Formalising Graph Algorithms with Coinduction

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

Graphs and their algorithms are fundamental to computer science, but they can be difficult to formalise, especially in dependently-typed proof assistants. Part of the problem is that graphs aren't as well-behaved as inductive data types like trees or lists; another problem is that graph algorithms (at least in standard presentations) often aren't structurally recursive. Instead of trying to find a way to make graphs behave like other familiar inductive types, this paper builds a formal theory of graphs and their algorithms where graphs are treated as coinductive structures from the beginning. We formalise our theory in Agda.

This approach has its own unique challenges: Agda is more comfortable with induction than coinduction. Additionally, our formalisation relies on quotient types, which tend to make coinduction even harder to deal with. Nonetheless, we develop reusable techniques to deal with these difficulties, and the simple graph representation at the heart of our work turns out to be flexible, powerful, and formalisable.

1 Introduction

Some data structures are easier to formalise than others. Generally speaking, especially in the dependently-typed world, a formalisation effort will be more pleasant if (1) everything involved is inductive, (2) there is nothing that needs quotienting, and (3) algorithmic efficiency is of no concern. Graphs fail on all three counts.

There are a few ways to overcome these hurdles: in the functional programming world, huge strides have been made by representing graphs as inductive data types [Erwig 2008; Gibbons 1995] or with typeclasses [Mokhov 2017]. Specifically treating graphs as matrices has also proved useful, especially in elucidating the link between semirings and search algorithms [Backhouse and Carré 1975; Conway 1971; Dolan 2013; Master 2021; Rivas et al. 2015].

We take an alternative route: our graph representation is fundamentally coinductive, based on a generalisation of adjacency lists. In this paper, a directed weighted graph with vertices of type V is a function from a vertex to a weighted set of its neighbours.

$$\begin{aligned}&\mathit{GraphOf} : \mathit{Type} \rightarrow \mathit{Type}\\&\mathit{GraphOf}\ V = V \rightarrow \mathit{Neighbours}\ V\end{aligned} \tag{1}$$

Though simple, this representation is powerful and is supported by a deep theoretical foundation.

We will leave the *Neighbours* type abstract for now: it represents weighted sets that are not necessarily finite. A value $\mathit{graph} : \mathit{GraphOf}\ V$ is a graph with vertices of type V: an example is given in Fig. 1. The neighbours of a, for instance, are given by:

$$\mathit{graph}\ a = \wr\ 7 \triangleright b\ , 2 \triangleright c\ \int$$

This line says that a has two outward edges, $a \mapsto b$ and $a \mapsto c$, with weights 7 and 2, respectively. In this example, the weights are drawn from $\mathbb{N}$, but our construction is generic over a large class of weights, which will be characterised in Section 2.1.

The rest of this paper will be devoted to exploring this representation and examining its practical and theoretical aspects. We start by implementing a standard graph algorithm: finding Hamiltonian paths. A Hamiltonian path is one that visits every vertex in a graph exactly once. In this paper, graph

Authors' Contact Information: 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.

2024. ACM 2475-1421/2024/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
data Vert : Type where
  a b c d : Vert

graph : GraphOf Vert
graph a = ⟅ 7 ▹ b , 2 ▹ c ⟆
graph b = ⟅ 1 ▹ c ⟆
graph c = ⟅ 3 ▹ d , 1 ▹ b ⟆
graph d = ⟅ 5 ▹ b ⟆
```

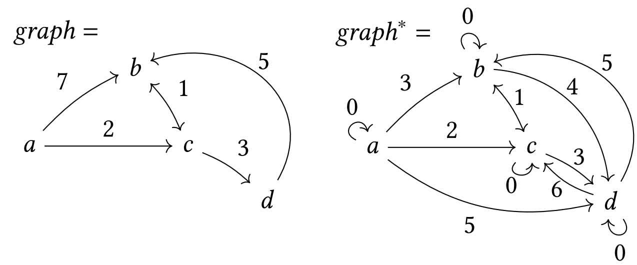


Fig. 1. A weighted graph with four vertices and its transitive closure

algorithms will be reframed as graph *transformations*. Accordingly, the algorithm for computing Hamiltonian paths for a finite type V is as follows:

$$\begin{aligned}&\mathit{hamiltonian} : \mathit{GraphOf}\ V \to \mathit{GraphOf}\ (\mathit{List}^{+}\ V)\\ &\mathit{hamiltonian}\ g = (\mathit{pathed}\ g \ggg \mathit{filtering}\ \mathit{uniq})^{*} \ggg \mathit{filtering}\ \mathit{covers}\end{aligned} \tag{2}$$

The expression *hamiltonian g* produces a graph whose vertices contain the Hamiltonian paths in g (List^{+} is the type of non-empty snoc lists). Taking the graph in Fig. 1 as an example, to extract a concrete collection of Hamiltonian paths we apply *hamiltonian graph* to some starting point:

$$\mathit{hamiltonian}\ \mathit{graph}\ [\ a\ ] \nrightarrow 15 \equiv ⟅\ 11 \triangleright [\ a\ ,\ b\ ,\ c\ ,\ d\ ]\ ,\ 10 \triangleright [\ a\ ,\ c\ ,\ d\ ,\ b\ ]\ ⟆$$

The $\nrightarrow$ operator restricts the output of search to a particular depth. So the expression above says that there are two Hamiltonian paths (in *graph*) with weight less than 15 that start at the vertex a: $[a, b, c, d]$, and $[a, c, d, b]$, with weights 11 and 10, respectively.

Algorithms to compute Hamiltonian paths are often complex. However, our implementation is simple, built out of small, algebraic components. For more detail on these components see Section 3.

The first component is the $*$ operator, which computes transitive closure. Fig. 1 contains a diagram of its use: graph^{*} is a graph where every vertex has an edge to every reachable vertex, with a weight equal to the sum of the weights on the shortest path to that vertex. For instance, there is an edge $(a \mapsto d) \in \mathit{graph}^{*}$ with weight 5, constructed from the path $a \mapsto c \mapsto d$ (note that in our formalisation, $*$ is not called directly on a graph, but rather on its *ideal*, as explained in Section 5.4).

Most of the algorithmic "work" is done by the $*$ function; the rest of the implementation is book-keeping and filtering. The *pathed* function, for instance, tags every vertex with a list representing the path taken to reach that vertex. The $\ggg$ operator connects graphs: here we use it in combination with *filtering* to filter the output of the algorithm, where $g \ggg \mathit{filtering}\ p$ basically filters the vertices of g according to some predicate p. We use the predicate *uniq* to remove paths with loops, and *covers* to restrict the output to only those paths that hit every vertex in the graph.

Finding Hamiltonian paths is an NP-complete problem, and the *hamiltonian* function presented here is not particularly optimised. However, there is nothing inherently slow about this graph representation or the combinators used: the core algorithmic step of the *hamiltonian* function, $*$, performs a simple breadth-first search in $O(n)$ when an efficient representation of *Neighbours* is used (Section 3.4). Finally, perhaps surprisingly, this core step ($*$) is fundamentally *coinductive*.

Let us take a moment to explain why coinduction is central to our approach. The obvious advantage of working with a coinductive representation is that we can work with graphs that have infinitely many vertices. For instance, finding all the Collatz sequences of length 5 or less is easy:

```
collatz : GraphOf ℕ
collatz 0          = ⦃⦄
collatz (suc n) =
  if suc n mod 6 ≡ᴺ 4
    then ⦃ 1 ▹ 2 * suc n , 1 ▹ n div 3 ⦄
    else ⦃ 1 ▹ 2 * suc n ⦄
```

(3)

```
((pathed collatz ⋙ filtering uniq)*) [ 1 ] ⋇ 5 ≡
  ⦃ 0 ▹ [ 1 ] , 1 ▹ [ 1 , 2 ] , 2 ▹ [ 1 , 2 , 4 ]
  , 3 ▹ [ 1 , 2 , 4 , 8 ] , 4 ▹ [ 1 , 2 , 4 , 8 , 16 ]
  , 5 ▹ [ 1 , 2 , 4 , 8 , 16 , 32 ]
  , 5 ▹ [ 1 , 2 , 4 , 8 , 16 , 5 ] ⦄
```

However, even when a graph has finitely many vertices, we would argue that traditional graph algorithms are usually not *inductive* in any real sense. A standard textbook exposition of depth-first search will not present a structurally recursive fold, but instead will describe an iterative algorithm that repeatedly expands a search until some condition is met. And while the termination condition might rely on the finiteness of vertices in a textbook, real-world implementations of such algorithms often use some "cutoff" based on time or distance ("stop searching after x seconds/steps").

It is our view that this style of algorithm deserves to be formalised: indeed, if we ever hope to formalise real software that works with graphs, the theory for how that software works needs to be established. This paper is a step towards establishing that theory.

Structure of this Paper

This paper is about the semantics and implementation of graphs and graph algorithms. Our focus is on the representation given in Eq. (1): though simple, this representation is flexible, powerful, and amenable to formalisation. We discuss the representation in detail in Section 2.

Coinduction is central to our representation: in Section 4 we explore the formal underpinnings of coinduction for our graph representation, and present a number of coinductive structures that can be used to work with (possibly infinite) graphs in a well-founded way.

Finally, Section 5 addresses the issue of combining quotients and coinduction in the context of graph algorithms. This is a well-known pain point in dependently-typed programming languages: this section presents a few approaches, culminating in the *Neighbours* type, a monad that can be used to represent well-founded coinductive graph algorithms.

Along these lines, we make the following contributions:

- We present a fully formalised representation of graphs that can handle infinite graphs and coinductive algorithms (Section 2). This representation uses quotients to equate graphs that differ only by, for instance, the order of their vertices.
- We present two semirings on graphs that are used to implement and structure various graph algorithms (Sections 3.1 and 3.5).
- We prove that the *Weighted* type is the free weight semimodule, and use this to present an optimisation of the algorithms implemented using semirings (Section 3.4).
- We implement a productive, coinductive version of the pairing heap, based on the cofree comonad, and use it to implement a search algorithm (Section 4.1).
- We present an application of completely iterative monads (CIMs) to the problem of graph search (Section 4.3), and formalise a new guardedness condition that allows this (Lemma 4.1).
- We present the *Bush* type, a quotiented version of the pairing heap (Section 5.2).
- We present the *Neighbours* type, a construction based on semigroup actions that is a monad, a monoid, and a CIM, and can represent coinductive graph algorithms as graph transformations (Section 5.3).

Our formalisation provided in the supplementary materials is in Cubical Agda [Vezzosi et al. 2021], giving us access to univalence, quotients, and Cubical Type Theory. We use quotients extensively in this paper, but our only real use of univalence is in Section 3.5. Outside of Section 3.5, then, the formalisation of this paper can be thought of as "Martin-Löf Type Theory (MLTT) with quotients".

The formalisation of this work is available to download from https://github.com/oisdk/formalising-graph-algorithms-with-coinduction. Every code block in this paper is hyperlinked to the source of that block online, rendered and highlighted and accessible from a browser without installing Agda. It is also possible to browse the source code alongside the paper by following https://oisdk.github.io/formalising-graph-algorithms-with-coinduction/README.html, which is a file organising the source to follow the structure of this paper.

2 Representing Graphs

Let's now look at the graph representation in a little more detail. This section will give a formal account of the types involved: first, we will describe the algebra that defines weights in a graph (Section 2.1), and then we will describe the data structure used to represent the neighbours of a vertex (Section 2.2). This section defines the inductive variant of weighted sets, later we will construct the coinductive variant (Sections 4 and 5).

2.1 Weights

The edges of the graph in Fig. 1 are each tagged with a *weight*. In this case, the weight is a simple natural number, but the framework we define in this paper actually works with a more general construction, based on the *monus* operation, written as $\dot{-}$, which is a notion of subtraction with truncation [Amer 1984]. For instance on $\mathbb{N}$: $5 \dot{-} 3 = 2$, and $3 \dot{-} 5 = 0$. To give its general definition, we must establish some of the algebraic structure that should exist on weights.

First, weights are monoidal. This is essential for being able to talk about paths through a graph: the path $a \mapsto c \mapsto d$ has a weight equal to the *sum* of the constituent edges (5, in this case). The binary operator on this monoid will be denoted with $\bullet$; in the case of the natural-number weights on the graph in Fig. 1, this operator is instantiated to $+$. A neutral element, ϵ, denotes the weight of the identity path, the path from a vertex to itself. Again on natural numbers this neutral element is instantiated to 0. Finally, the laws of associativity and identity follow from the expected behaviour of the paths: going from $(a \mapsto b \mapsto c) \mapsto d$ should have the same weight as $a \mapsto (b \mapsto c \mapsto d)$.

Secondly, weights should be ordered. The particular order we will use is the *algebraic preorder*:

$$x \leqslant y = \exists\, z \times (y \equiv x \bullet z)$$

This says that x is ordered before y iff there is some weight z that can be added to x to get to y.

We will insist that this relation forms a total order. The relation is automatically reflexive and transitive (these follow from the monoid laws), so this requirement amounts to the relation being antisymmetric and connected. Antisymmetry actually rules out *groups*, since $x \leqslant y$ for all x and y in the presence of additive inverses ($z = x^{-1} \bullet y$; $y = x \bullet z$). In fact, every monus is the positive cone of some group (the cone of a group is the monoid generated by taking the non-negative elements of that group; $(\mathbb{N}, +, 0)$ is the cone of $(\mathbb{Z}, +, 0)$).

In Agda, the statement "$\leqslant$ is connected" translates to the existence of the following function:

$$\_{\leqslant}|{\geqslant}\_ : \forall\; x\; y \rightarrow (x \leqslant y) \uplus (y \leqslant x)$$

From this function we can extract an implementation of the monus operator:

$$x \dot{-} y = (\mathit{const}\ \epsilon \mathbin{\triangledown} \mathit{fst})\ (x \leqslant|\geqslant y)$$

The operator $\triangledown$ is shorthand for *either*; and so $x \dot{-} y$ is z when $x = y \bullet z$, and constantly ϵ otherwise.

Finally, some later proofs will rely on the commutativity of $\bullet$, so we add this to the definition:

Definition 2.1 (Monus). A monus [Amer 1984] is a commutative monoid $(S, \bullet, \epsilon)$ such that the algebraic preorder is antisymmetric and connected.

The monoid of addition on natural numbers is a simple example of a monus, but we also have other positive cones of groups ($\mathbb{Q}^{+}$, etc.). Probability also forms a monus, albeit with a slightly strange preorder. The monoid is $(\mathbb{P}, \times, 1)$ (where the carrier set $\mathbb{P}$ is the interval of rationals $[0, 1]$), and the order is given by $x \leqslant y$ when x is *more likely* than y. Unweighted graphs are also supported by our framework: the trivial weight, $\top$, makes weighted sets degrade to simple finite sets.

2.2 Weighted Sets

Now that we have established the algebra for weights in a graph, we will define precisely the data structure that describes the neighbours of a vertex: the weighted set.

data *Weighted A* **where**

A value of type *Weighted A* is a weighted set of *A*s, where the weight is given by some type *S*. Note that this is *not* the same type as *Neighbours* (which will be fully introduced in Section 5.3): although a graph representation based on *Weighted* alone would allow the expression of infinite graphs, they would be restricted to finite *breadth* since *Weighted* represents finite weighted sets.

We will define this type using quotients: the particular implementation of quotients we will use is based on Higher-Inductive types (HITs) [Univalent Foundations Program 2013, chapter 5]. All of these constructions are formalised in Cubical Agda [Vezzosi et al. 2021].

Higher-inductive data types are defined not just by *point* constructors but also by *path* constructors. Point constructors are the “normal” data constructors for a type; they define how to construct values (points) of the type. For the weighted set, they are as follows:

$$\begin{aligned}&\wr\,\int &&: \mathit{Weighted}\ A\\ &\_\triangleright\_{::}\_ &&: (p : S)\ (x : A)\ (xs : \mathit{Weighted}\ A) \to \mathit{Weighted}\ A\end{aligned}$$

A weighted set is either an empty set, $\wr\int$, or a pair of a weight p and value x added to a set xs. Notice that this type has the same structure as a list of pairs of *A*s and *S*s.

The path constructors, on the other hand, specify equalities that hold on the type. For our purposes it is sufficient to think of them as ways to quotient a type (however, HITs are more general than just set quotients). The *Weighted* type has three path constructors, the first of which is:

$$\mathit{com} : \forall\ p\ x\ q\ y\ xs \to p \triangleright x :: q \triangleright y :: xs \equiv q \triangleright y :: p \triangleright x :: xs$$

This constructor says that the order of values in a weighted set don’t matter; two sets whose contents are permutations of each other should be regarded as equal. For instance, this constructor says that the edges of a in Fig. 1 could have been specified in any order (modulo syntactic sugar):

$$\mathit{com}\ 7\ b\ 2\ c\ \wr\int : \wr\ 7 \triangleright b\ , 2 \triangleright c\ \int \equiv \wr\ 2 \triangleright c\ , 7 \triangleright b\ \int$$

The next constructor specifies how to deal with key collision:

$$\mathit{dup} : \forall\ p\ q\ x\ xs \to p \triangleright x :: q \triangleright x :: xs \equiv p \sqcap q \triangleright x :: xs$$

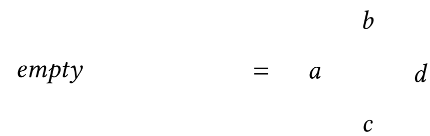
If the same key—x in this example—is present twice in a weighted set, we take the *minimum* of the two corresponding weights ($p \sqcap q$). In the context of graphs, this means that if there is more than one edge between two vertices, we will ignore all but the least-weight edge.

The final constructor, *trunc*, makes all equalities on the *Weighted* type equal:

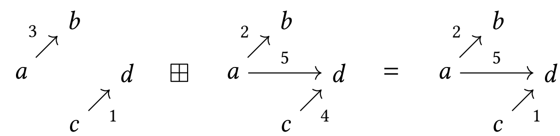
$$\mathit{trunc} : \forall\ xs\ ys\ (p\ q : xs \equiv ys) \to p \equiv q$$

Table 1. The Overlay Monoid on Weighted Graphs

$empty : GraphOf\ A$
$empty\ \_ = \wr\!\int$



$\_\boxplus\_ : GraphOf\ A \rightarrow$
$\quad GraphOf\ A \rightarrow$
$\quad GraphOf\ A$
$(f \boxplus g)\ v = f\ v \cup g\ v$



The full technical details of what's going on here are beyond the scope of this paper, but basically this constructor collapses the higher homotopy structure of *Weighted*, making behave like an MLTT type, rather than like one of the more exotic types available in HoTT.

With this definition of weighted sets, an initial implementation of our graph type becomes:

$$GraphOf\ V = V \rightarrow Weighted\ V \tag{4}$$

This type differs from our *GraphOf* type in the introduction only by the replacement of *Neighbours* with *Weighted*. This does not make this graph type finite, or inductive (for example, *collatz* from Eq. (3) is expressible in this form). The difference is that the neighbours of a vertex may not be expressed coinductively, which means that the definition of $*$ is not productive.

3 Algebraic Graphs

There is a natural algebraic structure on graphs that can be used as a language for expressing compositional graph algorithms, as we did with Hamiltonian paths. This algebraic structure is a *semiring*: commonly used as an abstraction in search and optimisation algorithms [Backhouse 1975; Backhouse and Carré 1975; Backhouse et al. 1994; Conway 1971; Dolan 2013; Master 2021; Rivas et al. 2015], here semirings will describe the ways that graphs can be combined.

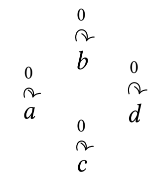
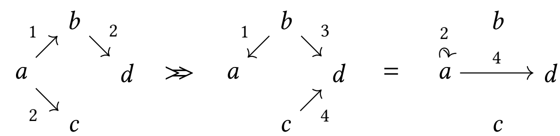
Definition 3.1 (Semiring). A semiring $(S, \oplus, \mathbb{0}, \otimes, \mathbb{1})$ is a commutative monoid, $(S, \oplus, \mathbb{0})$, and a monoid $(S, \otimes, \mathbb{1})$, such that the following laws are obeyed:

$$(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z) \qquad x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z) \qquad \mathbb{0} \otimes x = x \otimes \mathbb{0} = \mathbb{0}$$

The booleans form a semiring in a straightforward way ($Bool, \lor, false, \land, true$), as do the naturals $(\mathbb{N}, +, 0, \times, 1)$. Regular languages also form a semiring, and entire nondeterministic programs can be semirings, under disjunction and conjunction (which underlies many of the programs in Dolan [2013]). The graph semirings that we will look at also behave like conjunction and disjunction.

The edge semiring (Section 3.1) describes the combination of the edges of a graph; we will use it to implement algorithms including transitive closure (Section 3.2)—recall that we used this to find Hamiltonian paths (Section 1). Then, we will look at optimising the representation of the graph (Section 3.4). By implementing algorithms using only the semiring abstraction, we can optimise aggressively without changing the semantics of the algorithms. Finally, we will look at the vertex semiring (Section 3.5), which structures the combination of vertices in a graph.

Table 2. The Connection Monoid on Weighted Graphs

$\mathit{return} : \mathit{GraphOf}\ A$	return = 
$\_\gg\!\!=\_ : \mathit{GraphOf}\ A \to \mathit{GraphOf}\ A \to \mathit{GraphOf}\ A$	

$f\ a = ⦃\, 1 \triangleright b\,,\, 2 \triangleright c\, ⦄$
$f\ b = ⦃\, 2 \triangleright d\, ⦄$
$f\ \_ = ⦃⦄$

$g\ b = ⦃\, 1 \triangleright a\,,\, 3 \triangleright d\, ⦄$
$g\ c = ⦃\, 4 \triangleright d\, ⦄$
$g\ \_ = ⦃⦄$

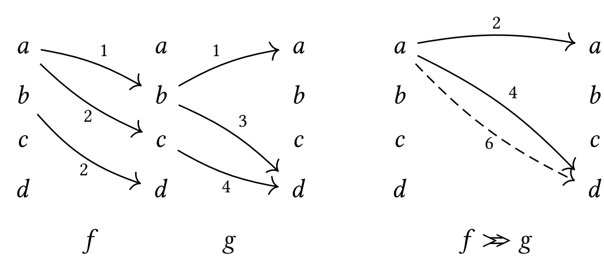


Fig. 2. The connection operator

3.1 The Edge Semiring

The *edge semiring* on graphs ($\mathit{GraphOf}\ V, \boxplus, \mathit{empty}, \ggg, \mathit{return}$) is particularly useful because it allows us to define some search algorithms. The constituent monoids are the *overlay monoid* ($\mathit{GraphOf}\ V, \boxplus, \mathit{empty}$), and the *connection monoid* ($\mathit{GraphOf}\ V, \ggg, \mathit{return}$).

The overlay monoid, which is defined in Mokhov [2017], consists of a binary operator $\boxplus$ and an identity *empty*. The binary operator $\boxplus$ takes the union of the edges of its operands, taking the minimum weight for overlapping edges, and the identity is the graph with no edges: these operations are diagrammed in Table 1.

The connection monoid is a little more complex (Table 2). The binary operator on this monoid, $\ggg$, connects corresponding edges, where the resulting weight of the new edge is given by the minimum sum of the constituent edges. The identity, *return*, is a graph where every vertex has an ϵ-weighted path to itself. Be careful to not be confused by the types here: *return* takes a vertex and returns a weighted set ($\mathit{return} : V \to \mathit{Weighted}\ V$). It does not take a vertex and return a graph. Recalling our definition of graphs ($\mathit{GraphOf}\ A = A \to \mathit{Weighted}\ A$, Eq.(4)), we can see that *return*'s type means it *is* a graph ($\mathit{return} : \mathit{GraphOf}\ V$).

Visualising how these operators work may help us understand them better. We can take the two graphs in Table 2, call them f and g, and redraw them in Fig. 2 as a bipartite graph where the vertices are duplicated. This form makes it easier to see what the connection operator is doing: the edges $a \mapsto b$ and $b \mapsto d$ are connected, and the resulting edge has a weight equal to their sum. There is also another edge $a \mapsto d$, but its weight is larger (6), so it is discarded.

The connection monoid is derived from the monad instance on weighted sets: in fact, the connection monoid is specifically the endomorphism monoid on the Kleisli category of weighted sets. Here we give the implementation of *return* and, for intuition, pseudocode for $\gg\!=$:

$$return\ x = \wr \epsilon \triangleright x \int \qquad xs \gg\!= k = \wr ((v \bullet w) \triangleright y) | (v \triangleright x) \in xs, (w \triangleright y) \in k\ x \int$$

The *return* is simple: it produces a singleton set with the empty weight. For every entry $v \triangleleft x$ in xs, the expression $xs \gg\!= k$ applies the continuation k to x, adds the weight v to the output, and concatenates the results. The actual code is not too much more complicated:

$$\begin{aligned} \wr\int \quad &\gg\!= k = \wr\int \\ (p \triangleright x :: xs) &\gg\!= k = (p \rtimes k\ x) \cup (xs \gg\!= k) \end{aligned} \qquad \begin{aligned} w \rtimes \wr\int \quad &= \wr\int \\ w \rtimes p \triangleright x :: xs &= w \bullet p \triangleright x :: w \rtimes xs \end{aligned} \tag{5}$$

$w \rtimes xs$ adds the weight w to every entry in xs. $\cup$ takes the union of two weighted sets. Our formalisation contains the proofs that show these functions respect the quotients on weighted sets.

The edge semiring and the vertex semiring (Section 3.5) can both be found in the Arrow library [Hughes 2000], where they are defined on Kleisli arrows for a MonadPlus.

3.2 Search Algorithms with the Edge Semiring

Under our framework, graph algorithms are graph *transformations*. In particular, some search algorithms can be expressed as variants of transitive closure. Transitive closure on graphs is diagrammed in Fig. 1. The transitive closure of *graph* is $graph^*$, where the weight of the edge $x \mapsto y$ in $graph^*$ is equal to the weight of the *shortest path* from x to y in *graph*.

Transitive closure has a natural interpretation as the Kleene star, where the Kleene star is an operator $*$ that satisfies both of the following equations for a semiring $(S, \oplus, \mathbb{0}, \otimes, \mathbb{1})$:

$$x^* \equiv \mathbb{1} \oplus (x^* \otimes x) \qquad\qquad x^* \equiv \mathbb{1} \oplus (x \otimes x^*)$$

On graphs, this operator could be naively implemented by simply copying either equation:

$$\begin{aligned} &\_^* : GraphOf\ A \to GraphOf\ A \\ &g^* = \mathbb{1} \oplus (g^* \otimes g) \end{aligned} \tag{6}$$

$$\begin{aligned} &\_^{*\prime} : GraphOf\ A \to GraphOf\ A \\ &g^{*\prime} = \mathbb{1} \oplus (g \otimes g^{*\prime}) \end{aligned} \tag{7}$$

Using the first definition Eq. (6) yields a function that computes, and the resulting weighted set does contain the length of the relevant shortest paths (if treated lazily):

$$\begin{aligned} graph^*\ a &\equiv \wr 0 \triangleleft a, 7 \triangleleft b, 2 \triangleleft c, 8 \triangleleft c, 5 \triangleleft d, 3 \triangleleft b, \dots \int \quad \text{(after applying the } \textit{Weighted} \text{ quotients)} \\ &\equiv \wr 0 \triangleleft a, 3 \triangleleft b, 2 \triangleleft c, 5 \triangleleft d, \dots \int \end{aligned}$$

Notice that this function traverses the graph in breadth-first order. Interestingly, the alternative implementation Eq. (7) traverses the graph in *depth*-first order.

Another search algorithm, iterative-deepening search, can be implemented using exponentiation.

$$\begin{aligned} &exp : S \to \mathbb{N} \to S \\ &exp\ x\ zero \quad = \mathbb{1} \\ &exp\ x\ (suc\ n) = x \otimes exp\ x\ n \end{aligned} \qquad \begin{aligned} &exp\ graph\ 0\ a \equiv \wr 0 \triangleleft a \int \\ &\qquad \cdots \\ &exp\ graph\ 3\ a \equiv \wr 9 \triangleleft b, 11 \triangleleft d, 4 \triangleleft c, 10 \triangleleft b \int \end{aligned}$$

The expression $exp\ x\ n$ on monoids is n copies of x multiplied with itself, on *graphs* it is a graph constructed of all the paths of length n in x. For some graph g, the expression $exp\ g\ 3\ v$ will return a list of vertices 3 steps from v. Iterative-deepening search (*ids*) involves searching successively deeper into a graph; using exponentiation this can be expressed as follows:

$$ids\ g = exp\ g\ 0 \oplus exp\ g\ 1 \oplus exp\ g\ 2 \oplus exp\ g\ 3 \oplus \dots$$

Using the semiring laws we show the definition above is equivalent to the other implementations of transitive closure. These three search algorithms, breadth-first, depth-first, and iterative-deepening, all compute the same graph transformation, but they differ in the order that they explore the graph.

However, there is a problem: the *Weighted* set computed by $graph^{*}\ a$ is infinite. The implementation of $*$ itself is recursive, and not well-founded. In fact, the definition in Eq. (6) does not pass Agda's termination checker. Even in a language like Haskell, where such a definition is acceptable, it is fraught with unclear semantics and possible errors. For example, the alternative function $*\prime$ (which, under the star semiring laws, should produce an equivalent definition) will produce a different weighted set, which will never return the actual shortest path to b. Some of this difficulty comes from mixing coinduction with quotients: since the weighted set is unordered, how can we productively return *any* element before another without imposing some kind of observable order?

The only way to deal with these problems is to address the coinduction inherent in the algorithm. We do exactly this in Section 4, where we will introduce a coinductive version of the weighted set.

3.3 Algorithmic Combinators

In the example implementation of Hamiltonian (Section 1), we used some useful combinators, like *pathed* and *filtering*. We will explain them in more detail here.

First, the *pathed* function:

$$\begin{aligned} &map_n : (A \to B) \to \\ &\qquad (Neighbours\ A \to Neighbours\ B) \end{aligned} \qquad \begin{aligned} &pathed : GraphOf\ \ V \to GraphOf\ (List^{+}\ V) \\ &pathed\ g\ (vs \mathbin{:\cdot} v) = \\ &\quad map_n\ (\lambda\ t \to vs \mathbin{:\cdot} v \mathbin{:\cdot} t)\ (g\ v) \end{aligned} \tag{8}$$

When used in combination with $*$ (or similar search functions), *pathed* tags each vertex with the path taken to reach that vertex. The tag is a non-empty snoc list, which is constructed by appending elements to the end with the constructor $:\cdot$. For instance, the edge $(a \mapsto d) \in graph^{*}$ becomes an edge $([a] \mapsto [a, c, d]) \in (pathed\ graph)^{*}$ (for the graph in Fig. 1). This lets our path finding algorithms actually produce observable paths.

The function works by converting each edge $s \mapsto t$ to a collection of edges: for each non-empty list p, an edge $(p \mathbin{:\cdot} s) \mapsto (p \mathbin{:\cdot} s \mathbin{:\cdot} t)$. The graph produced by a function like $*$ consists of sums of chains of $\ggg$, so when $*$ is applied to a graph after *pathed*, the effect is that each vertex passed through in a particular chain is accumulated and stored as a path on the last vertex of the chain.

$$\begin{aligned} (pathed\ graph)^{*} &= \mathbb{1} \oplus (pathed\ graph \ggg (pathed\ graph)^{*}) \\ &= exp\ (pathed\ graph)\ 0 \oplus exp\ (pathed\ graph)\ 1 \oplus exp\ (pathed\ graph)\ 2 \oplus \cdots \\ &\supseteq exp\ (pathed\ graph)\ 2 \\ &= pathed\ graph \ggg pathed\ graph \\ &\supseteq pathed\ \{a \mapsto c\} \ggg pathed\ \{c \mapsto d\} \\ &= \{(p \mathbin{:\cdot} a) \mapsto (p \mathbin{:\cdot} a \mathbin{:\cdot} c) \mid p \in List^{+}\ Vert\} \ggg \\ &\qquad\qquad \{(p \mathbin{:\cdot} c) \mapsto (p \mathbin{:\cdot} c \mathbin{:\cdot} d) \mid p \in List^{+}\ Vert\} \\ &= \{(p \mathbin{:\cdot} a) \mapsto (p \mathbin{:\cdot} a \mathbin{:\cdot} c \mathbin{:\cdot} d) \mid p \in List^{+}\ Vert\} \\ &\supseteq \{[a] \mapsto [a, c, d]\} \end{aligned}$$

The second function, *filtering*, is useful for removing edges from the output of an algorithm.

$$\begin{aligned} &filtering : (V \to Bool) \to GraphOf\ \ V \\ &filtering\ p\ v = if\ \ p\ v\ then\ \wr\ 0 \triangleright v\ \int\ else\ \wr\int \end{aligned} \tag{9}$$

The graph *filtering p* contains only the edges $x \mapsto x$ where $p\ x \equiv true$. For instance, *filtering even* is a graph of the natural numbers, where all of the even numbers have edges to themselves.

Connecting a graph to *filtering* has the effect of filtering the edges. So $g \ggg filtering\ p$ produces a graph where only edges in g that point to vertices for which the predicate p holds are retained.

3.4 More Efficient Representations

One of the advantages of describing algorithms using algebraic combinators is that the algebraic laws give strong guarantees about semantics, and allow us to change concrete implementations through homomorphisms while preserving those semantics. For instance, given a homomorphism $h : \mathit{Weighted}\ A \to D\ A$ to D, which is a monoid and a monad, if computation is more efficient on D, we can commute this homomorphism with any of the algorithms implemented here.

$$h\ (g^*\ v) = (h \circ g)^*\ v$$

If D is some more efficient representation than *Weighted*, then the right-hand side of this equation represents an optimisation.

A particularly strong framework for optimisation comes to us via Hinze [2012]. In brief, for any algebra represented by a typeclass C, the *free* object for that algebra (over some variable type A) is isomorphic to the type $\forall\ X \to C\ X \to (A \to X) \to X$ (where X is some set). This type is the "final encoding" of the free C. This type is also a monad (via the continuation monad), and it implements some operations vert efficiently. In particular, $\gg\!=$ and algebraic operations from C are $O(1)$ (although performance analysis on this type can be subtle: we have to consider converting to and from the type as well, which is not a constant-time).

This optimisation is particularly relevant to us, since the *Weighted* type is actually the *initial* encoding of the free object for an algebra that we will call the *weight semimodule.*

Definition 3.2 (Weight Semimodule). A weight semimodule consists of a semi-semiring $(S, \oplus, \otimes, \mathbb{1})$ and a commutative monoid $(V, \cup, \emptyset)$, together with an operation $\rtimes : S \to V \to V$, such that the following properties hold:

$$(x \otimes y) \rtimes z = x \rtimes (y \rtimes z) \qquad (x \oplus y) \rtimes z = (x \rtimes z) \cup (y \rtimes z) \qquad x \rtimes (y \cup z) = (x \rtimes y) \cup (x \rtimes z)$$

$$\mathbb{1} \rtimes x = x \qquad\qquad x \rtimes \emptyset = \emptyset$$

A semi-semiring is a semiring without a $\mathbb{0}$, and has all of the same laws as a semiring, except for those that involve $\mathbb{0}$.

The semi-semiring of relevance to us is fixed to be $(S, \sqcap, \bullet, \epsilon)$, for some monus S. Since the *Weighted* type is the free such semimodule, it is isomorphic to the final encoding:

$$\mathscr{W}\ A = \forall\ (V : \mathit{Type}) \to \mathit{isSet}\ V \to (\mathit{mod} : \mathit{WeightSemimodule}\ S\text{-}\mathit{weight}\ V) \to (A \to V) \to V$$

On this type, the operations $\cup$ and $\gg\!=$ are $O(1)$, where they are $O(n)$ on *Weighted*. Since the two encodings are isomorphic, $\mathscr{W}$ can be dropped-in as-is without fear of changing semantics.

In our formalisation we have proven that *Weighted* is the free weight semimodule, and that there is a split surjection from $\mathscr{W}\ A$ to *Weighted* A. Unfortunately, proving that this is an isomorphism requires parametricity, which is not available in Agda.

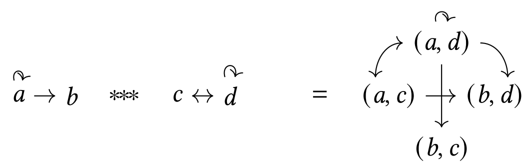
3.5 The Vertex Semiring

The edge semiring structured the combination of the edges of graphs; the vertex semiring organises the combination of vertices. Where the edge semiring was useful for searching and transforming graphs, the vertex semiring is useful for building them. The operations are diagrammed in Table 3.

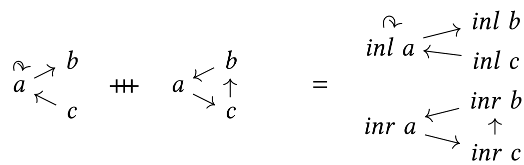
The first operator here, $*\!*\!*$, takes a kind of *product* of two graphs with different vertex types. The graph $f *\!*\!* g$ has vertices as the product of the vertices of f and g, and an edge $(x_f, x_g) \mapsto (y_f, y_g)$ has weight equal to the sum of the weights of the edges $x_f \mapsto y_f$ and $x_g \mapsto y_g$. The second operator, $+\!\!+\!\!+$, is a kind of *disjoint union*: $f +\!\!+\!\!+ g$ constructs a graph with vertices as the disjoint union of the vertices of f and g, and edges given by the union of their respective edges (the *wmap* function has type $(A \to B) \to \mathit{Weighted}\ A \to \mathit{Weighted}\ B$).

Table 3. Graph Combinators that Change the Vertices

$$
\begin{aligned}
&\_{*\!*\!*}\_ : \mathit{GraphOf}\ A \longrightarrow \\
&\qquad \mathit{GraphOf}\ B \longrightarrow \\
&\qquad \mathit{GraphOf}\ (A \times B) \\
&(g \mathbin{*\!*\!*} h)\ (v_l\ ,\ v_r) = \mathbf{do} \\
&\quad x \leftarrow g\ v_l \\
&\quad y \leftarrow h\ v_r \\
&\quad \mathit{return}\ (x\ ,\ y)
\end{aligned}
$$



$$
\begin{aligned}
&\_{+\!\!\!+\!\!\!+}\_ : \mathit{GraphOf}\ A \longrightarrow \\
&\qquad \mathit{GraphOf}\ B \longrightarrow \\
&\qquad \mathit{GraphOf}\ (A \uplus B) \\
&(g \mathbin{+\!\!\!+\!\!\!+} h) = \\
&\qquad \mathit{either}\ (\mathit{wmap}\ \mathit{inl} \circ g) \\
&\qquad\qquad (\mathit{wmap}\ \mathit{inr} \circ h)
\end{aligned}
$$



While these operators behave a little like the operators defined in the previous subsections, they have the crucial difference of *changing the type* of the underlying graphs. As such, these operations aren't monoid or semiring operators on the *GraphOf* type, instead, they form a semiring on a graph whose type depends on the values of its vertices.

$$
\begin{array}{lll}
\mathit{Graph} : \mathit{Type}_1 & \mathit{unit} : \mathit{Graph} & \mathit{void} : \mathit{Graph} \\
\mathit{Graph} = \Sigma[\ V : \mathit{Type}\ ] \times \mathit{GraphOf}\ V & \mathit{unit} = \top\ ,\ \mathit{return} & \mathit{void} = \bot\ ,\ \mathit{absurd}
\end{array}
$$

The identity for $*\!*\!*$ is *unit*. The first component of the pair is the type of vertices, $\top$ in this case, the type with one inhabitant; the second component is the neighbours function, which in this case is *return* and indicates a graph with one path, the identity path. The graph with no vertices and no paths is *void* (where $\mathit{absurd} : \forall(A : \mathit{Type}).\bot \longrightarrow A$), and it is the identity for $+\!\!\!+\!\!\!+$.

However, for these values to be the unit for their respective operators, equalities like $\mathit{unit} \mathbin{*\!*\!*} g \equiv g$ have to hold. If we unpack the pairs here we can see that we need to prove the equality $(\top \times V) \equiv V$, where V is the *type* of vertices of the graph g. We can use Cubical Agda [Vezzosi et al. 2021] to prove precisely this, via univalence (which states that any isomorphism $A \Leftrightarrow B$ implies an equality $A \equiv B$). We first prove that $(\top \times V)$ is *isomorphic* to V, and then we prove that that isomorphism is congruent through the equality on the right-hand-side of the pair. The proof of this is in our formalisation. This is the only use of univalence in this paper: all other results don't use the full power of CuTT, rather they use just simple quotient types.

The Plan. We are roughly halfway through the paper, and at this point the focus is going to change from *describing* graph algorithms to *verifying* them. Before moving on we will take a moment to recap what we have covered so far, and sketch what we will prove in the latter sections.

Section 2 gave our representation of graphs, and Section 3 presented an algebraic approach to search algorithms. The key development to bear in mind going forward is the $*$ function (Section 3.2), which defines transitive closure. In our framework, $*$ is a *specification* of search: it encompasses depth-first search, breadth-first search, and others, and the rest of the paper is devoted to implementing that specification, and verifying the implementation.

As discussed already, while it is tempting to use the specification as an implementation (as we did with Eqs. (6) and (7)), this ignores issues of well-foundedness. In Section 4, we will address well-foundedness, and in Section 4.3 we will use the theory of completely iterative monads (CIMs) to describe a template (Lemma 4.1) for a particular class of functions that can be said to implement recursive equations like Eq.(6) (and therefore $*$) in a well-founded way.

Finally, in Section 5, we will show how to instantiate this template while preserving the quotients on graphs that we established in Section 2. This will culminate in the *Neighbours* type, which can faithfully represent graphs as described in Section 2, can implement the graph algebras as described in Section 3, and can implement coinductive search as specified in Section 4.3.

4 Coinduction on Graphs

We have tiptoed around the issue of coinduction until now. In reality, coinduction is central to our graph representation and framework: the *GraphOf* type itself is in fact coinductive (even with an inductive *Weighted* type), and the transitive closure algorithms all can produce infinite results. This section will finally deal with coinduction *formally*, and provide a new framework for graphs and graph algorithms that can deal with infinite values in a principled way.

First, we will use the cofree comonad to construct a well-founded and efficient search algorithm on possibly-infinite graphs (Section 4.1). Then we will make progress towards redefining that algorithm as a graph transformation, using the coinductive resumption monad, from which we will derive the *Forest* type, a drop-in coinductive replacement for *Weighted* (Section 4.2). Finally we will use the theory of completely iterative monads (CIMs) to give a well-foundedness condition for coinduction on graphs (Section 4.3), and in particular for $*$ (Section 3.2).

4.1 Searching Infinite Graphs

To implement our coinductive search algorithm we will use a fundamental coinductive type: the cofree comonad [Ghani et al. 2003].

$$\mathit{Cofree}\ F\ A = \nu\ X\ .\ A \times F\ X \qquad (10) \qquad \_\blacktriangleleft\_ : A \to F\ (\mathit{Cofree}\ F\ A) \to \mathit{Cofree}\ F\ A$$

A value of type *Cofree F A* is a coinductive tree with internally-labelled nodes of type A, and a branching structure given by F. For instance, *Cofree List* is a rose tree [Ghani and Kurz 2007].

Unfortunately, the type definition of the Cofree comonad (Eq.(10)) is not strictly positive, and as such is rejected by Agda. Though we know that F in the definition must be positive, since it is a functor, we can't convince Agda of that fact on a meta-level. Furthermore, many higher-order functions which use *map* on the F will fail to pass the termination checker, even if they are truly structurally recursive. In our formalisation, we specialise the definition of *Cofree* (and ν, etc.) for the constructions that we formalise: *Heap* (Eq.(13)) and *Bush* (Eq.(21)).

The cofree comonad is defined using the greatest (i.e. coinductive) fixpoint ν.

$$\nu : (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \qquad \langle\!\langle\_\rangle\!\rangle : F\ (\nu\ F) \to \nu\ F \qquad \mathit{out} : \nu\ F \to F\ (\nu\ F)$$

An infinite nesting of Fs is given by $\nu\ F = F\ (\nu\ F) = F\ (F\ (F\ (F\ (\ldots))))$. Its interface consists of a constructor $\langle\!\langle\_\rangle\!\rangle$ and field *out*. We can construct elements of ν with the anamorphism, or *trace*:

$$\begin{aligned} &\mathit{ana} : (A \to F\ A) \to A \to \nu\ F \\ &\mathit{out}\ (\mathit{ana}\ \phi\ r) = F\ [\ \mathit{ana}\ \phi\ ]\ (\phi\ r) \end{aligned} \qquad (11) \qquad \begin{aligned} &\mathit{trace} : (A \to F\ A) \to A \to \mathit{Cofree}\ F\ A \\ &\mathit{trace}\ \psi = \mathit{ana}\ (\lambda\ x \to x\ ,\ \psi\ x) \end{aligned}$$

A *copattern* [Abel et al. 2013] is used to define *ana*, where $\mathit{out}\ (\mathit{ana}\ \phi\ r) = \ldots$ is equivalent to writing $\mathit{ana}\ \phi\ r = \langle\!\langle \ldots \rangle\!\rangle$. The expression $F[f] : F\ A \to F\ B$ maps $f : A \to B$ over the functor F.

The *trace* function can be used immediately as a graph algorithm:

$$\mathit{trace} : \mathit{GraphOf}\ A \to A \to \mathit{Tree}\ A \qquad \mathit{Tree} = \mathit{Cofree}\ \mathit{Weighted}$$

This takes a graph and produces a trace of searching through the graph from some starting node.

$$
\begin{aligned}
&\mathit{trace}\ \mathit{graph}\ a \equiv \\
&\quad a \blacktriangleleft \lbag\, 7 \triangleright b \blacktriangleleft \lbag\, 1 \triangleright c \blacktriangleleft \lbag\, 3 \triangleright d \blacktriangleleft \lbag\, 5 \triangleright b \blacktriangleleft \cdots \rbag , 1 \triangleright b \blacktriangleleft \cdots \rbag \quad \rbag \\
&\qquad , \ 2 \triangleright c \blacktriangleleft \lbag\, 3 \triangleright d \blacktriangleleft \lbag\, 5 \triangleright b \blacktriangleleft \cdots \quad \rbag , 1 \triangleright b \blacktriangleleft \cdots \rbag \rbag
\end{aligned} \tag{12}
$$

We can now transform a graph into a concrete tree representing a trace through the graph. Using the *pathed* function (Eq. (8)), this transformation can produce a tree of *paths* through the graph. The next step of this algorithm is to sort this tree, linearising it into a list of paths, ordered from least to greatest weight (and thereby allowing us to extract the shortest path). This linearisation process can be defined as a transformation between different instances of *Cofree*. The result of this algorithm, called on the graph in Fig. 1, is a *Chain* (defined below), that looks like the following:

$$[\ a\ ] \blacktriangleleft 2 \propto [\ c , a\ ] \blacktriangleleft 1 \propto [\ b , c , a\ ] \blacktriangleleft 1 \propto [\ c , b , c , a\ ] \blacktriangleleft 1 \propto [\ d , c , a\ ] \blacktriangleleft \cdots$$

This is a *Chain* of all of the shortest paths from a to every (reachable) vertex in Fig. 1, ordered by weight, where each *Link* in the *Chain* contains the difference in weight between adjacent paths. For instance, the shortest path from a is to itself, so it is at the head of the chain. Then, the distance to the next-longest path (to c) is 2. The path to b is the third-shortest, and it passes from a to c and then to b. This path has a total weight of 3, which can be calculated from the *Chain* by adding up all of the *preceding* weights in the chain. The data types involved here are the following:

$$
\begin{aligned}
&\mathit{Chain} = \mathit{Cofree}\ \mathit{Link} &\qquad& \textbf{pattern}\ \_\!\propto\!\_\ p\ x = \mathit{just}\ (p , x) \\
&\mathit{Link}\ A = \mathit{Maybe}\ (S \times A) && \textbf{pattern}\ \langle\rangle = \mathit{nothing}
\end{aligned}
$$

To implement this transformation we need to flatten this tree structure while ordering the paths according to weight. We need a function with the following type:

$$\mathit{search} : \mathit{Tree}\ A \longrightarrow \mathit{Chain}\ A$$

To implement this function efficiently, we can treat the *Tree* as a *heap*: notice that, when measured cumulatively, the weights in a *Tree* respect the heap ordering property (the weight of each node is less than or equal to the weight of its parent). An example is helpful here for explanation: take the traced tree in Eq. (12). While it certainly doesn't obey the heap ordering property as-is (because, for instance, b (with weight 7) is above c (with weight 1)), if we instead treat weights as cumulative (i.e. c is semantically tagged with the weight it takes to reach it from the root), we get the following tree, which does indeed respect the heap ordering property.

$$
\begin{aligned}
&a \blacktriangleleft \lbag\, 7 \triangleright b \blacktriangleleft \lbag\, 8 \triangleright c \blacktriangleleft \lbag\, 11 \triangleright d \blacktriangleleft \cdots , 9 \triangleright b \blacktriangleleft \cdots \rbag \quad \rbag \\
&\quad , \ 2 \triangleright c \blacktriangleleft \lbag\, 5 \triangleright d \blacktriangleleft \lbag\, 10 \triangleright b \blacktriangleleft \cdots \quad \rbag , 3 \triangleright b \blacktriangleleft \cdots \rbag \rbag
\end{aligned}
$$

The cumulative weight here is implicit: we won't actually transform the tree, rather we will semantically treat each weight as if it is equal to the sum of all the weights of its ancestors (plus itself). By this scheme, the root node has weight 0. For the heap property to hold on this cumulative view of *Tree*, we need precisely the property that $\forall x, y.\ x \leqslant x \bullet y$: this holds on all monuses.

What remains is to implement the necessary heap operations that would allow the transformation of a *Tree* into a list. As it happens, the *Tree* type bears a striking structural resemblance to a particularly efficient heap implementation: the *pairing heap* [Fredman et al. 1986]. We can adapt the pairing heap functions to work on our *Tree* type, preserving this efficiency.

One important thing to note is that we will discard, for now, the quotients on the *Tree* type, leaving us with the following type for comonadic, monus-based heaps:

$$\mathit{Heap} = \mathit{Cofree}\ (\mathit{List} \circ (S \times \_)) \tag{13}$$

Later we will see how to recover these quotients, but for now their inclusion would overcomplicate the heap implementation unnecessarily.

The overall goal here is an implementation of *search*. We can make some progress towards that implementation based only on the information we have so far. For instance, we know the type this function must have, and we know that—as a coinductive function—it must be implemented using *ana* Eq. (11). This leaves one missing piece: a function of type $\mathit{Heap}\ A \to A \times \mathit{Link}\ (\mathit{Heap}\ A)$.

$$\begin{aligned}&\mathit{search} : \mathit{Heap}\ A \to \mathit{Chain}\ A\\&\mathit{search} = \mathit{ana}\ (\boxed{\{!!\}} : \mathit{Heap}\ A \to A \times \mathit{Link}\ (\mathit{Heap}\ A))\end{aligned}$$

The term $\boxed{\{!!\}}$ is a *hole*: it denotes a missing piece of code that we have yet to write. This missing function is a kind of *popMin*: it should return the lowest-weight value in the heap, paired with the rest of the heap (if non empty). Since we know that the least-weight item in the heap is always at its root, we can further refine this hole, where $\mathit{map}_2 : (A \to B) \to C \times A \to C \times B$:

$$\begin{aligned}&\mathit{search} : \mathit{Heap}\ A \to \mathit{Chain}\ A\\&\mathit{search} = \mathit{ana}\ (\mathit{map}_2\ (\boxed{\{!!\}} : \mathit{List}\ (S \times \mathit{Heap}\ A) \to \mathit{Link}\ (\mathit{Heap}\ A)) \circ \mathit{out})\end{aligned}$$

This new hole is filled with the *merges* function.

$$\begin{aligned}&\mathit{merges} : \mathit{List}\ (S \times \mathit{Heap}\ A)\\&\quad \to \quad \mathit{Link}\ (\mathit{Heap}\ A)\\&\mathit{merges}\ [\,] = \langle\rangle\\&\mathit{merges}\ (x :: xs) = \mathit{just}\ (\mathit{merges}^{+}\ x\ xs)\end{aligned}$$

$$\begin{aligned}&\mathit{merges}^{+} : S \times \mathit{Heap}\ A \to \mathit{List}\ (S \times \mathit{Heap}\ A)\\&\quad \to \quad S \times \mathit{Heap}\ A\\&\mathit{merges}^{+}\ x_1\ [\,] = x_1\\&\mathit{merges}^{+}\ x_1\ (x_2 :: [\,]) = x_1 \bowtie x_2\\&\mathit{merges}^{+}\ x_1\ (x_2 :: x_3 :: xs) =\\&\quad (x_1 \bowtie x_2) \bowtie \mathit{merges}^{+}\ x_3\ xs\end{aligned}$$

This function collapses a list of heaps to a single heap: it doesn't follow a normal foldr-like pattern, instead performing a two-level merge which is vital to the performance of the heap as a whole.

It uses the $\bowtie$ function to combine two weighted heaps.

$$\begin{aligned}&\_\bowtie\_ : S \times \mathit{Heap}\ A \to S \times \mathit{Heap}\ A \to S \times \mathit{Heap}\ A\\&(w_l\ ,\ l \blacktriangleleft ls) \bowtie (w_r\ ,\ r \blacktriangleleft rs) = \mathit{if}\ \ \mathit{does}\ (w_l \leqslant?\ w_r)\ \mathit{then}\ w_l\ ,\ l \blacktriangleleft (w_r \mathbin{\dot{-}} w_l\ ,\ r \blacktriangleleft rs) :: ls\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ w_r\ ,\ r \blacktriangleleft (w_l \mathbin{\dot{-}} w_r\ ,\ l \blacktriangleleft ls) :: rs\end{aligned}$$

The merge of two weighted heaps $x \bowtie y$ produces a new heap with the lowest-weight node of x and y at the root, and the higher-weighted node as a subtree of that new heap. Note that the heap inserted as a subtree has its weight adjusted (in the first branch of the if-expression, we have $w_l, l \blacktriangleleft (w_r \mathbin{\dot{-}} w_l, r \blacktriangleleft rs) :: ls$, instead of $w_l, l \blacktriangleleft (w_r, r \blacktriangleleft rs) :: ls$). This is because the weights are semantically cumulative: the weight attached to r should be equal to w_r; if it was placed below w_l in the new tree it would be semantically equal to $w_l \bullet w_r$, so we have to correct for this by subtracting the parent weight, yielding $w_l \bullet w_r \mathbin{\dot{-}} w_l = w_r$.

Finally, the *search* function is as follows:

$$\begin{aligned}&\mathit{search} : \mathit{Heap}\ A \to \mathit{Chain}\ A\\&\mathit{search} = \mathit{ana}\ (\mathit{map}_2\ \mathit{merges} \circ \mathit{out})\end{aligned}$$

This function has complexity $O(n \log n)$, where n is the number of elements explored in the chain.

There is a similar algorithm presented by Kidney and Wu [2021]: that adaptation of the pairing heap, however, is based on the free monad, changing the mechanics of the algorithm. Furthermore, that version is implemented only in Haskell, and does not deal with coinduction.

4.2 The Coinductive Resumption Monad

To express coinductive algorithms as graph transformations, we need a coinductive variant of the *Weighted* type. What we need is a data structure with similar well-foundedness and coinductive properties to the cofree comonad, but has a monad instance similar to *Weighted*, so that the edge semiring is preserved. This structure is the *coinductive resumption monad*, *Res* [Piróg and Gibbons 2014]. To define *Res* we will first need the free "completely iterative monad" (cim):

$$F^{\infty}\ A = \nu\, X\ .\ (F\ X \uplus A)$$

We will return to cims in more detail in a moment, for now note that this type is identical to the cofree comonad Eq.(10) except that the binary product ($\times$) has been replaced with disjoint union ($\uplus$). It also bears a resemblance to the free monad ($\mathit{Free}\ F\ A = \mu X.\ F\ X \uplus A$), but it uses the coinductive fixpoint ν instead of the inductive fixpoint μ. This type is a possibly-infinite tree with leaves labelled with a value of type A, and branching structure given by F. It is a monad for any functor F.

The coinductive resumption monad, built on ∞, is defined as follows:

$$\mathit{Res}\ \Sigma\ M = M \circ (\Sigma \circ M)^{\infty}$$

$\mathit{Res}\ \Sigma\ M$ is a monad for any functor Σ and monad M. It is a possibly-infinite tree, with multiple layers of effects from M being interspersed by Σ.

The type F^{∞} is a monad with discrete layers of effects given by F: the monadic bind on F^{∞} preserves this distinction, maintaining the separation between layers. In contrast, the type $\mathit{Res}\ \Sigma\ M$ has discrete layers of Σ, but the effects given by M can interact: the monadic bind $x \ggg\!= k$ on $\mathit{Res}\ \Sigma\ M$ combines the effects at the leaves of x with the top-level effects in k.

For this paper, the relevant instantiation of *Res* sets $\Sigma := \mathit{id}$, and $M := \mathit{Weighted}$.

$$\mathit{Forest} = \mathit{Weighted} \circ \mathit{Weighted}^{\infty} \tag{14}$$

This type inherits the semimodule structure from *Weighted*, however this type is tree-shaped, where *Weighted* was a *flat* collection.

The monadic bind on *Forest* grafts sub-trees into leaves.

$$\begin{aligned} xs = \wr\ & 7 \triangleright \langle\!\langle\ \mathit{inl}\ \wr\ 1 \triangleright \langle\!\langle\ \mathit{inr}\ a\ \rangle\!\rangle \\ &\qquad\qquad\quad ,\ 2 \triangleright \langle\!\langle\ \mathit{inr}\ b\ \rangle\!\rangle\ \int\ \rangle\!\rangle \\ ,\ & 3\ \triangleright \langle\!\langle\ \mathit{inr}\ c\ \rangle\!\rangle\ \int \end{aligned}$$

$$\begin{aligned} k = \lambda\ \{\ & b \longrightarrow \wr\ 3 \triangleright \langle\!\langle\ \mathit{inr}\ 0\ \rangle\!\rangle\ \int \\ ;\ & c \longrightarrow \wr\ 5 \triangleright \langle\!\langle\ \mathit{inr}\ 1\ \rangle\!\rangle \\ &\qquad\quad ,\ 6 \triangleright \langle\!\langle\ \mathit{inr}\ 2\ \rangle\!\rangle\ \int \\ ;\ & \_ \longrightarrow \wr \int\ \} \end{aligned}$$

$$xs \ggg\!=\ k \equiv \wr\ 7 \triangleright \langle\!\langle\ \mathit{inl}\ \wr\ 5 \triangleright \langle\!\langle\ \mathit{inr}\ 0\ \rangle\!\rangle\ \int\ \rangle\!\rangle\ ,\ 8 \triangleright \langle\!\langle\ \mathit{inr}\ 1\ \rangle\!\rangle\ ,\ 9 \triangleright \langle\!\langle\ \mathit{inr}\ 2\ \rangle\!\rangle\ \int$$

Notice that two layers of monadic effects are merged, but the rest are kept separate: the 3 weight in *xs* is added to 5 and 6 in the output, but the 7 weight, being insulated one level above any leaves, is preserved. This allows the *Forest* type to represent coinductive algorithms: infinite structure can be guarded under nested sub-trees; in the *Weighted* type, such nesting would have to be flattened, making the structure too eager and making it impossible to represent infinite computations.

$$\begin{tikzcd} X \arrow[r, "e^\dagger"] \arrow[d, "e"] & MA \\ M(X+A) \arrow[r, "M\left[e^\dagger \triangledown \eta\right]"] & M^2A \arrow[u, "\mu"] \end{tikzcd}$$

(a) The solution to a recursive equation

$$\begin{tikzcd} \overline{M}(X+A)+A \arrow[r, "\sigma \triangledown \eta \circ inr"] & M(X+A) \\ X \arrow[u, dashed, "e_i"] \arrow[ur, "e"] & \end{tikzcd}$$

(b) Factoring a guarded equation

Fig. 3. Equations on CIMS

4.3 Completely Iterative Monads

The *Res* type is an example of a *completely iterative monad* [Aczel et al. 2003; Elgot et al. 1978; Milius 2005]: informally, this is a class of monads that support a certain kind of coinduction. For our purposes, CIMs will give us a formal theory that lets us describe what it means for an algorithm to be a well-founded implementation of recursive definitions like the Kleene star Eq. (6).

The machinery for coinduction with CIMs centres around recursive, guarded equations. A recursive equation in this context is a morphism $X \longrightarrow M\ (X + A)$ that represents a recursive function. In the equation, A is the final result, and X is the type of variables being recursed over. Let's use $*\prime$ Eq. (7) as an example: in this recursive function, both X and A are vertices of the graph. Below, we have rewritten $*\prime$ to *dfs* by inlining the definition of the edge semiring operators; the equation morphism corresponding to *dfs* is dfs_e.

$$\begin{aligned} &dfs_e : GraphOf\ A \longrightarrow A \longrightarrow Forest\ (A \uplus A) \\ &dfs_e\ g\ x = \\ &\quad return\ (inr\ x) \cup (g\ x \ggg \lambda\ y \longrightarrow return\ (inl\ y)) \end{aligned}$$

$$\begin{aligned} &dfs : GraphOf\ A \longrightarrow A \longrightarrow Forest\ A \\ &dfs\ g\ x = \\ &\quad return\ x \cup (g\ x \ggg \lambda\ y \longrightarrow dfs\ g\ y) \end{aligned}$$

In *dfs* there is a recursive call ($dfs\ g\ y$), in the corresponding equation dfs_e notice that this recursive call is replaced with a left injection into the sum ($inl\ y$).

The *solution* to the equation—the mechanism for turning dfs_e into *dfs*—is defined as the morphism $e^\dagger$ such that the diagram in Fig. 3a commutes. Of course, not every equation has a solution: for CIMs, all *guarded* equations have a solution.

Guardedness for CIMs is defined using ideals. Every CIM M has a related functor $\overline{M}$, called its ideal, with a natural transformation $\sigma : \overline{M} \longrightarrow M$. Informally, $\overline{M}\ A$ is the "guarded" subset of $M\ A$; it contains all of the computations which "make progress", in a coinductive sense. It does not include things like *return x*. A guarded equation, then, is one that factors through this ideal, as in Fig. 3b.

We can formalise this notion of a CIM as follows. First, given a monad M and its ideal $\overline{M}$ we can define the equation and flat equation morphisms:

$$Equation\ X\ A = X \longrightarrow M\ (X \uplus A) \qquad\qquad Flat\ X\ A = X \longrightarrow \overline{M}\ (X \uplus A) \uplus A$$

A well-founded equation is some $e : Equation\ X\ A$ such that $\exists (e_i : Flat\ X\ A).\ e \equiv (\sigma \triangledown \eta \circ inr) \circ e_i$. It can be cumbersome to work with "equations e such that they factor into $i : \ldots$", so instead we will work with the *Flat* type directly. A CIM, then, is a monad M with an ideal $\overline{M}$ where there is a function $\ddagger$ from a *Flat* morphism to a *Solution* morphism $X \longrightarrow M\ A$:

$$\_^\ddagger : Flat\ X\ A \longrightarrow Solution\ X\ A \qquad\qquad Solution\ X\ A = X \longrightarrow M\ A$$

That makes the diagram in Fig. 3a commute:

$$
\begin{aligned}
&\_\mathsf{Solves}\_ : \quad \mathit{Solution}\ X\ A \longrightarrow \mathit{Equation}\ X\ A \\
&\qquad\qquad\quad \longrightarrow \mathit{Type} \\
&e^{\dagger}\ \mathsf{Solves}\ e = e^{\dagger} \equiv \mu \circ \mathsf{M}[\ e^{\dagger}\ \triangledown\ \eta\ ] \circ e
\end{aligned}
\qquad
\begin{aligned}
&\mathit{solution} : (e_i : \mathit{Flat}\ X\ A) \longrightarrow \\
&\qquad\qquad e_i{}^{\ddagger}\ \mathsf{Solves}\ (\sigma\ \triangledown\ \eta \circ \mathit{inr}) \circ e_i
\end{aligned}
$$

Notice that coinduction, or, indeed, recursion, isn't mentioned directly here: this is a formalisation of coinduction that does not perform coinduction itself. It is any *implementation* of this class that will have to perform the well-foundedness: the burden of proof is shifted to the implementer.

The ideal for *Forest* is the same as the ideal for the coinductive resumption monad. It is as follows:

$$
\overline{\mathit{Forest}} = \mathit{Weighted} \circ \mathit{Forest}
\qquad\qquad
\begin{aligned}
&\sigma : \overline{\mathit{Forest}}\ A \longrightarrow \mathit{Forest}\ A \\
&\sigma = \mathsf{Weighted}[\ \langle\!\langle \_ \rangle\!\rangle \circ \mathit{inl}\ ]
\end{aligned}
$$

In other words, the ideal of a *Forest* is a *Forest* with at least two layers of nesting. The following is a guarded equation, with its solution on the right:

$$
\begin{aligned}
&\mathit{verts} : X \longrightarrow \overline{\mathit{Forest}}\ (X \uplus \mathit{Vert}) \uplus \mathit{Vert} \\
&\mathit{verts}\ x = \mathit{inl}\ \lbag\ 1 \triangleright \mathit{return}\ (\mathit{inr}\ a) \\
&\qquad\qquad\quad ,\ 2 \triangleright \mathit{return}\ (\mathit{inl}\ \ x)\ \rbag
\end{aligned}
\qquad
\begin{aligned}
&\mathit{soln} = \lbag\ 1 \triangleright \langle\!\langle\ \mathit{inl}\ \lbag\ 0 \triangleright \langle\!\langle\ \mathit{inr}\ a\ \rangle\!\rangle\ \rbag\ \rangle\!\rangle \\
&\qquad\ ,\ 2 \triangleright \langle\!\langle\ \mathit{inl}\ \lbag\ 1 \triangleright \langle\!\langle\ \mathit{inl}\ \lbag\ 0 \triangleright \langle\!\langle\ \mathit{inr}\ a\ \rangle\!\rangle\ \rbag\ \rangle\!\rangle \\
&\qquad\qquad\qquad\quad ,\ 2 \triangleright \langle\!\langle\ \mathit{inl} \cdots \rangle\!\rangle\ \rbag\ \rangle\!\rangle\ \rbag
\end{aligned}
$$

The solution on the right is the infinitely-nested forest generated by layering *verts* on the left: $\mathit{soln} \approx \mathit{verts}\ (\mathit{verts}\ (\mathit{verts}\ (\ldots)))$.

The guardedness condition given above doesn't quite work for equations like dfs_e. The factorisation in Fig. 3b through $\overline{M}(X + A) + A$, allows *only* guarded effects to be present in the step function. However, for functions like *dfs*, there are both guarded and unguarded effects in the step function ($\mathit{return}\ (\mathit{inr}\ x)$ is unguarded), but the equation is still well-founded, since all *recursion* is guarded.

We define a new guardedness condition that suits dfs_e better: it is given in Lemma 4.1. This factorisation condition allows effects on the right-hand-side of the equation, but only the parameters (A) may be returned purely; all variables for recursion must be guarded by the ideal ($\overline{M}\ X$). Any equation that factors in this way also factors as Fig. 3b.

Lemma 4.1. *For a* cim M*, any equation which factors through* $M(\overline{M}\ X + A)$ *as follows has a solution:*

$$
\begin{tikzcd}
M(\overline{M}\ X + A) \arrow[rr, "\mu \circ M[\sigma \circ \overline{M}[\mathit{inl}]\ \triangledown\ \eta \circ \mathit{inr}]"] & & M(X + A) \\
X \arrow[u, dashed, "e_j"] \arrow[urr, "e"'] & &
\end{tikzcd}
\tag{15}
$$

Proof. Given an equation $e : X \longrightarrow M(X + A)$, which factors through e_j as in Eq. (15), we must show that it has a solution $e^{\dagger} : X \longrightarrow M\ A$.

We first construct an equation $\tilde{e} : \overline{M}\, X \longrightarrow M(\overline{M}\, X + A)$, which factors as follows:

$$\begin{tikzcd} \overline{M}(\overline{M}\, X + A) + A \arrow[rr, "\sigma \,\triangledown\, \eta \circ inr"] & & M(\overline{M}\, X + A) \\ \overline{M}\, X \arrow[u, dashed, "\tilde{e}_i = inl \circ \overline{\mu} \circ \overline{M}[e_j]"] \arrow[urr, "\tilde{e}"'] & & \end{tikzcd}$$

By guardedness, this equation has a solution $\tilde{e}^{\dagger} : \overline{M}\ X \longrightarrow M\ A$, shown below on the left, from which we can derive a solution $e^{\dagger} = \mu \circ M\left[\tilde{e}^{\dagger} \,\triangledown\, \eta\right] \circ e_j$, shown below on the right.

$$\begin{tikzcd} \overline{M}\, X \arrow[r, "\tilde{e}^{\dagger}"] \arrow[d, "\tilde{e}"] & M\, A \\ M(\overline{M}\, X + A) \arrow[r, "M\left[\tilde{e}^{\dagger} \,\triangledown\, \eta\right]"] & M^2 A \arrow[u, "\mu"] \end{tikzcd} \qquad \begin{tikzcd} X \arrow[r, "e^{\dagger} = \mu \circ M\left[\tilde{e}^{\dagger} \,\triangledown\, \eta\right] \circ e_j"] \arrow[d, "e"] & M\, A \\ M(X + A) \arrow[r, "M\left[e^{\dagger} \,\triangledown\, \eta\right]"] & M^2 A \arrow[u, "\mu"] \end{tikzcd}$$

What remains is to show that $e^{\dagger}$ is indeed a solution, i.e. that the diagram on the right commutes, $e^{\dagger} = \mu \circ M[e^{\dagger} \triangledown \eta] \circ e$. This is proven in our formalisation, in the module Codata.CIM, or linked here. □

Using this guardedness condition, we can factor *dfs* as follows:

$$\begin{aligned} &dfs_j : (A \longrightarrow \overline{Forest}\ A) \longrightarrow A \longrightarrow Forest\ (\overline{Forest}\ A \uplus A) \\ &dfs_j\ g\ x = return\ (inr\ x) \cup return\ (inl\ (g\ x)) \end{aligned}$$

Notice that the type of the input graph had to be changed as well: we can only search graphs where there is a guarded step between a vertex and its neighbours. This condition makes sense! The only way for search to be productive is if every step is itself productive. It is possible to artificially add this guardedness step (simply with *return*∘), so *dfs* can be used with *any* graph, but this technique will not work in the next section where our guardedness conditions become more sophisticated.

The † we define in this lemma is analogous to the ∗ function; an instantiation of † that satisfies the *Solves* predicate is an implementation of search.

Summary. This section has explored coinduction in the context of graph algorithms, first using the *Heap* type to implement search through a graph. Then, we explored the *Forest* type, a data type suitable for representing the neighbours of a vertex in a weighted graph, which can replace the *Weighted* type as-is. It is a monad and a monoid, so the graph construction operations defined previously still apply, and it is *coinductive*, meaning that it can be used to define corecursive algorithms, of which *search* is an example. Finally, using the theory of CIMs, we gave a concise *guardedness* condition, which gives a template for implementing ∗.

5 Quotienting Coinductive Structures

While the *Forest* type (Eq. (14)) does function as a data structure for representing the neighbours of a vertex in a graph, it isn't a perfect fit as the coinductive version of *Weighted*. In particular, the *Forest* type is missing some quotients: it distinguishes some graphs which should be semantically equal. Take, for example, a simple graph with one edge $a \mapsto b$ with weight 2. Both g_1 and g_2 below are valid representations of this graph, despite containing observably different *Forest*s.

$$g_1\ a = \wr\ 2 \triangleright \langle\!\langle\ inr\ b\ \rangle\!\rangle\ \int \qquad\qquad g_2\ a = \wr\ 1 \triangleright \langle\!\langle\ inl\ \wr\ 1 \triangleright \langle\!\langle\ inr\ b\ \rangle\!\rangle\ \int\ \rangle\!\rangle\ \int$$

However, it is difficult to quotient out this difference. The thing distinguishing g_1 and g_2 above is the level of nesting. But since we use this nesting to guard coinduction, we can't just "forget" it with a quotient without breaking well-foundedness.

The difficulty is similar to the one faced by the delay monad [Chapman et al. 2019]. In general, dependently-typed programming languages can handle simple inductive types well. Coinductive types are less well supported, but there are a number of techniques available [Abel and Pientka 16ed; Abel et al. 2013; Gibbons and Hutton 1999]. Quotient types are less supported still, although with the recent development of Cubical Type theory [Vezzosi et al. 2021] they are quickly catching up [see also: Hewer and Hutton 2024]. It is the combination of the two—coinductive, quotiented types—that causes problems for us. While there has been some work in this direction [Birkedal et al. 2016; Joram and Veltri 2023; Veltri and Vezzosi 2023], these types remain difficult to work with.

This section will construct a coinductive version of the *Weighted* type: a coinductive, quotiented weighted set, that can serve as a representation for the neighbours of a graph that allows for search algorithms to be expressed as graph transformations in a well-founded way. This representation is based on a simple "bounding" operator (Section 5.1). We will use this operator to implement a well-founded search algorithm. Then, in Section 5.2, we will use this operator to quotient the *Forest* type, yielding the *Bush* type. Unfortunately, this type doesn't have a monad instance without a certain choice principle. In Section 5.3 we will develop an alternative type (*Neighbours*) that *does* have a monad instance; finally, in Section 5.4 we will show that this type is a CIM, and use this to implement a search algorithm.

5.1 A Terminating Bounding Operator

The strategy for quotienting coinductive types in this section will be to find a representation of the coinductive structure that doesn't actually rely on coinduction internally. These representations will use a kind of step-indexing [Appel and McAllester 2001], where the indexing quantity is weight.

The crucial function is the following "bounding" operator:

$$\begin{aligned} &\succ\!\!\!\!+ && : \mathit{Weighted}\ A \to W \to \mathit{Weighted}\ A \\ &s \succ\!\!\!\!+ w && = \wr u \triangleright x \mid u \triangleright x \leftarrow s, u \leqslant w \int \end{aligned} \tag{16}$$

The expression $s \succ\!\!\!\!+ w$ returns a set containing all of the entries in s with weights smaller than w.

$$\wr 2 \triangleright w, 5 \triangleright x, 1 \triangleright y, 3 \triangleright z, \int \succ\!\!\!\!+ 2 = \wr 2 \triangleright w, 1 \triangleright y \int$$

We will explore the theory of this operator and the representations it gives rise to in the rest of this section. First, let's use it to properly terminate depth-first search.

A simple way to reimplement $*$ (Eq.(6)) to be well-founded is to add a $\mathbb{N}$ parameter.

$$\begin{aligned} &{*\succ\!\!\!\!+} && : \mathit{GraphOf}\ A \to \mathbb{N} \to \mathit{GraphOf}\ A \\ &g{*\succ\!\!\!\!+}\ \ 0 && = \mathbb{0} \\ &g{*\succ\!\!\!\!+}(n{+}1) && = \mathbb{1} \oplus ((g *\succ\!\!\!\!+ n) \otimes g) \end{aligned} \tag{17}$$

$g *\succ\!\!\!\!+ n$ returns a list of all vertices n or fewer steps away from r in the graph g. The notation gives a hint as to the semantics: it looks like the composition of two functions, $*$ and $\succ\!\!\!\!+$. Of course, if we had implemented the function that way it would no longer be well-founded, since one of the intermediate steps would have performed unbounded recursion.

Instead, we write a single recursive function that performs both $*$ and $\succ\!\!\!\!+$. Because we recurse on the natural-number argument, this algorithm is clearly well-founded.

We will next generalise this technique to use a certain class of *weights*, rather than just $\mathbb{N}$. For a weight of type S, the new version of $*\succ\!\!\!\!+$ will have the following type:

$$*\succ\!\!\!\!+ : \mathit{GraphOf}\ A \to S \to \mathit{GraphOf}\ A$$

And we could imagine implementing it something like the following:

$$(g *\succ\!\!\!\!+ w)\ v = \wr \epsilon \triangleright v \int \cup \wr\, p \bullet q \triangleright y \mid p \triangleright x \leftarrow g\ v,\ p \preceq w,\ q \triangleright y \leftarrow (g *\succ\!\!\!\!+ (w \dot{-} p))\ x \int \tag{18}$$

The function $(g \ast\!\!\rightarrowtail w)\ v$ returns a weighted set of all vertices reachable from v in paths with weights no greater than w. It returns a weighted set consisting of first v, with weight ϵ (since every vertex is reachable by itself), and then recursively searches from the neighbours of v, ignoring neighbours whose edges weigh more than w, and continuing the search with a new reduced weight bound of $w \dot{-} p$, where p is the weight of the edge from v to the vertex in question.

$$(\mathit{graph} \ast\!\!\rightarrowtail 5)\ a = \wr 0 \triangleright a, 2 \triangleright c, 3 \triangleright b, 5 \triangleright d, 4 \triangleright c \wr$$

Justifying termination on Eq. (18) is more difficult than on Eq. (17). In the case of Eq. (17), the recursive call $g \ast\!\!\rightarrowtail n$ is safe because its argument (n) is structurally smaller than the argument to the top-level function ($n + 1$ in $g \ast\!\!\rightarrowtail (n + 1)$). In the case of Eq. (18), however, the recursive call is $g \ast\!\!\rightarrowtail (w \dot{-} p)$, and the top-level call is $g \ast\!\!\rightarrowtail w$. For this call to be safe, $w \dot{-} p$ must be structurally smaller than w: it must be smaller than w according to some well-founded relation.

A relation $\prec$ is *well-founded* if every chain $x_1 \prec x_2 \prec \ldots \prec x_n$, is finite. This is a generalisation of the "structurally smaller" recursion condition that many total languages test for. To verify Eq. (18) we need to come up with a well-founded relation on monuses such that $w \dot{-} p \prec w$ holds.

The simple less-than relation on monuses won't work: while this has a lower bound ($\forall x.\epsilon \preceq x$), consider $(\mathbb{Q}^+, +, 0)$, the additive monoid on the positive rational numbers. While this forms a valid monus, the less-than relation can construct infinite chains ($x > 0 \implies x > \frac{x}{2} > \frac{x}{3} > \frac{x}{4} > \ldots$).

Instead, we'll introduce the following relation $\prec_s$, for some step size s:

$$x \prec_s y \iff x \bullet s \preceq y \iff \exists k.\ y = x \bullet s \bullet k \tag{19}$$

This relation states that x is no greater than y, and the difference between x and y is at least s. When $s = \epsilon$, the relation reduces to the normal algebraic relation (i.e. $x \prec_\epsilon y \iff x \leqslant y$), but when $s \neq \epsilon$ this defines a less-than relation that may be suitable for well-founded recursion.

Definition 5.1 (Well-Founded Monus). A well-founded monus is one where the relation $\prec_s$, for $s \neq \epsilon$, is well-founded. We further require the monus to be cancellative.

And indeed $\prec_s$ on $\mathbb{N}$ as well as $\mathbb{Q}^+$ is well-founded. We require the monus to be cancellative (i.e. $x\bullet$ is injective for all x) for proofs like the one below.

One caveat of this relation is that algorithms using it have a minimum "resolution". For a recursive call to be safe, it must have a step size of at least s. In practice, this can mean that, for instance, a graph being searched cannot have edges of weight smaller than s. This is called the *step condition*.

Let's use this relation to verify the implementation of $\ast\!\!\rightarrowtail$ above is well-founded. The recursive call that needs to be verified is $(g \ast\!\!\rightarrowtail (w \dot{-} p))x$. This call is guarded by a condition that $p \preceq w$, so we know this condition holds before the call is made. The proof is as follows:

$$\begin{array}{ll}
w \dot{-} p \prec_s w & \iff \{\text{Definition of } \prec_s\} \\
(w \dot{-} p) \bullet s \preceq w & \iff \{\text{Adding } p \text{ to both sides}\} \\
(w \dot{-} p) \bullet s \bullet p \preceq w \bullet p & \iff \{\text{Rearranging}\} \\
(w \dot{-} p) \bullet p \bullet s \preceq w \bullet p & \iff \{p \preceq w \implies (w \dot{-} p) \bullet p = w\} \\
w \bullet s \preceq w \bullet p & \iff \{\text{Cancel } w\} \\
s \preceq p \quad \square & \ \{\text{The step condition}\}
\end{array}$$

To actually practically implement an algorithm using this well-founded recursion principle we will use the following data type:

$$\textbf{data}\ \mathit{Acc}\ \_\!\prec\!\_\ x\ \textbf{where}\ \mathit{acc} : (\forall\ y \to y \prec x \to \mathit{Acc}\ \_\!\prec\!\_\ y) \to \mathit{Acc}\ \_\!\prec\!\_\ x$$

A well-founded relation $\prec$ is one where it is possible to construct a value $\mathit{Acc}\ \_ \prec \_\ x$, for any x. A well-founded monus is one whereby a function exists with the type $\forall s \to s \not\equiv \epsilon \to \forall x \to \mathit{Acc}\ \ \prec_s\ \ x$.

We won't include an example of using *Acc* here, as it introduces a lot of clutter and boilerplate that is not relevant to the theory, but we use it to prove the later well-foundedness lemmas in the paper.

5.2 Quotienting by Up-To Equivalence

Our first attempt at a quotiented version of *Forest* will rely on the following up-to operator:

$$\_|\_\succ\!\!\!+\_ : \mathit{Forest}\ A \to \mathit{Step} \to W \to \mathit{Weighted}\ A \qquad (20) \qquad \mathit{Step} = \exists\ s \times s \not\equiv \epsilon$$

The expression $xs \mid s \succ\!\!\!+ w$ returns the contents of the tree xs, inspected to the depth w. As a result, it needs to use the supplied weight to bound termination, as *Forest* is a coinductive structure. For that purpose, this function also takes a *Step*, which is used to facilitate stepwise well-founded recursion via the well-founded monus (Definition 5.1).

Crucially, the result of this function *collapses* all of the level structure in the source tree. As such, it is suitable as a function to the equivalence class of trees quotiented by ignoring the levels. Practically speaking, that means we will give our new tree type as the *Bush* type quotiented by the following equivalence relation:

$$\mathit{Bush}\ A = \mathit{Forest}\ A\ /\ \mathit{Equiv\text{-}UpTo} \quad (21) \qquad \begin{array}{l}\mathit{Equiv\text{-}UpTo}\ xs\ ys = \\ \quad \forall\ s\ w \to xs \mid s \succ\!\!\!+ w \equiv ys \mid s \succ\!\!\!+ w\end{array}$$

There are a number of positive aspects to this type that might not be immediately apparent. Firstly, although the function Eq. (20) is "lossy", in that it ignores edges smaller than the step condition, the corresponding quotient is not, because of the universal quantification. Informally, if two structures are indistinguishable at *any* resolution, then they must be truly equal.

Secondly, since the type is quotiented by a function into an equivalence class, the original quotients on the *Forest* type are now superfluous, as the Eq. (20) function will find them for us anyway. As a result, we can define a new *Forest* type that is a little easier to work with:

$$\mathit{Forest} = \mathit{List} \circ ((W \times \_) \circ \mathit{List})^{\infty}$$

This is the same type as in Kidney and Wu [2021]; it allows zero-weight vertices to be placed in the lowest level forest without being tagged with a weight. This makes especially the *heap* operations (Section 4.1) simpler. Those heap operations are largely unchanged; here are the few differences:

$$\begin{array}{l}
\bowtie : W \times \mathit{Forest}\ A \to W \times \mathit{Forest}\ A \to \\
\qquad W \times \mathit{Forest}\ A \\
(w_l\ ,\ ls) \bowtie (w_r\ ,\ rs)\ \mathbf{with}\ w_l \leqslant\!|\!\geqslant w_r \\
...\ |\ \mathit{inl}\ (w_r \dot{-} w_l\ ,\ _) = w_l\ ,\ \langle\!\langle\ \mathit{inl}\ (w_r \dot{-} w_l\ ,\ rs)\ \rangle\!\rangle :: ls \\
...\ |\ \mathit{inr}\ (w_l \dot{-} w_r\ ,\ _) = w_r\ ,\ \langle\!\langle\ \mathit{inl}\ (w_l \dot{-} w_r\ ,\ ls)\ \rangle\!\rangle :: rs
\end{array}
\qquad
\begin{array}{l}
\mathit{partition} : (A \to B \uplus C) \to \mathit{List}\ A \to \\
\qquad \mathit{List}\ B \times \mathit{List}\ C \\
\mathit{swap} : A \times B \to B \times A \\
\mathit{merges} : \mathit{List}\ (W \times \mathit{Forest}\ A) \to \\
\qquad \mathit{Link}\ (\mathit{Forest}\ A)
\end{array}$$

$$\begin{array}{l}
\mathit{popMin} : \mathit{Forest}\ A \to \mathit{List}\ A \times \mathit{Link}\ (\mathit{Forest}\ A) \\
\mathit{popMin} = \mathit{map}_2\ \mathit{merges} \circ \mathit{swap} \circ \mathit{partition}\ \mathit{out}
\end{array}
\qquad
\begin{array}{l}
\mathit{search} : \mathit{Forest}\ A \to \mathit{Chain}\ (\mathit{List}\ A) \\
\mathit{search} = \mathit{ana}\ \mathit{popMin}
\end{array}$$

This *search* algorithm creates a *Chain* of *List*s of vertices of the same weight; to have this algorithm obey the quotient, we have to swap out those *List*s for sets, and accumulate along the returned *Chain*. We have not formalised this quotient-respecting version of *search*; however, *Bush* has a more serious problem: the monad instance.

Unfortunately, the *Bush* type runs into trouble when it comes to implementing the monad operations (*join*, in particular). This is actually a well-known difficulty: in order to implement *join* on a set quotient, a kind of *choice principle* is required on certain types. Here, the partially-implemented *join* illustrates the problem:

$$\begin{aligned} &\mathit{join} : \mathit{Bush}\ (\mathit{Bush}\ A) \longrightarrow \mathit{Bush}\ A \\ &\mathit{join} = \mathit{rec/}\ \mathit{squash/}\ \mathit{join\text{-}alg}\ \mathit{join\text{-}coh} \end{aligned}$$

$$\begin{aligned} &\mathit{join\text{-}alg} : \mathit{Forest}\ (\mathit{Bush}\ A) \longrightarrow \mathit{Bush}\ A \\ \\ &\mathit{join\text{-}coh} : (x\ y : \mathit{Forest}\ (\mathit{Bush}\ A)) \longrightarrow \\ &\qquad \mathit{Equiv\text{-}UpTo}\ x\ y \longrightarrow \\ &\qquad \mathit{join\text{-}alg}\ x \equiv \mathit{join\text{-}alg}\ y \end{aligned}$$

The *join-alg* function is the problem here: it needs to somehow recurse through the coinductive type *Forest*, and extract the quotients from its internals. It is not easy (not possible, we conjecture) to write a terminating implementation of such a function. Chapman et al. [2019, section 5] explains the problem in more detail: at some point, a kind of choice function is needed.

5.3 An Indexed Representation

Our final representation of search spaces will be derived directly from the $\rightarrowtail$ Eq. (16) operator, and specifically from the theory of semigroup actions.

Definition 5.2 (Semigroup Action). A (right) semigroup action for a semigroup S and a set A is an operator $\cdot : A \to S \to A$ such that:

$$\forall x, y, z.\ (x \cdot y) \cdot z = x \cdot (y \bullet z) \tag{22}$$

Weights form a semigroup under the min ($\sqcap$) operation, and the bounding operator $\rightarrowtail$ Eq. (16) implements a corresponding semigroup action.

$$\forall\ s\ v\ w \longrightarrow (s \rightarrowtail v) \rightarrowtail w \equiv s \rightarrowtail (v \sqcap w) \tag{23}$$

There is also such a thing as a monoid and a group action; these must follow the same law as the semigroup action Eq. (22), as well as the further law regarding the neutral element:

$$\forall x.\ x \cdot \epsilon = x \tag{24}$$

The $\bullet$ monoid on weights implements a (left) monoid action with $\rtimes$ Eq. (5).

There is a representation theorem for group actions that we can use to derive a representation of weighted sets. To get to this representation theorem we will need some category theory.

Definition 5.3 (S-Sets). For a semigroup S, there is a category S-Set of semigroup actions. The objects of this category are sets acted upon by S, and the morphisms are *equivariant maps*, which are functions between sets $f : X \to Y$ that commute with the actions:

$$\forall x, y.\ f x \cdot y = f(x \cdot y)$$

Any semigroup actually acts on itself, where $\cdot = \bullet$. As a result, S is an object in S-Set. Similarly, any monoid M is an object in its own M-Set. The monoid object has the special property of being a *representation* for the forgetful functor. This means that for a given object X, the arrows $\|M\| \to X$ (where $\|M\|$ is the object for the monoid M) is isomorphic to $|X|$ (where $|X|$ is the underlying set for the object X). To make this concrete, given the following definition of arrows:

$$X \longrightarrow Y = \Sigma[\ f : |\ X\ | \to |\ Y\ |\ ] \times \forall\ x\ y \to f\ x \cdot y \equiv f\ (x \cdot y)$$

we have the isomorphism:

$$(\|\ M\ \| \longrightarrow X) \Leftrightarrow |\ X\ |$$

Now, recall that there is a semigroup action on the *Weighted* type. This representation theorem seems to suggest that there is an isomorphic representation using the indexing operation. However, there's an issue: weights are not (necessarily) a monoid. There is no neutral element, no ∞ such that $\forall x.\ x \sqcap \infty = x$. This means that one direction of the isomorphism fails: we only have a monomorphism $\mathit{Weighted}\ A \rightarrowtail (\|W\| \longrightarrow \mathit{Weighted}\ A)$.

Far from being a problem, though, the lack of an inverse properly captures the desired semantics of the representation. The *Weighted* type is inductive, remember, and we're looking for a way to represent its coinductive variant. This coinductive variant should be larger than the inductive: that's exactly what's expressed by the monomorphism.

From another perspective, if there were a largest weight ∞, we would be able to apply the function $\|W\| \longrightarrow \mathit{Weighted}\ A$ to it, and get back the corresponding *Weighted A*. But if this function represents some infinite search, it won't *fit* in the inductive type *Weighted A*.

All of this together means that $\|W\| \longrightarrow \mathit{Weighted}\ A$ is a good representation of a coinductive *Weighted*. The actual type corresponding to $\|W\| \longrightarrow \mathit{Weighted}\ A$ is as follows:

$$
\begin{array}{l}
\textbf{record}\ \mathit{Neighbours}\ A\ \textbf{where}\\
\quad \textbf{field}\ \_{\ggg}\_ : W \to \mathit{Weighted}\ A\\
\qquad\quad \mathit{neighbourly} : \mathit{Neighbourly}\ \_{\ggg}\_
\end{array}
\qquad
\begin{array}{l}
\mathit{Neighbourly} : (W \to \mathit{Weighted}\ A) \to \mathit{Type}\\
\mathit{Neighbourly}\ f =\\
\quad \forall\ v\ w \to v \geqslant w \to f\ v \gg w \equiv f\ w
\end{array}
$$

From a high level, *Neighbours* represents a coinductive search routine: given a weight, it performs a search, returning all the results within the supplied weight bound. It is a (dependent) pair, where the first component, named $\ggg$, is a function $W \to \mathit{Weighted}\ A$ that takes a weight and returns the weighted set of all values that weigh less than the supplied weight. This function is named to be reminiscent of the $\gg$ function (Eq. (16); note the different number of vertical bars). Since $\ggg$ is a field in a record, as a *function* $\ggg$ has type $\mathit{Neighbours}\ A \to W \to \mathit{Weighted}\ A$ (compare to $\gg\ : \mathit{Weighted}\ A \to W \to \mathit{Weighted}\ A$).

The second component of the type *Neighbours* is a proof that the first function is "*Neighbourly*". This is a coherency condition: it ensures that the first component, the $\ggg$ function, is well-behaved. Semantically, $\ggg$ should behave like a partially-applied $\gg$: it should return all values in its search space with a weight smaller than the supplied argument. However, we can imagine some badly-behaved function that returns different values at different weights ($\lambda w.$ **if** $w \equiv 1$ **then** $\wr 1 \triangleright x \int$ **else** $\wr 1 \triangleright y \int$). *Neighbourly* is a predicate that prohibits such functions (as well as other incoherencies); it is actually equivalent to the predicate $\forall x, y.\ f\ x \cdot y = f(x \cdot y)$, the coherence condition on arrows in S-Set. It is slightly easier to work with *Neighbourly* in this context, however.

To define a value that inhabits the *Neighbours* type, we have to implement the search routine, and show that it is "*Neighbourly*". One such routine is the trivial search, which always returns a single element with zero weight.

$$
\begin{array}{l}
\eta : A \to \mathit{Neighbours}\ A\\
\eta\ x \ggg \_ = \wr\ \epsilon \triangleright x\ \int\\
\eta\ x\ .\mathit{neighbourly}\ v\ w\ v{\geqslant}w = \eta\text{-}\mathit{lemma}
\end{array}
$$

The search routine here is a constant function that returns a singleton weighted set containing x. The *neighbourly* proof has type $\wr \eta \triangleright x \int \gg w \equiv \wr \eta \triangleright x \int$, and is given by *η-lemma*.

A slightly more complex function is the one that searches a finite weighted set.

$$
\begin{aligned}
&\mathit{searched} : \mathit{Weighted}\ A \longrightarrow \mathit{Neighbours}\ A\\
&\mathit{searched}\ \mathit{xs} \mathbin{\succ\!\!\|}\ w = \mathit{xs} \mathbin{\succ\!\!|}\ w
\end{aligned}
$$

This function (*searched*) converts a finite weighted set into a value of type *Neighbours*. The routine here searches the supplied set (*xs*) to the given depth, by using the cutoff operator we have defined already Eq. (16). The coherence proof has the type $\mathit{xs} \mathbin{\succ\!\!|} v \mathbin{\succ\!\!|} w \equiv \mathit{xs} \mathbin{\succ\!\!|} w$ (given that $v \geqslant w$), but we will elide these proofs from now on in the text except where relevant (they are present in our formalisation).

Crucially, addressing the problems raised in Section 5.2, *Neighbours* is a monad. We have already seen η; *join* (μ) is more difficult. To collapse two layers of *Neighbours* to a weight w, we restrict the outer layer by w, yielding *Weighted* (*Neighbours A*); then we use the monadic bind on this outer *Weighted*, supplying the continuation that restricts the inner layer by w. Finally, we restrict the resulting set by w once again.

$$
\begin{aligned}
&\mu : \mathit{Neighbours}\ (\mathit{Neighbours}\ A) \longrightarrow\\
&\quad \mathit{Neighbours}\ A\\
&\mu\ s \mathbin{\succ\!\!\|}\ w = s \mathbin{\succ\!\!\|}\ w \mathbin{\succ\!\!|\!\!\|}\ w \mathbin{\succ\!\!|}\ w
\end{aligned}
$$

$$
\begin{aligned}
&\_\!\mathbin{\succ\!\!|\!\!\|}\!\_ : \mathit{Weighted}\ (\mathit{Neighbours}\ A) \longrightarrow\\
&\qquad W \longrightarrow \mathit{Weighted}\ A\\
&s \mathbin{\succ\!\!|\!\!\|}\ w = s \gg\!= \ \_\!\mathbin{\succ\!\!\|}\ w
\end{aligned}
$$

Though this operation seems intricate, implementing it is largeley a case of applying the cutoff operator repeatedly until the types line up. Similarly, the coherence condition looks complicated:

$$\forall\ v\ w \longrightarrow v \geqslant w \longrightarrow s \mathbin{\succ\!\!\|}\ v \mathbin{\succ\!\!|\!\!\|}\ v \mathbin{\succ\!\!|}\ v \mathbin{\succ\!\!|}\ w \equiv s \mathbin{\succ\!\!\|}\ w \mathbin{\succ\!\!|\!\!\|}\ w \mathbin{\succ\!\!|}\ w$$

But its proof requires only a little ingenuity, and some tedious applications of the monad laws and semigroup action laws.

5.4 Coinduction on Indices

The final piece of the puzzle to make *Neighbours* a usable data structure for graphs is to implement coinduction using it. This amounts to showing that *Neighbours* is a CIM.

The ideal, $\overline{\mathit{Neighbours}}_s$, is a weighted set where every member has weight at *least* equal to some minimum amount s. Defining such a type head-on, in the obvious way, turns out to be quite difficult. Consider the set $\lbag 2 \triangleright x, 5 \triangleright y \rbag$: is this a valid member of $\overline{\mathit{Neighbours}}_3$? What if $x \equiv y$? (recall that $p \triangleright x :: q \triangleright x :: \mathit{xs} \equiv p \sqcap q \triangleright x :: \mathit{xs}$) Is it possible to answer this question without decidable equality on the entries?

The solution is to represent $\overline{\mathit{Neighbours}}_s$ *implicitly*. First, define the operator $\rtimes_n$, which behaves like $\rtimes$ Eq. (5), but defined on *Neighbours* rather than *Weighted*. $w \rtimes_n x$ adds w to every entry in x.

$$
\begin{aligned}
&\_\!\rtimes_n\!\_ : W \longrightarrow \mathit{Neighbours}\ A \longrightarrow \mathit{Neighbours}\ A\\
&(w \rtimes_n s) \mathbin{\succ\!\!\|}\ v\ \textbf{with}\ w \leqslant?\ v\\
&\ldots \mid \mathit{yes}\ (v \dot{-} w\ ,\ \_) = w \rtimes s \mathbin{\succ\!\!\|}\ v \dot{-} w\\
&\ldots \mid \mathit{no}\ \_ = \lbag\rbag
\end{aligned}
$$

Then, a *Neighbours* set with every entry heavier than w must be equal to some lighter set with w added to it, giving the following ideal:

$$
\begin{aligned}
&\mathit{Heavier} : W \longrightarrow \mathit{Neighbours}\ A \longrightarrow \mathit{Type}\\
&\mathit{Heavier}\ w\ x = \exists\ \mathit{lighter} \times x \equiv w \rtimes_n \mathit{lighter}
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathit{Neighbours}}_s\ A =\\
&\quad \Sigma[\ x : \mathit{Neighbours}\ A\ ] \times \mathit{Heavier}\ s\ x
\end{aligned}
$$

However, this ideal is in fact *isomorphic* to *Neighbours*. Instead of representing the ideal as the whole sigma, we will represent it as just the *lighter* weighted set.

Instead of storing a collection xs and a proof that all of the weights in xs are greater than some weight s, we will store a collection where the weights are the *differences* between the actual weights and the minimum. For example, the collection $\wr 3 \triangleright x, 5 \triangleright y \wr$ with minimum 2 is represented by the actual value $\wr 1 \triangleright x, 3 \triangleright y \wr$. This ideal then implements σ as follows:

$$\sigma : \overline{Neighbours}_s\ A \longrightarrow Neighbours\ A \qquad\qquad \sigma\ (\wr 1 \triangleright x, 3 \triangleright y \wr : \overline{Neighbours}_2) = \wr 3 \triangleright x, 5 \triangleright y \wr$$
$$\sigma\ x = s \rtimes_n x$$

To show that this constructed ideal makes *Neighbours* a CIM, we need to establish a solution function. Concretely, this is a function, given some $s : W$ and $s \not\equiv \epsilon$:

$$solve : (X \longrightarrow \overline{Neighbours}_s\ (X \uplus A) \uplus A) \longrightarrow X \longrightarrow Neighbours\ A$$

And further we must show that is an actual solution. Formally:

$$\forall\ x \longrightarrow solve\ e_i\ x \equiv (\mu \circ map_n\ (solve\ e_i \triangledown \eta) \circ (\sigma \triangledown \eta \circ inr) \circ e_i)\ x$$

The proof of this lemma is in our formalisation.

Note that when this formulation is used to implement, for instance, $*$, we call $*$ on the *ideal* of the graph, not the graph itself.

Summary. In this section, we have tried to construct a quotiented form of the coinductive data structures presented in the previous section. We have been partially successful: we have seen the *Bush* type, which does faithfully represent a coinductive search space, and indeed could be used to efficiently implement search algorithms. However, this type is only a monad when countable choice holds. We then looked at the *Neighbours* type, which gives an inductive interface to coinductive algorithms. This type had a monad instance, and was able to perform the search algorithms, and supported coinduction through the CIM framework.

6 Case Studies

Having presented the theory for coinductive graphs, we will now look at some case studies of using our approach to graphs to implement some standard graph algorithms. In this section we will use some Haskell to illustrate how our general approach can be adapted to non-total languages while still preserving some of the valuable algebraic structure from our Agda library.

6.1 Topological Sort

For our first algorithm, we will look at topological sort. The Haskell representation of graphs we will use is the following:

```
type GraphOf a = a → [a]
```

This is a representation of *unweighted* graphs, where neighbours are represented with a list. We are going to implement topological sort, so weights are unnecessary. Since lists preserve the order of their contents, to convert to this form from the Agda representation in Eq.(4) we would need a total order on the vertices. Such an order would always be required to implement topological sort, to break ties in the sorting algorithm, so no generality has been lost.

The implementation of topological sort is simple, but subtle. Here we provide both the Haskell and Agda versions:

```
topoSort :: ∀a.Ord a ⇒ GraphOf a
           → [a] → [a]
topoSort g = fst ∘ sortF ([ ], ∅)
  where
  sortF :: ([a], Set a) → [a] →
           ([a], Set a)
  sortF = foldr sortT

  sortT :: a → ([a], Set a)
             → ([a], Set a)
  sortT v (sorted, seen) =
    if v ∈ seen
      then (sorted, seen)
      else first (v:)
             (sortF (sorted, {v} ∪ seen) (g v))
```

```
topo-sort :  GraphOf A
           → List A → List A
topo-sort g = fst ∘ sort-f ([] , ∅) ∘ trace g
  where mutual
  sort-f : List A × 𝒦 A → Forest A →
           List A × 𝒦 A
  sort-f ac []       = ac
  sort-f ac (n :: ns) = sort-t n (sort-f ac ns)

  sort-t : Tree A → List A × 𝒦 A
                  → List A × 𝒦 A
  sort-t (v & cs) (sorted , seen) =
    if does (v ∈? seen)
      then (sorted , seen)
      else map₁ (v ::_)
             (sort-f (sorted , v :: seen) cs)
```

First, let's explain the type signature. Since our representation doesn't attach a collection of vertices to every graph that collection has to be provided separately. Consequently, the type of *topoSort* takes a graph and a list of vertices to be sorted. Rather than being a downside, however, we think this restriction actually *improves* the clarity of the types: this type for *topoSort* shows that the algorithm transforms a graph into a sorting function on lists.

The algorithm itself proceeds by folding right over the supplied list (*sortF*), accumulating (from the right) a set of already-seen vertices. For every new vertex v encountered, if it is not in the set of already-seen vertices, it is consed to the output, and then the sorting function is recursively called on its neighbours ($g\ v$).

Notice that the recursion pattern here is quite complex: output is built from the left, with the leftmost vertex in the input appearing first in the output list. However, the crossing-off of already-seen vertices is done from the right. Furthermore, the recursive call takes as an argument the updated *seen* set, but its output is placed *after* the vertex inserted into that set.

To implement this in Agda we have to deal with this complex termination issue, while preserving the structure of the algorithm. As is clear above, both algorithms work quite similarly: one notable difference is that the Agda implementation cannot use a higher-order function like *foldr* because that would obscure the structural recursion from the termination checker. Another difference is that the Agda version uses *trace*; this converts a graph to a finite tree, so that we can use the tree to bound termination. We convert the graph to a finite tree using Noetherian finiteness [Firsov et al. 2016], here implemented as an inductive data type:

```
data NoethAcc (seen : 𝒦 A) : Type a where
  nacc : (∀ x → x ∈ Dom → x ∉ seen → NoethAcc (x :: seen)) → NoethAcc seen
```

This is actually a special case of the *Acc* data type we have seen already.

This case study demonstrates that our neighbours-based graph representation, though simple, can still be used to implement traditional algorithms, even non-search algorithms.

6.2 Dijkstra

Many of the standard graph algorithms can be easily expressed as instances of transitive closure. We saw in the introduction (Section 1) that Hamiltonian paths were one such instance; here we will sketch how Dijkstra's algorithm can be expressed in the same way.

From a high level, the steps involved in these algorithms are quite similar: like with Hamiltonian paths, we first produce the transitive closure of all paths through the graph, filtering out cycles, using *pathed* and $*$. This produces a graph of all the loop-free paths through the graph.

Where the Hamiltonian paths were given by restricting the output to only paths which covered the entire graph, we implement Dijkstra by restricting the paths to those with a specified end-point.

$$\begin{aligned}&\mathit{dijkstra} : \mathit{Vert} \longrightarrow \mathit{GraphOf}\ \mathit{Vert} \longrightarrow \mathit{Neighbours}\ (\mathit{List}^{+}\ \mathit{Vert})\\&\mathit{dijkstra}\ s\ g = ((\mathit{pathed}\ g \ggg \mathit{filtering}\ \mathit{uniq})^{*})\ [\ s\ ]\end{aligned}$$

From here, it is not too difficult to extract the shortest path from this *Neighbours* structure.

To get an efficient implementation, we can turn to the heap structure from Section 4.1. A version of this algorithm was already presented in [Kidney and Wu 2021]. Here, the pairing heap will allow us to efficiently extract the shortest paths in question. Note that this requires a slightly different instantiation of graphs, one that does not have the quotiented structure.

7 Related Work

7.1 Comparison to Other Haskell Approaches

While the focus of this paper is on the formalisation and theory of graph algorithms, we think that our algebraic treatment of graphs (Section 3) could have a lot of practical use for the Haskell programmer. As such, in this subsection we will briefly compare our approach to other major Haskell graph treatments. For a more in-depth study of the algorithmic aspects of our approach in Haskell, we direct the interested reader to our earlier work [Kidney and Wu 2021].

As a running example for this subsection, we will translate Eq. (7) to Haskell on unweighted graphs, giving depth-first search.

$$\begin{aligned}&\mathit{dfs} :: \mathrm{GraphOf}\ a \longrightarrow \mathrm{GraphOf}\ a\\&\mathit{dfs}\ g = \mathbb{1} \oplus (g \ggg \mathit{dfs}\ g)\end{aligned}$$

Of course, this implementation and representation does not grapple with quotients or coinduction, but since Haskell doesn't support quotients and does not enforce productivity this is the most faithful translation available to us.

Algebraic Graphs with Class. Mokhov [2017] is perhaps the paper closest in spirit to ours: there, graphs are described algebraically, with the *Graph* class. Mokhov's approach differs from ours primarily in the treatment of vertices: their representation of graphs is a data structure which contains a concrete collection of vertices. Our representation represents vertices as a *type*: while more general, it does preclude us from writing a generic function that traverses all the vertices of a particular graph. Furthermore, the titular algebra of graphs is a single semiring-like algebra that manipulates both edges and vertices, whereas we have two separate algebras (the edge (Section 3.1) and vertex semiring Section 3.5). For example, their `overlay` operation constructs a graph by taking the union of the edges and vertices of its operands, whereas our overlay ($\boxplus$, Section 3.1) takes the union of the edges of two graphs who have the same vertices.

The primary advantage of our work over Mokhov [2017] is that we can handle algorithms like depth-first search directly. Our definition of depth-first search is $O(n)$ already, whereas the to implement depth-first search in Mokhov [2017] they have to convert to another graph representation first. Mokhov [2017] is focused on graph *construction* rather than search algorithms, so this is no great surprise; hopefully our work can be seen as complementary, building on the algebraic approach while preserving algorithmic efficiency.

Functional Graph Library. The FGL library and paper [Erwig 2001, 2008] is perhaps the best-known functional approach to graphs: the approach presented there is in a sense the inverse of the one presented in this paper. There, graphs are given a representation as an inductive data type, like lists: algorithms are then expressed as inductive recursive functions over this type. There are clearly many advantages to using an inductive type: these types are well-supported and well-understood, and algorithms over them are clear and simple to understand. One particularly notable advantage of FGL over our approach is the automation of the removal of "already-seen vertices" in depth-first search implementations. Here is their implementation of depth-first search:

$$\begin{array}{llll} dfs :: [\,\mathsf{Node}\,] \to \mathsf{Graph}\ a\ b \to \mathsf{Node} & & & \\ dfs\ [\,]\ & g & = [\,] \\ dfs\ (v : vs)\ & (c\ \&^{v}\ g) & = v : dfs\ (suc\ c \mathbin{+\!\!+} vs)\ g \\ dfs\ (v : vs)\ & g & = dfs\ vs\ g \end{array}$$

The expression $dfs\ \ vs\ \ g$ takes a stack of vertices vs, and a graph g, and searches the stack of vertices through the graph in depth-first order. The second clause in the function uses the special constructor $\&^{v}$, which matches the node for the vertex v. The remaining bound graph, g, has that vertex v removed.

We direct the reader to Kidney and Wu [2021] to see our approach to avoiding already-seen vertices in Haskell (we use monad transformers), however we feel that the more important difference between our approach and FGL is that we preserve the algebraic treatment of graphs even when implementing algorithms. As well as this, our graph type has a more solid formal grounding (in terms of quotients and formalisation), but that is not so useful to a Haskell programmer. We think the main advantage of our approach is that graphs and algorithms are presented algebraically: like Mokhov [2017], we present an algebra of graphs, which can be used to construct and define graphs; but we go further and extend this same algebra to define graph algorithms.

Gibbons [1995] also treats graphs as an inductive type using initial algebras, and explores various graph algorithms as catamorphisms. However, that work is limited to acyclic graphs.

Structuring Depth-First Search Algorithms in Haskell. King and Launchbury [1995] provides a number of example implementations of graph algorithms in Haskell, using depth-first search as the central reusable algorithm. The containers library [Feuer 2022] bases its graph module on the algorithms in this paper. Our approach is similar to King and Launchbury [1995] in that we also use depth-first search (or, more specifically, transitive closure), as a core, reusable algorithm. Our approach differs in our graph representation. King and Launchbury's representation is a simple array-backed adjacency list:

$$\textbf{type}\ \mathsf{Graph'} = \mathsf{Array}\ \mathsf{Int}\ [\,\mathsf{Int}\,]$$

However, while our representations differ, it is in fact possible to use our approach on the above representation without paying anything for the conversion. The conversion function is simply indexing:

$$\begin{array}{l} convert :: \mathsf{Graph'} \to \mathsf{GraphOf}\ \mathsf{Int} \\ convert = (!) \end{array}$$

As such, functions like dfs above work as-is.

The focus of King and Launchbury [1995] is quite different from ours, being more concerned with the efficiency of implementation of certain algorithms, but we would like to implement the algorithms there on our framework in future work.

7.2 Algebraic Graphs

Another algebraic approach to graphs is given by Master [2021, 2022]. These papers generalise some of the constructions presented here, although they also deal with shortest path problems.

The idea of treating graph algorithms as semiring-based problems can be traced back to Backhouse and Carré [1975]; Conway [1971]; although those approaches focus on matrices.

Kidney and Wu [2021] contains many constructions that are built upon in this paper. In particular, monuses for weight, the *Weighted* set, and a version of the *Forest* type (as a pairing heap) all have versions present in that paper (albeit slightly different versions). However, that paper does not deal with coinduction, or the problem of quotienting coinductive structures.

The edge and vertex semiring in this paper are present (in a slightly different form) in the Arrow library [Hughes 2000; Paterson 2003].

The papers Liell-Cock and Schrijvers [2024]; Mokhov [2022] expand on the algebraic graph treatment in Mokhov [2017]. The spirit of these papers is very similar to this work, especially in that both approaches hold algebraic reasoning in high regard. However, our choice to represent vertices as a type is a major design difference with knock-on effects. As such, while we would like to incorporate the sophisticated algebraic structure developed in these papers into our work it is not clear how to do so at the moment.

7.3 Agda and Coinduction

We used Cubical Agda to formalise our work [Vezzosi et al. 2021] because of its support for quotients which are used in our representation of graphs, and facilities for functional programming. In the future, perhaps Liquid Haskell [Vazou et al. 2014] or Quotient Haskell [Hewer and Hutton 2024] could be viable settings for similar work.

Picard and Matthes [2011] also deals with the problem of formalising graphs, and especially focuses on formalising *coinductive* graphs. The type of graphs given in that paper is equivalent to the *Heap* type: they do not deal with quotients to the same extent as this paper.

There a few facilities for coinduction in Agda [Abel and Pientka 16ed]. In HoTT, coinduction is arguably better supported than in MLTT [Ahrens et al. 2015]. It is difficult, though tractable, to combine quotients with coinductive types [Chapman et al. 2019], and Cubical Agda has made things a little easier [Joram and Veltri 2023; Veltri and Vezzosi 2023].

The coinductive resumption monad, and associated machinery of cims [Piróg and Gibbons 2014], underpins much of the work on coinduction in this paper.

8 Conclusion

We do not think that the *GraphOf* type should be the *only* representation for writing graph algorithms; many of the other representations have significant advantages, depending on the situation. However, we do think that it is a good default that has been largely overlooked: Even its unquotiented, unweighted counterpart supports the semirings described in this paper, and enjoys many of the same properties as the fancier version. Most importantly, though, we hope that this paper motivates programmers to get out of their inductive comfort zone when the situation demands it: don't avoid coinduction, embrace it! Be lazy!

References

Andreas Abel and Brigitte Pientka. 2016/ed. Well-Founded Recursion with Copatterns and Sized Types. *Journal of Functional Programming* 26 (2016/ed). https://doi.org/10.1017/S0956796816000022

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2429069.2429075

Peter Aczel, Jiří Adámek, Stefan Milius, and Jiří Velebil. 2003. Infinite Trees and Completely Iterative Theories: A Coalgebraic View. *Theoretical Computer Science* 300, 1 (May 2003), 1–45. https://doi.org/10.1016/S0304-3975(02)00728-4

Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. 2015. Non-Wellfounded Trees in Homotopy Type Theory. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.TLCA.2015.17*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.TLCA.2015.17 arXiv:1504.02949

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

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Transactions on Programming Languages and Systems* 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712

Roland Backhouse. 1975. *Closure Algorithms and the Star-Height Problem of Regular Languages.* Ph. D. Dissertation. Imperial College London. https://cir.nii.ac.jp/crid/1570854175242787968

R. C. Backhouse and B. A. Carré. 1975. Regular Algebra Applied to Path-finding Problems. *IMA Journal of Applied Mathematics* 15, 2 (April 1975), 161–186. https://doi.org/10.1093/imamat/15.2.161

Roland C. Backhouse, J. P. H. W. van den Eijnde, and A. J. M. van Gasteren. 1994. Calculating Path Algorithms. *Science of Computer Programming* 22, 1 (April 1994), 3–19. https://doi.org/10.1016/0167-6423(94)90005-1

Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, Vol. 50. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2. arXiv:1606.05223 http://arxiv.org/abs/1606.05223

James Chapman, Tarmo Uustalu, and Niccolò Veltri. 2019. Quotienting the Delay Monad by Weak Bisimilarity. *Mathematical Structures in Computer Science* 29, 1 (Jan. 2019), 67–92. https://doi.org/10.1017/S0960129517000184

John Horton Conway. 1971. *Regular Algebra and Finite Machines.* Chapman and Hall, London.

Stephen Dolan. 2013. Fun with Semirings: A Functional Pearl on the Abuse of Linear Algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13, Vol. 48)*. ACM, New York, NY, USA, 101–110. https://doi.org/10.1145/2500365.2500613

Calvin C. Elgot, Stephen L. Bloom, and Ralph Tindell. 1978. On the Algebraic Structure of Rooted Trees. *J. Comput. System Sci.* 16, 3 (June 1978), 362–399. https://doi.org/10.1016/0022-0000(78)90024-7

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

Martin Erwig. 2008. Functional Graph Library/Haskell. http://web.engr.oregonstate.edu/~erwig/fgl/haskell/

David Feuer. 2022. Containers: Assorted Concrete Container Types. Haskell. https://hackage.haskell.org/package/containers

Denis Firsov, Tarmo Uustalu, and Niccolò Veltri. 2016. Variations on Noetherianness. *Electronic Proceedings in Theoretical Computer Science* 207 (April 2016), 76–88. https://doi.org/10.4204/EPTCS.207.4 arXiv:1604.01186

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

Neil Ghani and Alexander Kurz. 2007. Higher Dimensional Trees, Algebraically. In *Algebra and Coalgebra in Computer Science (Lecture Notes in Computer Science)*, Till Mossakowski, Ugo Montanari, and Magne Haveraaen (Eds.). Springer, Berlin, Heidelberg, 226–241. https://doi.org/10.1007/978-3-540-73859-6_16

Neil Ghani, Christoph Lth, Federico De Marchi, and John Power. 2003. Dualising Initial Algebras. *Mathematical Structures in Computer Science* 13, 2 (April 2003), 349–370. https://doi.org/10.1017/S0960129502003912

Jeremy Gibbons. 1995. An Initial-Algebra Approach to Directed Acyclic Graphs. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*, Bernhard Möller (Ed.). Springer, Berlin, Heidelberg, 282–303. https://doi.org/10.1007/3-540-60117-1_16

Jeremy Gibbons and Graham Hutton. 1999. Proof Methods for Structured Corecursive Programs. In *Proceedings of 1st Scottish Workshop on Functional Programming.*

Brandon Hewer and Graham Hutton. 2024. Quotient Haskell: Lightweight Quotient Types for All. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 785–815. https://doi.org/10.1145/3632869

Ralf Hinze. 2012. Kan Extensions for Program Optimisation or: Art and Dan Explain an Old Trick. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 324–362. https://doi.org/10.1007/978-3-642-31113-0_16

John Hughes. 2000. Generalising Monads to Arrows. *Science of Computer Programming* 37, 1 (May 2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

Philipp Joram and Niccolò Veltri. 2023. Constructive Final Semantics of Finite Bags. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2023.20*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2023.20

Donnacha Oisín Kidney and Nicolas Wu. 2021. Algebras for Weighted Search. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 72:1–72:30. https://doi.org/10.1145/3473577

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

Jack Liell-Cock and Tom Schrijvers. 2024. Let a Thousand Flowers Bloom: An Algebraic Representation for Edge Graphs. *The Art, Science, and Engineering of Programming* 8, 3 (Feb. 2024), 9. https://doi.org/10.22152/programming-journal.org/2024/8/9 arXiv:2403.02273 [cs]

Jade Master. 2021. The Open Algebraic Path Problem. https://doi.org/10.48550/arXiv.2005.06682 arXiv:2005.06682

Jade Master. 2022. How to Compose Shortest Paths. https://doi.org/10.48550/arXiv.2205.15306 arXiv:2205.15306

Stefan Milius. 2005. Completely Iterative Algebras and Completely Iterative Monads. *Information and Computation* 196, 1 (Jan. 2005), 1–41. https://doi.org/10.1016/j.ic.2004.05.003

Andrey Mokhov. 2017. Algebraic Graphs with Class (Functional Pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/3122955.3122956

Andrey Mokhov. 2022. United Monoids: Finding Simplicial Sets and Labelled Algebraic Graphs in Trees. *The Art, Science, and Engineering of Programming* 6, 3 (Feb. 2022), 12:1–12:23. https://doi.org/10.22152/programming-journal.org/2022/6/12

Ross Paterson. 2003. Arrows and Computation. In *The Fun of Programming*, Jeremy Gibbons and Oege de Moor (Eds.). Palgrave, 201–222. http://www.soi.city.ac.uk/~ross/papers/fop.html

Célia Picard and Ralph Matthes. 2011. Coinductive Graph Representation: The Problem of Embedded Lists. *Electronic Communications of the EASST* 39 (2011). https://hal-enac.archives-ouvertes.fr/hal-02015853

Maciej Piróg and Jeremy Gibbons. 2014. The Coinductive Resumption Monad. *Electronic Notes in Theoretical Computer Science* 308 (Oct. 2014), 273–288. https://doi.org/10.1016/j.entcs.2014.10.015

Exequiel Rivas, Mauro Jaskelioff, and Tom Schrijvers. 2015. From Monoids to Near-Semirings: The Essence of MonadPlus and Alternative. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. ACM, 196–207. https://doi.org/10.1145/2790449.2790514

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Niccolò Veltri and Andrea Vezzosi. 2023. Formalizing CCS and π-Calculus in Guarded Cubical Agda. *Journal of Logical and Algebraic Methods in Programming* 131 (Feb. 2023), 100846. https://doi.org/10.1016/j.jlamp.2022.100846

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2021. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Journal of Functional Programming* 31 (Jan. 2021), e8. https://doi.org/10.1017/S0956796821000034