, where the first component is k and the second is an equality proof.

This relation is sometimes called the "minimal" [Wehrung 1992] preorder, and it is a preorder on all monoids. It follows 1 and 2, as the following Agda proofs show. First, $\epsilon \leqslant x$ because $x = \epsilon \bullet x$.

```
positive : ∀ x → ε ≤ x                    ε• : ∀ x → ε • x ≡ x
positive x = x , sym (ε• x)
```

Proving the second law is a little more involved: if there is a k such that $z = y \bullet k$, then by associativity $x \bullet z = (x \bullet y) \bullet k$, and therefore $x \bullet y \leqslant x \bullet z$.

```
•-cong : ∀ x {y z} → y ≤ z → x • y ≤ x • z     lemma : x • z ≡ (x • y) • k
•-cong x {y} {z} (k , z≡y•k) = k , lemma        lemma = x • z        ≡⟨ cong (x •_) z≡y•k ⟩
                                                        x • (y • k) ≡˘⟨ assoc x y k ⟩
                                                        (x • y) • k ∎
```

Showing that the relation is reflexive and transitive is similar, and is not included here.

The second restriction we will make for simplicity is that we will require the cost monoid to be commutative. This is perhaps a more tenuous requirement—we only really require it to fully generalise monus to semiring, and for some proofs about the type in Section 5—but since all cost monoids relied on in this paper are commutative, and indeed it is quite difficult to formulate a non-commutative cost monoid which is suitable for search algorithms, we do not feel any great guilt in including it.

Finally, then, the algebra for costs: the monus.

Definition 3 (Monus). A monus is a commutative monoid $(M, \bullet, \epsilon)$ such that the relation defined by $x \leqslant y \iff \exists z.\ y = x \bullet z$ is total and antisymmetric.

The following two things are derivable for free from the monus definition:

```
|_-_| : M → M → M                    zeroSumFree : ∀ x y → x • y ≡ ε → x ≡ ε
| x - y | with x ≤? y                zeroSumFree x y x•y≡ε =
... | inl (k , y≡x•k) = k              antisym (y , sym x•y≡ε) (positive x)
... | inr (k , x≡y•k) = k
```

The first function here is an "absolute difference" operator; this function is integral to some of the optimisations in Section 5. The second property says that any costs which sum to ϵ must equal ϵ.

4.2 Generalising from Monus to Semiring

Before proceeding forward with the monus type, it's worth stopping to notice an opportunity for generalisation. The monus algebra provides almost enough information to define a semiring (Definition 1): the min-plus semiring, in particular. The multiplicative monoid of this semiring is given by the underlying monoid on the monus $(M, \bullet, \epsilon)$. The additive monoid is partially specified, as the minimum function is an appropriate binary operator:

```
_⊓_ : M → M → M              •-distribˡ-⊓ : ∀ x y z → (x ⊓ y) • z ≡ (x • z) ⊓ (y • z)
x ⊓ y with x ≤? y             •-distribʳ-⊓ : ∀ x y z → x • (y ⊓ z) ≡ (x • y) ⊓ (x • z)
... | inl x≤y = x
... | inr y≤x = y
```

However a value for 0 is missing: $\sqcap$ forms only a semigroup, not a monoid in other words.

One value that would fit for 0 is the supremum of the monus: it is the identity for $\sqcap$, and annihilates with $\bullet$. While not every monus has a maximal element, it is possible to simply adjoin a maximal element and proceed from there. This extra value does not make any difference to the algorithms in later sections, and it is always possible to retrieve the underlying monus. For the remainder of this section we will work with semirings in general; the monus will return in Section 5.

4.3 Defining a Monad for Least-Cost Search

This section will develop the Weighted type: a monad for expressing nondeterministic computations weighted by some semiring R. This monad will give a theoretical underpinning to the more efficient and practical implementations later in the paper, while also providing some insight into the parallels between weighted search and probabilistic programming.

The type is based on lists:

```
data Weighted (A : Type a) : Set (a ℓ⊔ ℓ) where
  []      : Weighted A
  _◃_∷_ : (p : R) (x : A) (xs : Weighted A) → Weighted A
```

This type is a list of weight-value pairs. For example, the value $2 \triangleleft x :: 5 \triangleleft y :: []$ represents a computation with a value x of weight 2, and a value y of weight 5.

The following three constructors define quotients, equations which hold on the Weighted type:

```
com : ∀ p x q y xs → p ◃ x ∷ q ◃ y ∷ xs ≡ q ◃ y ∷ p ◃ x ∷ xs
dup : ∀ p q x xs   → p ◃ x ∷ q ◃ x ∷ xs ≡ p + q ◃ x ∷ xs
del  : ∀ x xs        → 0# ◃ x ∷ xs         ≡ xs
```

com ensures that the list is unordered. This is essential for ordered search: it means that only the weights or the values themselves may be used to order the computations. dup says that when two computations return the same value the overall weight should be equal to the sum of their respective weights. Specialised to the monus, where $+ = \sqcap$, this says that when a search hits the same outcome twice the cost to reach that outcome is the minimum of those respective costs. Finally, del says that 0-weighted things should be ignored. Specialised to the monus again, this rule can be thought of as stating that computations which are infinitely costly to reach are not in fact reachable at all.

The final constructor for this type is the following:

```
trunc : isSet (Weighted A)
```

Without the inclusion of this constructor it would be possible to construct equalities on the Weighted type which are not themselves equal: in other words, there may be more than one value which can inhabit the type $x \equiv y$ if x and y are values in the Weighted monad (this is a consequence of the inclusion of quotients on the Weighted type). The trunc constructor states that "all equalities on Weighted are equal", effectively collapsing the higher homotopy structure generated by the quotients. While this higher homotopy structure is fascinating, it is not the subject of this paper, and the version of the Weighted monad without trunc does not faithfully represent the precise monad that we are interested in.

4.4 Paramorphism-Based Recursion Schemes for Quotiented Data Types

Rather than defining recursive functions on this type directly we will define them in terms of recursion schemes; in particular we will define them as *paramorphisms* [Meertens 1992]. The reason for this is practical: by using folds we can avoid some of the boilerplate involved in defining functions over quotient types in Cubical Agda. Our approach to the development of these recursion schemes is a little unorthodox, so we will go through it briefly here.

The first step is to define the *base functor* for the Weighted type. Here we present the "normal" base functor for a type like Weighted on the left, and the version that we use on the right:

```
data 𝔚 (A : Type a) (B : Type b) :
              Type (a ℓ⊔ b ℓ⊔ ℓ) where
  []     : 𝔚 A B
  _◂_∷_ : R → A →
          B → 𝔚 A B
```

```
data 𝔚 (A : Type a) (P : Weighted A → Type p) :
              Type (a ℓ⊔ p ℓ⊔ ℓ) where
  []         : 𝔚 A P
  _◂_∷_⟨_⟩ : R → A →
              (xs : Weighted A) → P xs → 𝔚 A P
```

The type on the left is perfectly adequate for defining functions like sum : Weighted ℕ → ℕ, which can be written as simple catamorphisms. Dependent functions, on the other hand, like map-ident : (xs : Weighted A) → map id xs ≡ xs cannot be written as catamorphisms: they are instead a kind of *fibered paramorphism*, where the algebra has access to the structure being recursed over, and returns a type family over the inductive type rather than a simple type like ℕ. This more complicated recursion scheme requires the base functor definition on the right.

We can convert from this base functor type into the Weighted type with the ⟪_⟫ function, which further allows us to define algebras on the base functor:

```
⟪_⟫ : 𝔚 A P → Weighted A
⟪ [] ⟫                  = []
⟪ w ◂ x ∷ xs ⟨ _ ⟩ ⟫ = w ◂ x ∷ xs
```

```
Alg : (A : Type a) (P : Weighted A → Type p) →
        Type _
Alg A P = (xs : 𝔚 A P) → P ⟪ xs ⟫
```

The next added complexity of these recursion schemes is that we haven't *technically* defined a base functor on the Weighted type, because 𝔚 makes no mention of the quotients on Weighted. To define algebras on Weighted we actually need to restrict Alg to those algebras which are *coherent*, where we define coherency with the following type:

```
record Coherent (ψ : Alg A P) : Type (p ℓ⊔ a ℓ⊔ ℓ) where
  field c-set : ∀ xs → isSet (P xs)
        c-dup : ∀ p q x xs ψ⟨xs⟩ →    ψ (p ◂ x ∷ (q ◂ x ∷ xs) ⟨ ψ (q ◂ x ∷ xs ⟨ ψ⟨xs⟩ ⟩) ⟩)
                                       ≡[ i ≔ P (dup p q x xs i) ]≡
                                        ψ ((p + q) ◂ x ∷ xs ⟨ ψ⟨xs⟩ ⟩)
        c-com : ∀ p x q y xs ψ⟨xs⟩ → ψ (p ◂ x ∷ (q ◂ y ∷ xs) ⟨ ψ (q ◂ y ∷ xs ⟨ ψ⟨xs⟩ ⟩) ⟩)
                                       ≡[ i ≔ P (com p x q y xs i) ]≡
                                        ψ (q ◂ y ∷ (p ◂ x ∷ xs) ⟨ ψ (p ◂ x ∷ xs ⟨ ψ⟨xs⟩ ⟩) ⟩)
        c-del : ∀ x xs ψ⟨xs⟩ →          ψ (0# ◂ x ∷ xs ⟨ ψ⟨xs⟩ ⟩) ≡[ i ≔ P (del x xs i) ]≡ ψ⟨xs⟩
```

This type says that an algebra (ψ) is coherent if it respects the four quotients on Weighted. Each field corresponds to a quotient. Because these algebras are dependently-typed, the equalities involved are heterogeneous: an equality x ≡[i ≔ p]≡ y means that x is equal to y, and p proves that their types are equal at the points given by i.

We can now finally define dependent (Ψ) and non-dependent (Φ) paramorphisms on Weighted:

```
Ψ : (A : Type a) (P : Weighted A → Type p) →
      Type _
Ψ A P = Σ (Alg A P) Coherent
```

```
Φ : Type a → Type b → Type (a ℓ⊔ b ℓ⊔ ℓ)
Φ A B = Ψ A (λ _ → B)
```

And we can run one of the recursion schemes using the ⟦_⟧ function (the implementation of which we have elided):

```
⟦_⟧ : Ψ A P → (xs : Weighted A) → P xs
```

We have further defined the category of algebras on Weighted: we have included the definition of objects, arrows, composition, and identity in our formalisation.

4.5 The Monad

Thankfully, it is far easier to use these recursion schemes than it is to implement them. As an example function to demonstrate their use we will implement the scalar multiplication operator `_⋊_ : R → Weighted A → Weighted A`, which is needed to define the monad instance. The operator multiplies the weight of every item in the list by a given value.

$$w \rtimes (w_1 \triangleleft x :: w_2 \triangleleft y :: w_3 \triangleleft z :: []) \equiv (w \times w_1) \triangleleft x :: (w \times w_2) \triangleleft y :: (w \times w_3) \triangleleft z :: []$$

There are three steps to implement a function like this: first, we need to define the computational component of the algebra itself.

```
⋊-alg : R → Φ A (Weighted A)
⋊-alg w .fst (p ◃ x ∷ xs ⟨ w⋊xs ⟩) = (w × p) ◃ x ∷ w⋊xs
⋊-alg w .fst []                     = []
```

Then, we need to prove that this is coherent:

```
⋊-alg w .snd .c-com p x q y xs w⋊xs = com (w × p) x (w × q) y w⋊xs

⋊-alg w .snd .c-dup p q x xs w⋊xs =
                           w × p ◃ x ∷ w × q ◃ x ∷ w⋊xs ≡⟨ dup (w × p) (w × q) x w⋊xs ⟩
                           w × p + w × q ◃ x ∷ w⋊xs     ≡˘⟨ cong (_◃ x ∷ w⋊xs) (×⟨+⟩ w p q) ⟩
                           w × (p + q) ◃ x ∷ w⋊xs ■

⋊-alg w .snd .c-del x xs w⋊xs =
                           w × 0# ◃ x ∷ w⋊xs ≡⟨ cong (_◃ x ∷ w⋊xs) (×0 w) ⟩
                           0# ◃ x ∷ w⋊xs     ≡⟨ del x w⋊xs ⟩
                           w⋊xs ■
```

And finally we wrap all of this up in a function:

```
_⋊_ : R → Weighted A → Weighted A
x ⋊ xs = ⟦ ⋊-alg x ⟧ xs
```

We have implemented the rest of the monad functions in a similar fashion, and proven that the monad laws are satisfied. We will not include the full proofs here, just the computational content of bind and pure.

```
_»=_ : Weighted A → (A → Weighted B) →
         Weighted B
xs »= f = ⟦ bind-alg f ⟧ xs

pure : A → Weighted A
pure x = 1# ◃ x ∷ []
```

```
bind-alg : (A → Weighted B) →
               Φ A (Weighted B)
bind-alg f .fst (p ◃ x ∷ _ ⟨ xs ⟩) = (p ⋊ f x) ∪ xs
bind-alg f .fst [] = []
```

4.6 The Free Semimodule

An algebraic understanding of Weighted comes from the fact that it is the free semimodule. Semimodules are commutative monoids that are scaled by a semiring with the ⋊ operator. An example is sets: they form a commutative monoid under union, and they have a semimodule with the booleans.

$$\text{false} \rtimes x = \emptyset \qquad\qquad \text{true} \rtimes x = x$$

Another well-known example of a semimodule is finite the probability monad [Erwig and Kollmansberger 2006], where the semiring which scales is the probability semiring.

Definition 4 (Semimodule). A semimodule is a commutative monoid $(M, \cup, \emptyset)$, with a semiring $(R, +, 0, \times, 1)$, and an operation $\rtimes : R \to M \to M$ which satisfies the laws:

$$
\begin{aligned}
(x \times y) \rtimes z &= x \rtimes (y \rtimes z) & 1 \rtimes x &= x \\
(x + y) \rtimes z &= (x \rtimes y) \cup (x \rtimes z) & 0 \rtimes x &= \emptyset \\
x \rtimes (y \cup z) &= (x \rtimes y) \cup (x \rtimes z) & r \rtimes \emptyset &= \emptyset
\end{aligned}
$$

There is significant research on semimodules and their uses [Golan 2003]. They encapsulate, in a sense, the unordered containers: sets, bags, and weighted maps. We provide a proof that the Weighted type is in fact the free semimodule in the supplementary material.

The free semimodule is a generalisation of the Levels construction. Where Levels grouped computations by discrete steps, Weighted groups by any arbitrary semiring. In fact, Levels is a specialisation of the Weighted type: Levels can be retrieved by setting the semiring in the Weighted type to the bag-index semiring, which is a semiring of bags of natural numbers, with the additive monoid $(\wr\mathbb{N}\int, \cup, \wr\int)$ and multiplicative monoid $(\wr\mathbb{N}\int, \wr\times\int, \wr 0\int)$. The multiplication operator here is defined as follows:

$$xs \wr\times\int ys = \wr x + y \mid x \leftarrow xs, y \leftarrow ys \int$$

It takes the Cartesian product of its two operands, and adds together the corresponding indices.

Summary. This section has shown the derivation of the free semimodule monad, as well as how breadth-first search is a special case of algebras on this monad. We have further shown that this monad encapsulates polynomials, finite probability distributions, sets and multisets, and we have formalised all of this in Cubical Agda.

5 OPTIMISING WEIGHTED SEARCH

The theoretical grounding for weighted search has now been established as the free semimodule monad Weighted. Practically speaking, however, the type has a few problems: it is both inefficient and does not compose well with other monadic effects. This section resolves these two issues by implementing the HeapT type, which is a variant of the free semimodule monad[4].

This is achieved by applying some simple optimisations [Jaskelioff and Rivas 2015] and deriving a pairing heap [Fredman et al. 1986]. We will define a monad transformer on this type, and we will use this transformer to efficiently implement Dijkstra's algorithm, and probabilistic sampling.

5.1 Improving Efficiency

The Weighted type is clearly many optimisations short of an efficient monad for weighted search. All of the key operations ($\cup$, $\gg\!=$, $\rtimes$, etc.) are $O(n)$, which makes most uses slow, but $\gg\!=$ in particular has a pernicious performance problem. Consider bind on lists, which is also $O(n)$: in that case, chains of binds are only a problem if they're left-nested $(((xs \gg\!= f) \gg\!= g) \gg\!= \ldots)$; right-nested chains $(xs \gg\!= (\lambda x \to ys \gg\!= (\lambda y \to zs \gg\!= \ldots)))$ are $O(n)$ overall (modulo some caveats). In fact, many optimisation techniques (like the Cayley transform) on monads simply turn left-nested expressions into right-nested ones.

[4]We have not proven this formally, but we think it is likely not possible to implement a traditional monad transformer for the semimodule type. The reason is that a law-abiding monad transformer for list-like structures necessitates ordering on the elements of the list, which of course our type does not have. For commutative monads, a transformer may be possible (of the form M (Weighted A)), but this would be only for commutative monads and not all monads.

On the Weighted type, however, chains of binds are *always* at least $O(n^2)$, no matter which way the chain is associated. This is because $\gg=$ traverses both the left and right-hand-side argument, with a map and $\rtimes$ respectively, meaning that there is no way to avoid the pathological case.

This means that an implementation of, say, Dijkstra's algorithm *cannot* be in the right complexity class using the current implementation of Weighted: even if we carefully reassociate every operator the right way around we will still pay roughly exponential cost to search.

The solution to this problem is surprisingly simple and practical, and comes from Jaskelioff and Rivas [2015]. There, they use it to improve the complexity of operations like $+\!\!+$, but they show how to generalise it to cases like ours. The idea is simple: take the expensive operation that needs to be optimised away, and add it as a constructor to the type itself. Then, add the normalisation logic to the other functions on the type, taking care to ensure work is shared. This is essentially a pure functional version of the same trick used by structures like splay trees [Sleator and Tarjan 1985].

For the free semimodule, the expensive operation is $\rtimes$. This is the operation we need to optimise so that right-nested binds become efficient. A first approximation for an optimised type might look something like the following:

```
data Heap w a = w ◃ [Either a (Heap w a)]
```

But this is a slight inaccuracy: to implement the monad operations correctly, we should push the application of the $\rtimes$ operator down one level, giving us the following type[5]:

```
newtype Heap w a =                              data Node w a b = Leaf a | !w ◃ b
  Heap {runHeap :: [Node w a (Heap w a)]}
```

This is a forest of rose trees, where internal nodes are labelled by weight, and leaves are labelled with values. Crucially, right-nested binds are now $O(n)$.

For the remainder of this section we will not optimise this type further. However we should note that it is entirely possible to apply the same optimisation technique to yield an $O(1)$ $\langle|\rangle$, $O(1)$ $\gg=$ (from both sides), etc. Though such optimisations would likely result in better performance in practice, our goal is to achieve the correct asymptotic bounds for algorithms like Dijkstra's algorithm, which are accomplished with this optimisation alone.

5.2 Semantics of the Heap Monad

The implementation of the $\gg=$ operation on the Heap monad can be derived from the fact that the Heap monad is in fact a *free* monad. The free monad plus over the writer monad, to be specific:

```
Heap w a ≃ FreePlus ((,) w) a ≃ FreeT ((,) w) [] a

newtype FreeT f m a = FreeT {runFreeT :: m (Either a (f (FreeT f m a)))}
```

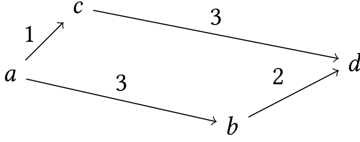
As a result of this type coincidence, the Monad and MonadPlus instances on Heap can be simply copied from the same instances on the free monad. The semantics of $\gg=$ on Heap are the following: given a tree, it substitutes all the nodes with a new tree, representing one step of exploration from that node in the graph. The weight associated with any value in the tree is equal to the sum of the weights on the path to the value from the root.

In order to illustrate these semantics in a little more detail, we will briefly describe how we can use the Heap type to manipulate graphs. We can represent graphs using the following type:

```
type Graph a = a → [(a, Dist)]
```

[5]The bang before w here is a strictness annotation: it means the field will be strict, forcing its contents when the constructor is pattern-matched on. It provides a modest performance improvement in this case.

The type Graph a represents graphs with vertices of type a, and edge weights of type Dist. Semantically it is a function that, when given a vertex, returns the neighbouring vertices tagged with their distances. Here is a graph along with its representation in code:



$$\begin{aligned} &\mathit{graph}\ a = [(b, 3), (c, 1)] \\ &\mathit{graph}\ b = [(d, 2)] \\ &\mathit{graph}\ c = [(d, 3)] \\ &\mathit{graph}\ d = [\,] \end{aligned}$$

There is something interesting going on with the Graph type, which we will explore here briefly. As a formal definition of graphs it's really not too far off, despite its simplicity: unlike definitions which rely on inductive types, it handles cycles and infinite graphs without any semantic trouble. The only issue is that it uses lists: really, we shouldn't specify an order on the neighbours of a vertex (other than one derived from the edge weights). Furthermore, we probably don't want to include duplicates in the list. In actual fact, what we want is precisely the quotients we defined on Weighted in the previous section. This means that $A \to$ Weighted A characterises graphs precisely: graphs are endomorphisms in the Kleisli category of the free semimodule monad.

So how do the semantics of $\gg\!=$ relate to graphs? Since graphs themselves are in fact Kleisli arrows over the Weighted monad, our optimised version of that monad—the Heap type—should itself also be able to represent graphs in some sense. And indeed it can. Conversion from a graph is possible with the following function:

$$\begin{aligned} &\mathit{fromGraph} :: \mathsf{Graph}\ a \to a \to \mathsf{Heap}\ \mathsf{Dist}\ a \\ &\mathit{fromGraph}\ g = \mathit{choices}\ (\lambda(x, w) \to \mathsf{Heap}\ [\,w \triangleleft \mathsf{Heap}\ [\mathsf{Leaf}\ x\,]\,]) \circ g \end{aligned}$$

This turns a graph into a function which returns a Heap of the neighbours of a vertex.

It is possible to manipulate the graph directly in this form. Transitive closure, for instance, can be defined on graphs in terms of the Kleene star. The Kleene star, in turn, can be defined using the semiring operators of the MonadPlus class:

$$\begin{aligned} &\mathit{star} :: \mathsf{MonadPlus}\ m \Rightarrow (a \to m\ a) \to a \to m\ a \\ &\mathit{star}\ f\ x = \mathit{pure}\ x \langle|\rangle (f\ x \gg\!= \mathit{star}\ f) \end{aligned}$$

When run on the graph above, this does indeed produce the transitive closure:

$$\mathit{star}\ (\mathit{fromGraph}\ \mathit{graph})\ a = \mathsf{Heap}\ [\mathsf{Leaf}\ a, 3 \triangleleft \underbrace{\mathsf{Heap}\ [\mathsf{Leaf}\ b, 2 \triangleleft \underbrace{\mathsf{Heap}\ [\mathsf{Leaf}\ d]}]}, 1 \triangleleft \underbrace{\mathsf{Heap}\ [\mathsf{Leaf}\ c, 3 \triangleleft \underbrace{\mathsf{Heap}\ [\mathsf{Leaf}\ d]}]}]$$

5.3 Prioritised Sampling

We now know how to construct values of the Heap type efficiently; next we need to figure out how to *use* it efficiently. The core operator we need for weighted search is the following:

$$\mathit{popMin} :: \mathsf{Ord}\ w \Rightarrow \mathsf{Heap}\ w\ a \to ([\,a\,], \mathsf{Maybe}\ (w, \mathsf{Heap}\ w\ a))$$

This function returns two things:

(1) Those items in the search who have the minimum weight associated with them (ϵ).
(2) The rest of the search (if there is any more to search), along with its weight.

Building a search algorithm using this operator is not difficult: by iteratively applying it to the entire search space we can list the elements of the space in order of their weight.

The following is the implementation of *popMin*:

$$\begin{array}{l} \mathit{popMin} = \mathit{second}\ \mathit{comb} \circ \mathit{partition} \circ \mathit{runHeap} \end{array}$$

$$\begin{array}{l} \mathit{partition} :: [\mathsf{Node}\ w\ a\ b] \to ([a], [(w, b)]) \\ \mathit{partition} = \mathit{foldr}\ f\ ([\,], [\,])\ \mathbf{where} \\ \quad f\ (\mathsf{Leaf}\ x)\ (\mathit{xs}, \mathit{ys}) = (x : \mathit{xs}, \mathit{ys}) \\ \quad f\ (w \triangleleft y)\ \ (\mathit{xs}, \mathit{ys}) = (\mathit{xs}, (w, y) : \mathit{ys}) \end{array}$$

This function has three components: first, it removes the newtype wrapper (*runHeap*); then, it partitions the top-level list into the values with no weight attached and those nested one level deeper (*partition*); and finally, it runs the *comb* function on those nested values, which merges them into a single heap.

So, *popMin* will return any Leaf nodes at the top level of the tree, along with any left over nodes at the next level. The remaining function to be defined here is *comb*:

$$\mathit{comb} :: \mathsf{Ord}\ w \Rightarrow [(w, \mathsf{Heap}\ w\ a)] \to \mathsf{Maybe}\ (w, \mathsf{Heap}\ w\ a)$$

The definition of this function has a huge influence on the overall performance of *popMin*; it turns out that we can take the definition almost directly from the same function on a pairing heap. The pairing heap [Fredman et al. 1986] is a priority queue with $\mathcal{O}(1)$ insert, merge, and find min, and $\mathcal{O}(\log n)$ delete min. It is also an extremely fast heap in practice: Larkin et al. [2013] found that it was the fastest pointer-based heap in a number of experiments covering sorting and Dijkstra's algorithm. In an interesting coincidence of types, the pairing heap is structurally the same type as Maybe $(w, \mathsf{Weighted}\ w\ a)$. This means the definition of *comb* can be lifted directly from its implementation on pairing heaps:

$$\begin{array}{l} \mathit{comb}\ [\,] \qquad\ = \mathsf{Nothing} \\ \mathit{comb}\ (x : \mathit{xs}) = \mathsf{Just}\ (\mathit{go}\ x\ \mathit{xs}) \end{array}$$

$$\begin{array}{l} \mathit{go}\ x\ \ [\,] \qquad\qquad = x \\ \mathit{go}\ x_1\ [x_2] \qquad\quad = x_1 \diamondsuit x_2 \\ \mathit{go}\ x_1\ (x_2 : x_3 : \mathit{xs}) = (x_1 \diamondsuit x_2) \diamondsuit \mathit{go}\ x_3\ \mathit{xs} \end{array}$$

This is the so-called "pairing merge", and is the key to the pairing heap's efficiency. Yet to be defined is the $\diamondsuit$ operator: that will be covered in the next section.

5.4 Monus, Again

$\diamondsuit$ is the last function to define, and it turns out to be slightly tricky, as the semantics of the Heap type are slightly different from the semantics of the pairing heap. The following, for instance, won't work:

$$\begin{array}{l} (\diamondsuit) :: \mathsf{Ord}\ w \Rightarrow (w, \mathsf{Heap}\ w\ a) \to (w, \mathsf{Heap}\ w\ a) \to (w, \mathsf{Heap}\ w\ a) \\ (x_w, \mathsf{Heap}\ \mathit{xs}) \diamondsuit (y_w, \mathsf{Heap}\ \mathit{ys}) \\ \quad \mid x_w \leqslant y_w\ \ = (x_w, \mathsf{Heap}\ ((y_w \triangleleft \mathsf{Heap}\ \mathit{ys}) : \mathit{xs})) \\ \quad \mid \mathit{otherwise} = (y_w, \mathsf{Heap}\ ((x_w \triangleleft \mathsf{Heap}\ \mathit{xs}) : \mathit{ys})) \end{array}$$

Remember that the semantics of the monad are that the children branches inherit the cost of their parents; here, if $x_w \leqslant y_w$ holds, $\diamondsuit$ returns $y_w \triangleleft \mathsf{Heap}\ \mathit{ys}$ as a sub-node with the parent weight x_w. This duplicates the weight x_w, semantically giving ys the incorrect weight of $x_w \diamond y_w$ (the same error is present if $x_w \leqslant y_w$ does not hold).

What is needed is a way to decompose the larger weight into its two components. Happily, the monus algebra (Definition 3) enables just that. Here is that same algebra defined in Haskell:

$$\mathbf{class}\ (\mathsf{Ord}\ a, \mathsf{Monoid}\ a) \Rightarrow \mathsf{Monus}\ a\ \mathbf{where}\ |\cdot - \cdot| :: a \to a \to a$$

This definition is slightly different from the one in Agda, as the Agda definition would be quite difficult to use without dependent types. Here, monuses are required to implement an absolute difference operator. Though the two formulations of monus seem quite different, they are actually equivalent, as long as instances of the Haskell class obey the following two laws:

$$x \leqslant x \diamond y \qquad\qquad x \leqslant y \implies x \diamond |x - y| = y$$

Finally, this operator gives a principled way to implement the $\diamondsuit$ function:

```
(x_w, Heap xs) ◇ (y_w, Heap ys)
  | x_w ⩽ y_w  = (x_w, Heap ((|x_w − y_w| ◂ Heap ys) : xs))
  | otherwise = (y_w, Heap ((|x_w − y_w| ◂ Heap xs) : ys))
```

5.5 Implementing a Transformer

As is, the weighted monad cannot layer effects (like exceptions or state) through its search. However, since the monad is explicitly constructed from the free monad transformer it is not difficult to define the transformer by swapping out lists for ListT [Jaskelioff and Moggi 2010; Piróg 2016].

```
type HeapT w m = FreeT ((,) w) (ListT m)
```

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

Inlined sufficiently, the type of the heap transformer is the following:

```
newtype HeapT w m a = HeapT {runHeapT :: ListT m (Node w a (HeapT w m a))}
```

Again, because this type is built out of free components, it definitionally is a law-abiding transformer.

The ◇ operator can be defined on this type, similarly to its pure implementation on Heap:

```
(x, xv) ◇ (y, yv)
  | x ⩽ y     = (x, (tell |x − y| ≫ yv) <|> xv)
  | otherwise = (y, (tell |x − y| ≫ xv) <|> yv)
```

```
tell :: Monad m ⇒ w → HeapT w m ()
tell w = HeapT (pure (w ◂ pure ()))
```

This function also uses the convenience function *tell*, which adds weight to a computation.

The rest of the algorithms are roughly the same as we have implemented them before, albeit with some slight modification to work around the wrapped effects. The following is the transformer version of *popMin*, for instance:

```
popMinT :: (Monus w, Monad m) ⇒ HeapT w m a → m ([a], Maybe (w, HeapT w m a))
popMinT = fmap (second comb ∘ partition) ∘ toListT ∘ runHeapT
```

A broad array of functions can be derived from this one operator, in transformer and non-transformer variants alike. *search*, and *searchT*, for instance, return all of the outcomes in the search space tagged with their weight, ordered by weight:

```
search :: Monus w ⇒ Heap w a → [(a, w)]
```

```
searchT :: (Monad m, Monus w) ⇒
             HeapT w m a → m [(a, w)]
```

6 APPLICATIONS OF WEIGHTED SEARCH

With all the structures in place to express and evaluate weighted search algorithms, this section demonstrates a number of applications (Section 6.1) before measuring performance (Section 6.2).

6.1 Algorithms

6.1.1 Reservoir Sampling. A useful monus that has not yet been mentioned is the monus of probability. Its monoid is the $(\mathbb{P}, \times, 1)$ monoid, and the total order is most-likely first (i.e. $\frac{7}{8} \leqslant \frac{1}{8}$). It has the following Monus instance:

$$\textbf{instance}\ \text{Monus}\ \mathbb{P}\ \textbf{where}\ |x - y| = \textbf{case}\ \mathit{compare}\ x\ y\ \textbf{of}\ \text{LT} \to \frac{y}{x}; \text{EQ} \to 1; \text{GT} \to \frac{x}{y}$$

The slightly odd ordering on $\mathbb{P}$ is actually precisely what one might want for a *sampling* monad: with this ordering, the heap returns values with the highest probability first.

Reservoir sampling [Vitter 1985] is an algorithm for sampling items from a discrete distribution without replacement. We consider the simplified case where only a single item is sampled:

```
r ⟵ 1;
repeat
    x ⟵ nextEvent();
    p ⟵ r × probabilityOf(x);
    return x with probability p;
    r ⟵ r / (1−p) ;
```

```
sample :: Heap Prob a → IO a
sample = go 1 ∘ search where
  go r ((x, px) : xs) = do
    let n/d = r ∗ px
    c ← randomRIO (1, d)
    if c ⩽ n then pure x else go (r / (1 − n/d)) xs
```

We iterate through the possible events in the distribution one-by-one, performing a probabilistic test at each event and possibly returning from the function if the test passes.

The performance of this algorithm is influenced directly by the order of items returned by the `nextEvent` function. Because the *search* function orders items from most to least likely, it reduces the amount of tests have to be done before an item is found and returned.

6.1.2 Shortest Subset Sum. The shortest subset sum problem is an NP-complete problem, where the task is to find the smallest subset such that it sums to the desired value. Implementing it using our heap is quite straightforward:

```
shortest :: Int → [Int] → [Int]
shortest t xs = head ∘ map fst ∘ search $ do
  subset ← filterM (const inclusion) xs
  guard (sum subset ≡ t)
  pure subset
```

```
inclusion :: Monad m ⇒ HeapT Dist m Bool
inclusion = (tell 0 ≫ pure False)
          <|> (tell 1 ≫ pure True)
```

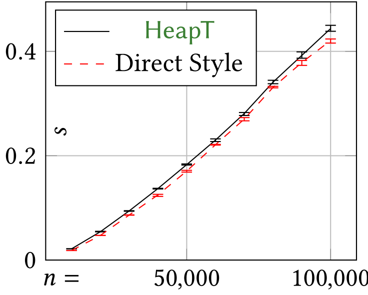
This function works by first applying the monadic filtering function *filterM* to the input list: this creates a search space of the power set of the input list. This is actually very similar to a classic Haskell chestnut for generating the power set of a list (*powerset* = *filterM* (*const* [True, False])), although the version here creates a weighted power set, where smaller sublists are ordered first. This is because the *inclusion* function weights the inclusion of an element with 1, and the exclusion with 0, meaning the filtered list will have a weight equal to its length.

Finally, the function checks that the chosen subset does in fact sum to the target, and returns it.

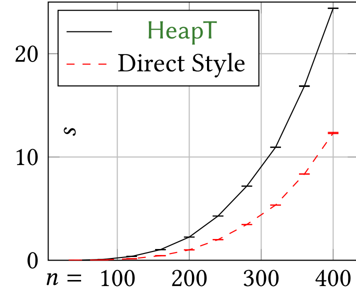
6.1.3 Dijkstra's Algorithm. An important invariant of Dijkstra's algorithm is that the same node is not searched from multiple times, to avoid cycles. The fact that HeapT is a transformer means that it is straightforward to interleave a state effect which maintains this invariant.

```
unique :: Ord a ⇒ a → HeapT w (State (Set a)) a
unique x = do seen ← get; guard (x ∉ seen); modify ({x} ∪); pure x
```

For a given node x, this function *gets* the set of already-seen nodes, checking that x is not a member of that set with *guard*. Then, it updates the set, adding in the new node, and returns it.



(a) Time taken (in seconds) to sort n Integer values using heap sort



(b) Time taken (in seconds) to find all shortest paths from a single node, using Dijkstra's algorithm, on a graph with n vertices with a 10% edge density

Fig. 1. Benchmarks comparing the HeapT monad transformer to a direct-style pairing heap

With the *unique* and *star* functions defined, Dijkstra's algorithm can be implemented as follows:

$$\mathit{dijkstra} :: \mathsf{Ord}\ a \Rightarrow \mathsf{Graph}\ a \rightarrow a \rightarrow [\,(a, \mathsf{Dist})\,]$$
$$\mathit{dijkstra}\ g\ x =$$
$$\quad \mathit{evalState}\ (\mathit{searchT}\ \ (\mathit{star}\ (\mathit{choices}\ (\lambda(x, w) \rightarrow \mathit{tell}\ w \gg \mathit{unique}\ x) \circ g) \lll \mathit{unique}\ x))\ \ \{\}$$

This function takes the Kleene star of a Graph, weighting each edge with the *tell* function, and avoiding duplicates with *unique*. The *searchT* function then enumerates the entire search space, smallest weight first, and finally the state effect is run with *evalState*.

6.2 Performance

So far, an efficient implementation of weighted search algorithms including Dijkstra's algorithm has been developed through compositional and algebraic means. The aim has not been to provide a new implementation of Dijkstra's algorithm, but rather to prove that it is possible to achieve reasonable efficiency in weighted search while presenting a flexible monadic interface and relying on a solid theoretical foundation. That said, the HeapT type does indeed exhibit good performance in practice. Benchmarks confirm that the implementation is within the same complexity class as a direct-style implementation of a pairing heap (Figure 1, completed on a 2016 MacBook Pro with 16GB RAM and a 2.9 GHz Quad-Core Intel Core i7)

When used as a heap directly, for instance in a sorting algorithm, there is no performance degradation when compared to a direct-style pairing heap. On Dijkstra's algorithm, there is a roughly $2x$ slowdown. This is in line with our expectation: the monadic implementation of Dijkstra's algorithm which uses the *star* function does seem to perform two passes over the data when querying a vertex for its neighbours, where a direct style implementation performs only one pass. It may well be possible to remove this extra pass: in fact, as mentioned in Section 5.1, there are in many opportunities for optimisation of the HeapT type; but we will leave exploration of those optimisations to later work. We feel it is most important to demonstrate the optimisation which changes complexity class, and achieves tolerable efficiency; aggressive micro-optimisation may have cluttered the presentation and obscured the theory.

7 RELATED WORK

Expressing nondeterminism through effects is well-trodden ground in functional programming, and in particular in Haskell. The nondeterminism DSL which is now standard in the language has

its roots by Wadler [1985, 1990]. A full API using the monad and monad plus classes, along with an efficient continuation-based implementation, was described in Hinze [1999] (a derivation of the same interface was given in Hinze 2000). All of these approaches are focused on depth-first search: they each basically present an interface to the list monad, albeit with some more efficient continuation-based implementations, or slightly generalised transformer-based interfaces. Our work uses the same interface (monad transformers and the monad plus class), but with a focus on breadth-first and weighted search.

On the problem of breadth-first search, Jones and Gibbons [1993] and Okasaki [2000] give the basic breadth-first tree traversal algorithms which underpin the work we do in Section 2 of this paper. In particular, Gibbons and Jones [1998] examine in depth a number of breadth-first algorithms on the same tree type we examine in Section 2, and they provide an equivalent definition of our efficient implementation of *bfe* in Section 2.5. Our work generalises these algorithms to a reusable monad, then to a monad transformer, and the to other monoidal categories like applicatives.

The first step of that generalisation is the Levels type: an early version of the type can be seen in Spivey [2000], where it is defined as a stream of bags. This was further expanded on in Spivey and Seres [2003], which also introduced the *wrap* function. The name Levels comes from Fischer [2009], which introduced a version of the type where bags were abstracted to any instance of a nondeterminism class. In Section 3 we present the monad transformer version of this type: this type is closely related to LogicT [Kiselyov et al. 2005], although our type is fairer and is a law-abiding monad (modulo the caveats we describe in Section 3).

One problem with the Levels type and its transformer variant is that they are inefficient. In Section 3.2 we use *hyperfunctions* [Launchbury et al. 2000] to improve some of this performance. Hyperfunctions have been used before to implement breadth-first search algorithms: the queues presented in Allison [2006] and Smith [2009] structurally have a lot in common with hyperfunctions; and Berger et al. [2019] also presents a slight variant on hyperfunctions to perform a breadth-first enumeration of a tree. Our use of them generalises the type slightly, allowing us to implement zip fusion on a monadic fold.

Our reinterpretation of Levels as a polynomial takes some inspiration from McIlroy [1999], which describes an implementation of power series in Haskell. More concretely, Rivas and Jaskelioff [2014] and Rivas et al. [2018] provide the theoretical bridge between applicatives and alternatives and polynomials over semirings. The *near*-semirings of their abstraction is replaced by semirings in our case; the extra laws on semirings are precisely what prevent us from defining the free semiring in Haskell, and what motivate the use of Cubical Agda.

Focusing on applicatives, we use some of the free applicative types from Capriotti and Kaposi [2014], and the Phases type from Easterly [2019] to implement our applicative polynomial. We then use the Cayley representation from Rivas et al. [2018] to derive an efficient *bft*.

Turning to weighted search, our semiring and semimodule-based exploration is similar to that presented by Mohri [2002] and especially Höfner and Möller [2012] (though neither considers semimodules). Our approach is more concerned with the categorical understanding of the problem, and in particular the monadic exposition of the algorithms involved.

The monus construction we use was developed almost entirely by Wehrung [1992] and Amer [1984]. Of course our work is more concerned with the constructivist interpretation of the theories presented, but nonetheless what we have presented is simply an application of the theory already described (with regard to monuses specifically).

The free semimodule type we define bears most resemblance to the finite probability monad from Erwig and Kollmansberger [2006]; although our definition is more general.

A more recent innovation in search in functional programming is selection functions [Escardó and Oliva 2010; Hedges 2014]: these give some topological understanding to search algorithms.

There is little overlap with the work we have here, although in the future it may be worth exploring using the selection monad to describe some of the algorithms we provide here.

Finally, on graphs there is work on functional implementations of depth-first search [King and Launchbury 1995] and search generally [Erwig 2001]. Both of these papers have quite a different perspective on search as ours: they describe how to implement efficient graph algorithms in a functional style, rather than our work which seeks to define graph search algorithms as compositions of operations on a monad, while retaining an efficient implementation. Vandenbroucke et al. [2015] takes the same perspective as us in this regard, although they do not present an efficient weighted search algorithm, and we implement duplicate removal through monad transformers, which we consider simpler than their approach. For our efficient implementation we do of course rely heavily on the pairing heap [Fredman et al. 1986], which we derive by applying the optimisation in Jaskelioff and Rivas [2015] to the free monad plus.

8 CONCLUSION

The connection between the free monoid and lists is well established as folklore for functional programmers, and this relationship between abstract and concrete representations has led to important properties and optimisations. This paper has shown that there are similar connections to be found between the free semiring and the Levels type for breadth-first search, and between free semimodules and the Weighted type for weighted search.

Working at a high level of abstraction has made it possible to use the Cayley transformation to enable the implementation of optimised datastructures for these searches, where the Queue type is as an efficienct version of Levels, and the Heap type is an efficient version of Weighted. We have demonstrated that this interface is general enough to capture several important weighted search algorithms, and that the derived implementation is efficient.

In the future, we would like to see exploration into other uses for the Heap type, such as probabilistic parsing. One drawback of our approach is that it does not seem possible to formulate a monus for heuristic-augmented search algorithms, like A*: we would be interested in generalisations of the monus algebra which might allow for the inclusion of such heuristics. Finally, we wonder if there is a generalisation of the Heap type which might be able to facilitate the implementation of a broader class of semiring-based algorithms, not just those that rely on a priority queue. In particular, the optimisation that yielded the Heap type is quite similar in structure to Horner's rule, which can be used to reduce multiplications in the evaluation of polynomials over some semiring.

$$c_0 x^0 + c_1 x^1 + c_2 x^2 + c_3 x^3 + c_4 x^4 \ldots = c_0 + x(c_1 + x(c_2 + x(c_3 + x(c_4 + \ldots))))$$

For the Heap type the semiring in question is the min-plus semiring, but it seems like it should be possible to generalise the type to any semiring. Indeed, there is a class of algorithms (which includes belief propagation, the Viterbi algorithm, and bucket elimination) which seem to perform this very optimisation, i.e. using Horner's rule on some specific semiring to reduce computation. We think it should be possible to develop a data structure which abstracts this pattern, enabling the implementation of semiring algorithms in a similar fashion to how the finger tree [Hinze and Paterson 2006] enables the implementation of monoid-based algorithms.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for providing invaluable criticism, which greatly improved the presentation of this paper. Particular thanks are due to Jeremy Gibbons, who gave insightful feedback. This work is supported by the Engineering and Physical Sciences Research Council, grant number EP/S028129/1 on "SCOPE: Scoped Contextual Operations and Effects".

REFERENCES

Lloyd Allison. 2006. Circular Programs and Self-Referential Structures. *Software: Practice and Experience* 19, 2 (Oct. 2006), 99–109. https://doi.org/10.1002/spe.4380190202

Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. 2011. Definable Quotients in Type Theory. (2011). http://www.cs.nott.ac.uk/~psztxa/publ/defquotients.pdf

K. Amer. 1984. Equationally Complete Classes of Commutative Monoids with Monus. *Algebra Universalis* 18, 1 (Feb. 1984), 129–131. https://doi.org/10.1007/BF01182254

Ulrich Berger, Ralph Matthes, and Anton Setzer. 2019. Martin Hofmann's Case for Non-Strictly Positive Data Types. In *24th International Conference on Types for Proofs and Programs (TYPES 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 130)*, Peter Dybjer, José Espírito Santo, and Luís Pinto (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22. https://doi.org/10.4230/LIPIcs.TYPES.2018.1

Paolo Capriotti and Ambrus Kaposi. 2014. Free Applicative Functors. *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), 2–30. https://doi.org/10.4204/EPTCS.153.2 arXiv:1403.0749

A. Cayley. 1854. *On the Theory of Groups as Depending on the Symbolic Equation $þeta$n= 1 Phil.* Mag.

Keith L. Clark and Sten-Åke Tärnlund. 1977. A First Order Theory of Data and Programs. In *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*, Bruce Gilchrist (Ed.). North-Holland, 939–944.

Noah Easterly. 2019. Functions and Newtype Wrappers for Traversing Trees: Rampion/Tree-Traversals. https://github.com/rampion/tree-traversals

Martin Erwig. 2001. Inductive Graphs and Functional Graph Algorithms. *J. Funct. Program.* 11, 5 (Sept. 2001), 467–492. https://doi.org/10.1017/S0956796801004075

Martin Erwig and Steve Kollmansberger. 2006. FUNCTIONAL PEARLS: Probabilistic Functional Programming in Haskell. *J. Funct. Prog.* 16, 1 (Jan. 2006), 21–34. https://doi.org/10.1017/S0956796805005721

Martín Escardó and Paulo Oliva. 2010. Selection Functions, Bar Recursion and Backward Induction. *Mathematical Structures in Computer Science* 20, 2 (April 2010), 127–168. https://doi.org/10.1017/S0960129509990351

Sebastian Fischer. 2009. Reinventing Haskell Backtracking. In *Informatik 2009, Im Fokus Das Leben (ATPS'09)*. GI Edition, 15. http://www-ps.informatik.uni-kiel.de/~sebf/data/pub/atps09.pdf

Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. 1986. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica* 1, 1-4 (Jan. 1986), 111–129. https://doi.org/10.1007/BF01840439

Jeremy Gibbons and Geraint Jones. 1998. The Under-Appreciated Unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. Association for Computing Machinery, Baltimore, Maryland, USA, 273–279. https://doi.org/10.1145/289423.289455

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

Jonathan S. Golan. 2003. Semirings. In *Semirings and Affine Equations over Them: Theory and Applications*, Jonathan S. Golan (Ed.). Springer Netherlands, Dordrecht, 1–26. https://doi.org/10.1007/978-94-017-0383-3_1

Jules Hedges. 2014. Monad Transformers for Backtracking Search. *Electron. Proc. Theor. Comput. Sci.* 153 (June 2014), 31–50. https://doi.org/10.4204/EPTCS.153.3 arXiv:1406.2058

Ralf Hinze. 1999. Efficient Monadic-Style Backtracking. (Sept. 1999), 51.

Ralf Hinze. 2000. Deriving Backtracking Monad Transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montréal, Canada, September 18-21, 2000*, Vol. 35. ACM, 186. https://karczmarczuk.users.greyc.fr/TEACH/Doc/Hinze_back.pdf

Ralf Hinze and Ross Paterson. 2006. Finger Trees: A Simple General-Purpose Data Structure. *Journal of Functional Programming* 16, 2 (2006), 197–217. http://www.staff.city.ac.uk/~ross/papers/FingerTree.html

Peter Höfner and Bernhard Möller. 2012. Dijkstra, Floyd and Warshall Meet Kleene. *Formal Aspects of Computing* 24, 4-6 (July 2012), 459–476. https://doi.org/10.1007/s00165-012-0245-4

R. John Muir Hughes. 1986. A Novel Representation of Lists and Its Application to the Function "Reverse". *Inform. Process. Lett.* 22, 3 (March 1986), 141–144. https://doi.org/10.1016/0020-0190(86)90059-1

Graham Hutton, Mauro Jaskelioff, and Andy Gill. 2010. Factorising Folds for Faster Functions. *Journal of Functional Programming* 20, 3-4 (July 2010), 353–373. https://doi.org/10.1017/S0956796810000122

Nathan Jacobson. 1985. *Basic Algebra I* (2nd ed ed.). W.H. Freeman, New York.

Mauro Jaskelioff and Eugenio Moggi. 2010. Monad Transformers as Monoid Transformers. *Theoretical Computer Science* 411, 51 (Dec. 2010), 4441–4466. https://doi.org/10.1016/j.tcs.2010.09.011

Mauro Jaskelioff and Exequiel Rivas. 2015. Functional Pearl: A Smart View on Datatypes. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 355–361. https://doi.org/10.1145/2784731.2784743

Geraint Jones and Jeremy Gibbons. 1993. *Linear-Time Breadth-First Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips*. Technical Report 71. Dept of Computer Science, University of Auckland. http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/linear.ps.gz

Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Johan Jeuring, and Erik Meijer (Eds.). Vol. 925. Springer Berlin Heidelberg, Berlin, Heidelberg, 97–136. https://doi.org/10.1007/3-540-59451-5_4

Donnacha Oisín Kidney and Nicolas Wu. 2021. Supporting Code for Algebras for Weighted Search. Zenodo. https://doi.org/10.5281/zenodo.4774319

David J. King and John Launchbury. 1995. Structuring Depth-First Search Algorithms in Haskell. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '95*. ACM Press, San Francisco, California, United States, 344–354. https://doi.org/10.1145/199448.199530

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 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, 192–203. https://doi.org/10.1145/1086365.1086390

Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. 2013. A Back-to-Basics Empirical Study of Priority Queues. In *2014 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, 61–72. https://doi.org/10.1137/1.9781611973198.7

John Launchbury, Sava Krstic, and Timothy E. Sauerwein. 2000. *Zip Fusion with Hyperfunctions*. Technical Report. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.4961

Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (Jan. 2008), 1–13. https://doi.org/10.1017/S0956796807006326

M. Douglas McIlroy. 1999. Power Series, Power Serious. *J. Funct. Program.* 9, 3 (May 1999), 325–337. https://doi.org/10.1017/S0956796899003299

Lambert Meertens. 1992. Paramorphisms. *Formal Aspects of Computing* 4, 5 (Sept. 1992), 413–424. https://doi.org/10.1007/BF01211391

Mehryar Mohri. 2002. Semiring Frameworks and Algorithms for Shortest-Distance Problems. *J. Autom. Lang. Comb.* 7, 3 (Jan. 2002), 321–350. http://dl.acm.org/citation.cfm?id=639508.639512

Chris Okasaki. 2000. Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 131–136. https://doi.org/10.1145/351240.351253

Ross Paterson. 2012. Constructing Applicative Functors. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*, Jeremy Gibbons and Pablo Nogueira (Eds.). Springer Berlin Heidelberg, 300–323.

Maciej Piróg. 2016. Eilenberg–Moore Monoids and Backtracking Monad Transformers. *Electron. Proc. Theor. Comput. Sci.* 207 (April 2016), 23–56. https://doi.org/10.4204/EPTCS.207.2 arXiv:1604.01184

Exequiel Rivas and Mauro Jaskelioff. 2014. Notions of Computation as Monoids. *arXiv:1406.4823 [cs, math]* (May 2014). arXiv:1406.4823 [cs, math] http://arxiv.org/abs/1406.4823

Exequiel Rivas, Mauro Jaskelioff, and Tom Schrijvers. 2018. A Unified View of Monadic and Applicative Non-Determinism. *Science of Computer Programming* 152 (Jan. 2018), 70–98. https://doi.org/10.1016/j.scico.2017.09.007

Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (July 1985), 652–686. https://doi.org/10.1145/3828.3835

Leon P Smith. 2009. Lloyd Allison's Corecursive Queues: Why Continuations Matter. *The Monad.Reader* 14, 14 (July 2009), 28. https://meldingmonads.files.wordpress.com/2009/06/corecqueues.pdf

J. Michael Spivey. 2009. Algebras for Combinatorial Search. *Journal of Functional Programming* 19, 3-4 (July 2009), 469–487. https://doi.org/10.1017/S0956796809007321

J. Michael Spivey and Silvija Seres. 2003. Combinators for Logic Programming. In *The Fun of Programming*. Palgrave. https://www.cs.ox.ac.uk/publications/books/fop/dist/fop/chapters/9/Logic.hs

Michael Spivey. 2000. Combinators for Breadth-First Search. *Journal of Functional Programming* 10, 4 (July 2000), 397–408. https://www.cambridge.org/core/journals/journal-of-functional-programming/article/combinators-for-breadthfirst-search/60383337C85657F3F6549C18F4E345BA

Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. 2015. Fixing Non-Determinism. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages - IFL '15*. ACM Press, Koblenz, Germany, 1–12. https://doi.org/10.1145/2897336.2897342

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 87:1–87:29. https://doi.org/10.1145/3341691

Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57. https://doi.org/10.1145/3147.3165

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction (Lecture Notes in Computer Science, Vol. 5133)*. Springer, Berlin, Heidelberg, 388–403. https://doi.org/10.1007/978-3-540-70594-9_20

Philip Wadler. 1985. How to Replace Failure by a List of Successes: A Method for Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*. 113–128. https://doi.org/10.1007/3-540-15975-4_33

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. Association for Computing Machinery, New York, NY, USA, 61–78. https://doi.org/10.1145/91556.91592

Friedrich Wehrung. 1992. Injective Positively Ordered Monoids I. *Journal of Pure and Applied Algebra* 83, 1 (Nov. 1992), 43–82. https://doi.org/10.1016/0022-4049(92)90104-N