# Graphical algebraic foundations for monad stacks

a functional pearl

Ohad Kammar
University of Cambridge
ohad.kammar@cl.cam.ac.uk

## Abstract

HASKELL incorporates computational effects modularly using sequences of monad transformers, termed *monad stacks*. The current practice is to find the appropriate stack for a given task using intractable brute force and heuristics. By restricting attention to *algebraic* stack combinations, we provide a linear-time algorithm for generating all the appropriate monad stacks, or decide no such stacks exist. Our approach is based on Hyland, Plotkin, and Power's algebraic analysis of monad transformers, who propose a graph-theoretical solution to this problem. We extend their analysis with a straightforward connection to the *modular decomposition* of a graph and to *cographs*, a.k.a. *series-parallel graphs*.

We present an accessible and self-contained account of this monad-stack generation problem, and, more generally, of the decomposition of a combined algebraic theory into sums and tensors, and its algorithmic solution. We provide a web-tool implementing this algorithm intended for semantic investigations of effect combinations and for monad stack generation.

***Categories and Subject Descriptors*** Theory of computation [*Semantics and reasoning*]: Program semantics, program constructs; Mathematics of computing [*Discrete mathematics*]: Graph theory.

***General Terms*** Languages, Theory

***Keywords*** algebraic theory of effects, monad stacks, monad transformers, algebraic effects, sum of theories, tensor of theories, series-parallel graphs, denotational semantics cographs, modular decomposition of graphs.

## 1. Introduction

Computational effects, such as modifying and accessing a global state, or raising an exception, must currently be explicitly declared in HASKELL using monads (Wadler 1995). For example, computations involving a global state cell of type $\mathbb{S}$, and computations that may throw an exception of type $\mathbb{E}$, are represented, respectively, using the following monads:

$$\mathbf{type}\ \mathbf{State}\ \mathbb{S}\ a = \mathbb{S} \to (\mathbb{S}, a)$$
$$\mathbf{data}\ \mathbf{Error}\ \mathbb{E}\ a = \mathit{Throw}\ \mathbb{E} \mid \mathit{Return}\ a$$



*ICFP '14*, September 1–3, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Monads are not modular in the sense that combining effects does not amount to composing monads. For example, the monad for combining two different global state cells is $\mathbf{State}\ (\mathbb{S}, \mathbb{S})\ a$ rather than $\mathbf{State}\ \mathbb{S}\ (\mathbf{State}\ \mathbb{S}\ a)$. To allow some form of modularity, functional programmers use *monad transformers* (Liang et al. 1995): mappings that construct a monad out of any given monad suitable for implementing the additional effects. For example, given any monad $M$, the state transformer and the exception monad transformers are given, respectively, as follows:

$$\mathbf{type}\ \mathbf{StateT}\ \mathbb{S}\ m\ a = \mathbb{S} \to m\ (\mathbb{S}, a)$$
$$\mathbf{type}\ \mathbf{ErrorT}\ \mathbb{E}\ m\ a = m\ (\mathbf{Error}\ \mathbb{E}\ a)$$

Monad transformers are modular, and may be composed in any order. However, the order of composition matters. For example, computations which may abort due to an exception maintaining the state they reached when the exception occurs can be expressed by the following transformation of the identity monad:

$$(\mathbf{ErrorT} \circ \mathbf{StateT})\ (\mathbf{Id})\ a \equiv \mathbb{S} \to (\mathbb{S}, (\mathbf{Error}\ \mathbb{E}\ a))$$

Composing the transformers in the opposite order expresses computations that discard the global state when an exception occurs:

$$(\mathbf{StateT} \circ \mathbf{ErrorT})\ (\mathbf{Id})\ a = \mathbb{S} \to \mathbf{Error}\ \mathbb{E}\ (\mathbb{S}, a)$$

More generally, a *monad stack* is a sequence of monad transformers. Given a collection of $n$ monad transformers, there are $n!$ monad stacks. Some of these stacks give rise to the same monad transformer, e.g., the following two transformers are equivalent:

$$\mathbf{ErrorT}\ \mathbb{E}' \circ \mathbf{ErrorT}\ \mathbb{E} \equiv \mathbf{ErrorT}\ \mathbb{E} \circ \mathbf{ErrorT}\ \mathbb{E}'$$

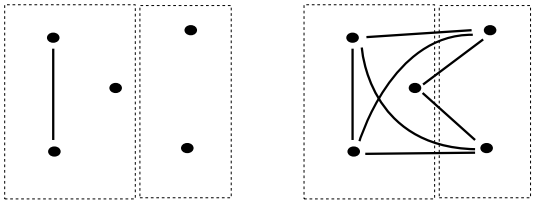
Even a modest combination of 7 effects yields $5,040$ different stacks. We would like an efficient systematic method to choose the appropriate one, if it even exists.

Our solution requires manually considering the $n(n-1)/2$ interactions between every pair of monad transformers instead. Once these interactions are determined, we have a straightforward linear-time algorithm for deciding whether there exists a stack realising this stack, and if so, generating it.

The crux of the solution is given by Hyland, Plotkin, and Power in their algebraic analysis of monad transformers (Hyland et al. 2006). This analysis relies on the established idea that monads arise from equational presentations[1]. In this paper, we will present such presentations in HASKELL monadic notation. For example, the presentation for a global state $\mathbb{S}$ arises from the signature involving two parameterised operations $\mathbf{lookup} :: () \to m\ \mathbb{S}$ and $\mathbf{update} :: \mathbb{S} \to m\ ()$, and equations such as the following:

$$\mathbf{update}\ s_0; \mathbf{lookup}\ () \equiv \mathbf{update}\ s_0; \mathbf{return}\ s_0$$

[1] More precisely, single-sorted parameterised finitary signatures with equations between them.



$G_1 + G_2$
sum/parallel composition

$G_1 \otimes G_2$
tensor/serial composition

**Figure 1.** graph combination

The exception monad arises from the signature involving a single parameterised operation $\mathbf{throw} :: \mathbb{E} \to m\ \emptyset$ and no equations. We will later, in Section 2, explain this idea formally in more detail. Hyland et al. observe that applying the exception monad transformer to a monad arising from a presentation yields the monad obtained by adding a new operation $\mathbf{throw}$ with no additional equations to this presentation. They also observed that applying the $\mathbb{S}$ state monad transformer to a monad arising from a presentation yields the monad obtained by adding the operations $\mathbf{lookup}$ and $\mathbf{update}$ to that presentation, alongside their equations, as well as adding equations for commutativity. For exceptions, these equations manifest as:

$$\mathbf{lookup}\ () \equiv \mathbf{throw}\ e \qquad \mathbf{update}\ s_0 \equiv \mathbf{throw}\ e$$
$$\mathbf{throw}\ e \qquad \mathbf{throw}\ e$$

These observations allow us to reformulate the stack generation problem, assuming each monad transformer arises as the sum or the tensor of algebraic presentations. Consider a collection of presentations, and represent them as the vertices of a graph. Connect two presentations by an edge if all their operations commute with each other. The sum and tensor operations on presentations then have graph-theoretic counter-parts. The disjoint union, also known as *parallel composition*, of the two graphs corresponding to the sum of the two presentations. Similarly, by connecting each vertex of one graph with each vertex of the other graph, we obtain the graph corresponding to the tensor of two presentations. This construction is known as *serial composition* (see Figure 1). If the required combination of presentations can be expressed as a monad stack, or, more generally, as combination of sum and tensor, then the graph corresponding to this combination can be constructed using parallel and serial composition of graphs. Such graphs are known as *series-parallel (SP) graphs*, and they have been extensively studied since the 60's (Gallai 1967). In particular, there are linear-time algorithms for checking whether a given graph is an SP graph, and if so, find the sequence of graph operations that exhibits it as one. These observations, sans the connection with SP graphs and their algorithm, appear in passing at the concluding section of Hyland et al. (Hyland et al. 2006).

Relying on these observations, we obtain an efficient and systematic algorithm to find a stack of monad transformers that not only achieves the desired combination, but does so based solely on their commutative interaction, and the latter is quadratic in the number of effects. While tightly related to our motivating monad-stack problem, this problem is subtly different: there might not exist a monad stack that is based solely on the commutative interaction of the effects, but there might still exist a monad stack exhibiting all desired interactions.

This paper makes the following three contributions:

- It contains a more accessible account of Hyland et al.'s algebraic analysis of monad transformers.
- It synthesises Hyland et al.'s graph-theoretical connection with well-established results on SP-graphs to produce a linear-time algorithm for monad stack generation.
- It describes a web-tool implementing these results aimed for investigating the algebraic combination of presentations and for generating monad stacks.

This paper synthesises well-established results that are either folklore or inaccessible to the HASKELL community. It is aimed at readers who are familiar with the fundamentals of monads over the category of sets and functions, and undergraduate-level graph theory for computer scientists. When possible, we provide proof sketches to the various results, unless these extend beyond the scope of this paper, where we provide adequate citations. As our work is of limited technical novelty, of expository nature, and ultimately describes a program, our web-tool, we classified this paper as a functional pearl.

The remainder of the paper is organised as follows: Section 2 summarises Plotkin and Power's algebraic theory of effects. Section 3 summarises the relevant graph theoretical results. Section 4 presents a linear-time algorithm for monad stack generation, its web-tool implementation, and the nuances involved in our approach. Section 5 concludes. Appendix A provides further technical details about equational logic.

## 2. Monads and algebraic effects

We briefly summarise of the algebraic understanding of monads over sets and functions. We abuse HASKELL's syntax whenever possible for standard set-theoretic notions: the empty set ($\emptyset$), a distinguished singleton set ($() = \{()\}$), binary and arbitrary cartesian products ($(A, B)$, $\prod_{i\in I} A_i$) with pairs and tuples ($(a, b)$, $(a_i)_{i\in I}$), binary and arbitrary disjoint unions ($A + B$, $\sum_{i\in I} A_i$) with their injections ($(1, a)$, $(2, b)$, $(i, a)$), and function spaces ($A \to B$).

### 2.1 Algebraic operations

Let $(M, \mathbf{return}, \gg\!=)$ be a monad over the category of sets and functions. An *algebraic operation*[2] for $M$ is a function $\mathbf{op} :: P \to MA$. The set $P$ is called the *parameter type* and the set $A$ is called the *arity type* of $\mathbf{op}$. When $A$ is a finite set, we say that $\mathbf{op}$ is finitary.

**Example 1.** Let $\mathbb{S} = \{s_1, \ldots, s_n\}$ be a finite set of states. The global state monad $\mathbf{State}\ \mathbb{S}\ A \coloneqq \mathbb{S} \to (\mathbb{S}, A)$ has the following two finitary algebraic operations:

$$\mathbf{lookup} :: () \to \mathbf{State}\ \mathbb{S}\ A$$
$$\mathbf{lookup}\quad () = \lambda s \to (s, s)$$

$$\mathbf{update} :: \mathbb{S} \to \mathbf{State}\ \mathbb{S}\ ()$$
$$\mathbf{update}\quad s_0 = \lambda s \to (s_0, s)$$ □

**Example 2.** Let $\mathbb{E}$ be any set of exceptions. The exception monad $\mathbf{Error}\ \mathbb{E}\ A \coloneqq \mathbb{E} + A$ has the following finitary algebraic operation:

$$\mathbf{throw} :: \mathbb{E} \to \mathbf{Error}\ \mathbb{E}\ \emptyset$$
$$\mathbf{throw}\quad e = (1, e)$$ □

### 2.2 Equational logic

The well-established[3] fundamental observation underlying the algebraic theory of effects is that each monad is determined by its collection of algebraic operations and their relationships. To present this observation, we will need some syntax, and reuse HASKELL's syntax whenever possible.

[2] These are called *Kleisli arrows* for $M$. Plotkin and Power (Plotkin and Power 2003) also call these functions *generic operations* for $M$.

[3] See Linton (Linton 1966) following Lawvere's thesis (Lawvere 1963).

Value judgements $\Gamma \vdash V :: A$:

$$\Gamma \vdash x :: A, \text{when } \Gamma(x) = A \qquad \Gamma \vdash a :: A, \text{when } a \in A$$

$$\frac{\text{for all } i = 1, \ldots, n\text{: } \Gamma \vdash V_i :: A_i}{\Gamma \vdash (V_1, \ldots, V_n) :: A_1, \ldots, A_n}$$

$$\frac{\Gamma \vdash V :: A_{i_j}}{\Gamma \vdash I.i_j\ V :: I.i_1\ A_1 \mid \ \ldots \ \mid I.i_n\ A_n}$$

Computation judgements $\Gamma \vdash N :: m\ A$:

$$\frac{\Gamma \vdash V :: A}{\Gamma \vdash \mathbf{return}\ V :: m\ A}$$

$$\frac{\Gamma \vdash N_1 :: m\ A_1 \quad \Gamma, x :: A_1 \vdash N_2 :: m\ A_1}{\Gamma \vdash x \leftarrow N_1; N_2 :: m\ A_2}$$

$$\frac{\Gamma \vdash V :: I.i_1\ A_1 \mid \ \ldots \ \mid I.i_n\ A_n \quad \text{for all } i = 1, \ldots, n\text{: } \Gamma, x_i :: A_i \vdash N_i :: m\ A}{\Gamma \vdash \mathbf{case}\ V\ \mathbf{of}\ I.i_1\ x_1 \to N_1; \ldots I.\ i_n\ \ x_n \to N_n :: m\ A}$$

$$\frac{\Gamma \vdash V :: A_1, \ldots, A_n \quad \Gamma, x_1 :: A_1, \ldots, x_n :: A_n \vdash N :: m\ A}{\Gamma \vdash \mathbf{case}\ V\ \mathbf{of}\ (x_1, ..., x_n) \to N :: m\ A}$$

$$\frac{\Gamma \vdash V :: P}{\Gamma \vdash \mathbf{op}\ V :: m\ A}(\mathbf{op} :: P \to m\ A \in \sigma)$$

**Figure 3.** type system

Let $\mathbb{Com}\mathbb{s}$ be a countable set whose elements are constructor identifiers (e.g. *Nothing*, *Just*, etc.). If $I$ ranges over the finite subsets of $\mathbb{Com}\mathbb{s}$, we define the *ground types* as:

$$A ::= () \Big| \emptyset \Big| A_1, \ldots, A_n \Big| I.i_1\ A_1 \mid \ \ldots \ \mid I.i_n\ A_n$$

A *finitary signature* $\sigma$ consists of a pair $(\pi, \mathrm{ar})$, where $\pi$ is a set whose elements are called *parametric operation symbols*, and ar assigns to each operation symbol $\mathbf{op}$ in $\pi$ a pair of ground types $P$ and $A$. Instead of writing $\mathrm{ar}(\mathbf{op}) = (P, A)$, we write $\mathbf{op}$::$P \to m\ A$. In this notation, $m$ is just part of the syntax, and does not refer to any monad. Let $\mathbb{Var}$ be a countable set of identifiers (e.g., $x$, $y$, etc.). Given a signature $\sigma$ and a set $A$, Figure 2 defines the set of terms with values in some set $G$ thought of as a primitive type. This syntax is richer than in the equational logic literature, but closer to our computational interpretation of interest. The terms have a straightforward type system, given in Figure 3. As usual, contexts $\Gamma = x_1 :: A_1, \ldots, x_n :: A_n$ are partial functions with from $\mathbb{Var}$ to types a finite domain.

**Example 3.** Let $Bool \subseteq \mathbb{Var}$ be the set consisting of the two constructors *True* and *False*. We encode Boolean types as $Bool ::= Bool.True\ () \mid Bool.False\ ()$ (heavily abusing notation), with the truth values $Bool.True\ ()$ and $Bool.False\ ()$. We encode the conditional statement by:

$$\begin{array}{l}\mathbf{case}\ V\ \mathbf{of}\\ \quad Bool.True\ \_ \to N_1\\ \quad Bool.False\ \_ \to N_2\end{array}$$

More generally, if $I = \{C_1, \ldots, C_n\} \subseteq \mathbb{Var}$ is a set of constructors, we also use $I$ for the type $I ::= I.C_1\ () \mid \ \ldots \ \mid I.C_n\ ()$. □

This syntax allows us to express equations in context:

$$\frac{\Gamma \vdash N_1 :: m\ A \quad \Gamma \vdash N_2 :: m\ A}{\Gamma \vdash N_1 \equiv N_2 :: m\ A}$$

A *presentation* is a pair $(\sigma, \mathbf{Ax})$ consisting of a signature $\sigma$ and a set $\sigma$ of equations in context with no primitive values (i.e., $G = \emptyset$) considered as *axioms*. We derive further equations using a straightforward equational logic. The full details of this logic are given by Figure 7 in Appendix A.

**Example 4.** Let $\mathbb{S} = \{s_1, \ldots, s_n\} \subseteq I$ be a finite set of states. The signature for global $\mathbb{S}$-state is

$$\{\mathbf{lookup} :: () \to m\ \mathbb{S}, \mathbf{update} :: \mathbb{S} \to m\ ()\}$$

The global $\mathbb{S}$-state presentation then consists of the following three axioms over this signature::

$$\vdash \mathbf{return}\ () \equiv x \leftarrow \mathbf{lookup}\ (); \mathbf{update}\ x$$

$$x :: \mathbb{S} \vdash \mathbf{update}\ x;\ \mathbf{lookup}\ () \equiv \mathbf{update}\ x;\ \mathbf{return}\ x$$

$$x :: \mathbb{S}, y :: \mathbb{S} \vdash \mathbf{update}\ x;\ \mathbf{update}\ y \equiv \mathbf{update}\ y$$

From these three equations follows the following equation:

$$\begin{array}{l}\vdash x_1 \leftarrow \mathbf{lookup}\ (); \quad x \leftarrow \mathbf{lookup}\ ();\\ \quad x_2 \leftarrow \mathbf{lookup}\ (); \equiv \mathbf{return}\ (x, x)\\ \quad \mathbf{return}\ (x_1, x_2)\end{array}$$

using the following argument due to Melliès (Melliès 2010).

First, note that:

$$\begin{array}{l}x :: \mathbb{S} \vdash\\ \quad u \leftarrow \mathbf{update}\ x;\\ \quad y \leftarrow \mathbf{lookup}\ ();\\ \quad N\\ \qquad \equiv \ \{\text{-by sequencing -}\}\\ \quad y \leftarrow (u \leftarrow \mathbf{update}\ x; \mathbf{lookup}\ ());\\ \quad N\\ \qquad \equiv \ \{\text{-by axiom -}\}\\ \quad y \leftarrow (u \leftarrow \mathbf{update}\ x; \mathbf{return}\ x);\\ \quad N\\ \qquad \equiv \ \{\text{-by sequencing -}\}\\ \quad u \leftarrow \mathbf{update}\ x;\\ \quad y \leftarrow \mathbf{return}\ x\\ \quad N\\ \qquad \equiv \ \{\text{-by bind-}\beta\text{ -}\}\\ \quad u \leftarrow \mathbf{update}\ x;\\ \quad N\ [x\ /\ y]\end{array}$$

On the one hand, we have:

$$\begin{array}{l}\vdash x_1 \leftarrow \mathbf{lookup}\ ();\\ \quad x_2 \leftarrow \mathbf{lookup}\ ();\\ \quad \mathbf{return}\ (x_1, x_2)\\ \qquad \equiv \ \{\text{-by bind-}\beta\text{ -}\}\\ \quad u \leftarrow \mathbf{return}\ ();\\ \quad x_1 \leftarrow \mathbf{lookup}\ ();\\ \quad x_2 \leftarrow \mathbf{lookup}\ ();\\ \quad \mathbf{return}\ (x_1, x_2)\\ \qquad \equiv \ \{\text{-by axiom -}\}\\ \quad z \leftarrow \mathbf{lookup}\ ();\\ \quad u \leftarrow \mathbf{update}\ z;\\ \quad x_1 \leftarrow \mathbf{lookup}\ ();\\ \quad x_2 \leftarrow \mathbf{lookup}\ ();\\ \quad \mathbf{return}\ (x_1, x_2)\\ \qquad \equiv \ \{\text{-by the previous calculation -}\}\\ \quad z \leftarrow \mathbf{lookup}\ ();\\ \quad u \leftarrow \mathbf{update}\ z;\end{array}$$

$$
\begin{array}{llll}
\textit{Types} & A & ::= G \mid () \mid \emptyset \mid A_1, \ldots, A_n \mid I.i_1\ A_1 \mid \ \ldots\ \mid I.i_n\ A_n & (I = \{i_1, \ldots, i_n\} \subseteq \mathbb{C}\mathtt{ons}) \\
\textit{Values} & V & ::= a \mid x \mid (V_1, \ldots, V_n) \mid I.i\ V & (a \in G, x \in \mathbb{V}\mathtt{ar}, i \in I \subseteq \mathbb{C}\mathtt{ons}) \\
\textit{Computations} & N & ::= \mathbf{return}\ V \mid x \leftarrow N_1; N_2 & (x \in \mathbb{V}\mathtt{ar}) \\
 & & \mid \mathbf{case}\ V\ \mathbf{of}\ \ I.i_1\ x_1 \to N_1 & (I = \{i_1, \ldots, i_n\} \subseteq \mathbb{C}\mathtt{ons}) \\
 & & \qquad\qquad \ldots \\
 & & \qquad\qquad I.\ i_n\ \ x_n \to N_n & \\
 & & \mid \mathbf{case}\ V\ \mathbf{of}\ (x_1, \ldots, x_n) \to N & (x_1, \ldots, x_n \in \mathbb{V}\mathtt{ar}) \\
 & & \mid \mathbf{op}\ V & (\mathbf{op} \in \pi)
\end{array}
$$

**Figure 2.** terms with values in $G$

$x_2 \leftarrow \mathbf{lookup}\ ();$
$\mathbf{return}\ (z, x_2)$
$\quad \equiv$ {-by the previous calculation -}
$z \leftarrow \mathbf{lookup}\ ();$
$u \leftarrow \mathbf{update}\ z;$
$\mathbf{return}\ (z, z)$

On the other hand, a similar argument shows:

$$
\begin{array}{ll}
\vdash x \leftarrow \mathbf{lookup}\ (); & \equiv z \leftarrow \mathbf{lookup}\ (); \\
\quad \mathbf{return}\ (x, x) & \qquad u \leftarrow \mathbf{update}\ z; \\
 & \qquad \mathbf{return}\ (z, z)
\end{array}
$$

□

**Example 5.** Let $\mathbb{E} \subseteq I$ be any finite set of exceptions. The presentation for exceptions does not require any axioms. Using the elimination term for sum types, we can show that once we obtain a value of the empty type, every two terms are equal. Indeed, if $u$ is fresh to $N$, then:

$$z :: \emptyset \vdash N \equiv N\,[z\ /\ u] \equiv \mathbf{case}\ z\ \mathbf{of} \quad \text{-- Empty case split.}$$

Therefore throwing an exception halts any further computation:

$$
\begin{array}{ll}
e :: \mathbb{E} \vdash z \leftarrow \mathbf{throw}\ e; & \equiv z \leftarrow \mathbf{throw}\ e; \\
\quad N & \qquad N'
\end{array}
$$

□

### 2.3 Semantics

Let $\sigma$ be a signature. Our syntax reflects our intended semantics. Types $A$ denote sets using products and sums, and contexts $\Gamma$ denote the product $[\![\Gamma]\!] \coloneqq \prod_{(x::A)\in\Gamma} [\![A]\!]$, i.e., the set of environments of context $\Gamma$. Given a monad $(M, \mathbf{return}, \gg\!=)$ and, for every $\mathbf{op} :: P \to m\ A$ in $\sigma$, an algebraic operation $[\![\mathbf{op}]\!] :: [\![P]\!] \to M\,[\![A]\!]$, the term language has straightforward interpretation. The full details are given by Figure 6 in Appendix A. Value terms $\Gamma \vdash V : A$ denote, given a context $\gamma$, an element $[\![V]\!]\,(\gamma) \in [\![A]\!]$. Similarly, computation terms $\Gamma \vdash N :: m\ A$ denote, given an environment $\gamma$, an element $[\![N]\!]\,(\gamma) \in M\,[\![A]\!]$. We say that a given choice of monad and operations *satisfies* an equation $\Gamma \vdash N \equiv N' :: m\ A$, if, for all environments $\gamma$, $[\![N]\!]\,(\gamma) = [\![N']\!]\,(\gamma)$. A *model* for a presentation $(\sigma, \mathbf{Ax})$ is a monad and operations satisfying all the equations in $\mathbf{Ax}$.

**Example 6.** The global state monad from Example 1 and the exception monad from Example 2 are, respectively, models for the global state presentation of Example 4 and the exception presentation of Example 5. □

A straightforward inductive argument shows that all equations derived from a given presentation are satisfied by all its models, i.e. $N \equiv N' \implies [\![N]\!] = [\![N']\!]$. Thus our logic formalises the axiomatic approach to reasoning about monadic code advocated by Gibbons and Hinze (Gibbons and Hinze 2011). Moreover, as is well-known, such presentations can act as specifications for monads:

**Theorem 1.** *For every presentation $\mathcal{P}$ there exists a model $M_{\mathcal{P}}$ for $\mathcal{P}$ such that $N \equiv N'$ is derived from $\mathcal{P}$ if and only if $[\![N]\!] = [\![N']\!]$ in the model.*

The model $M_{\mathcal{P}}$ is constructed by defining $MA$ to be set of computation terms with primitive values in $G \coloneqq A$, quotiented by the $\equiv$ equivalence relation $\mathcal{P}$ induces on these terms. This construction is sketched in more detail in Appendix A.

Not every monad arises as $M_{\mathcal{P}}$ for some presentation $\mathcal{P}$, notably the continuation monad. However, it is still fruitful to view monads as arising from a suitable generalised notion of presentation. This view can be made precise using set-theoretical universes. Substantiating these two claims is beyond the scope of this paper.

Plotkin and Power's algebraic analysis of computational effects (Plotkin and Power 2002) identifies presentations $\mathcal{P}$ such that $M_{\mathcal{P}}$ is a computationally meaningful monad. We briefly summarise one of their examples.

**Example 7.** The global state monad from Example 1 is isomorphic to the monad specified by the presentation in Example 4. One way to prove this fact is to note that every term $\vdash N :: m\ A$ can be normalised by sequencing $x \leftarrow \mathbf{lookup}\ (); \mathbf{update}\ x$ before $N$ using the first axiom in the presentation, and then using the other two axioms to reduce the term to a term of the form:

$$
\begin{array}{l}
x \leftarrow \mathbf{lookup}\ (); \\
\mathbf{case}\ x\ \mathbf{of} \\
\quad s_1 \to \mathbf{update}\ s_{i_1} \\
\qquad\quad \mathbf{return}\ a_1 \\
\quad \vdots \\
\quad s_n \to \mathbf{update}\ s_{i_n} \\
\qquad\quad \mathbf{return}\ a_n
\end{array}
$$

These terms are in bijection with the functions $\mathbb{S} \to (\mathbb{S}, A)$, and they denote different functions in the global state monad model. Therefore they represent different equivalence classes of terms. □

The exception monad from Example 2 is similarly specified by the presentation in Example 5.

### 2.4 Monad transformers

Signatures compose readily. Let $\sigma_1 = (\pi_1, \mathrm{ar}_1)$ and $\sigma_2$ be two signatures. We define their *sum* as the signature $\sigma_1 + \sigma_2$ given by $(\pi_1 + \pi_2, [\mathrm{ar}_1, \mathrm{ar}_2])$, i.e., the set of operations is the disjoint union of the two sets of operations, and the arity of each $\mathbf{op}$ from $\pi_i$ in the combined signature is its arity in the appropriate component, $\mathrm{ar}_i(\mathbf{op}_i)$. This classic idea, with signatures expressed using functors, has been fruitful in HASKELL, e.g., Swierstra's solution to the expression problem (Swierstra 2008).

Consequently, presentations compose readily. Consider any two presentations $\mathcal{P}_1 = (\sigma_1, \mathbf{Ax}_1)$ and $\mathcal{P}_2 = (\sigma_2, \mathbf{Ax}_2)$. Their *sum* is the presentation $\mathcal{P}_1 + \mathcal{P}_2$ whose signature is $\sigma_1 + \sigma_2$, and whose axioms are the union of the axioms $\mathbf{Ax}_1$ and $\mathbf{Ax}_2$, suitably renamed for the combined signature. Hyland et al. (Hyland et al.

2006) observe that several common monad transformer arise as the sum of presentations:

**Example 8.** Let $\mathcal{P}_{\mathbb{E}}$ be the presentation for $\mathbb{E}$-exceptions from Example 5. Given any presentation $\mathcal{P}$, we have:

$$M_{\mathcal{P}_{\mathbb{E}}+\mathcal{P}} = \mathbf{ErrorT}\ \mathbb{E}\ (M_{\mathcal{P}_{Exc}})$$

Thus, the action of the exception monad transformer on the monad arising from $\mathcal{P}$ can be recovered by summing $\mathcal{P}$ with the exception presentation and generating the corresponding monad.

More generally, consider any finite signature:

$$\sigma = \{\mathbf{op}_1 :: P_1 \to m\ A_1, \ldots, \mathbf{op}_n :: P_n \to m\ A_n\}$$

Let $F$ be its corresponding functor, namely:

$$F_\sigma\ a = Op_1\ (P_1 \to A_1, a) \mid \cdots \mid Op_n\ (P_1 \to A_n, a)$$

Then $M_{(\sigma,\emptyset)}$ is the free monad (Swierstra 2008) $\mathbf{Free}\ F_\sigma$ and, for any other presentation $\mathcal{P}$,

$$M_{(\sigma,\emptyset)+\mathcal{P}} = \mathbf{FreeT}\ F_\sigma\ M_{\mathcal{P}}$$

I.e., by summing with a free presentation we recover the free monad transformer. □

Presentations can be composed in an additional way. Consider any two presentations $\mathcal{P}_1 = (\sigma_1, \mathbf{Ax}_1)$ and $\mathcal{P}_2 = (\sigma_2, \mathbf{Ax}_2)$. Consider the set of equations consisting ofthe equations, for every $\mathbf{op}_1 :: P_1 \to m\ A_1$ in $\sigma_1$ and $\mathbf{op}_2 :: P_2 \to m\ A_2$ in $\sigma_2$:

$$\begin{array}{l} p_1 :: P_1, p_2 :: P_2 \vdash a_1 \leftarrow \mathbf{op}_1\ p_1 \\ \qquad\qquad\quad a_2 \leftarrow \mathbf{op}_2\ p_2 \\ \qquad\qquad\quad \mathbf{return}\ (a_1, a_2) \\ \qquad\qquad :: m\ (A_1, A_2) \end{array} \quad \equiv \quad \begin{array}{l} a_2 \leftarrow \mathbf{op}_2\ p_2 \\ a_1 \leftarrow \mathbf{op}_1\ p_1 \\ \mathbf{return}\ (a_1, a_2) \end{array}$$

We call these equations the *commutativity* equations. The *tensor* of $\mathcal{P}_1$ and $\mathcal{P}_2$ is the presentation $\mathcal{P}_1 \otimes \mathcal{P}_2$ obtained by adding the commutativity equations to the sum $\mathcal{P}_1 + \mathcal{P}_2$. Hyland et al. (Hyland et al. 2006) observe that several common monad transformer arise as the tensor of presentations:

**Example 9.** Let $\mathcal{P}_{\mathrm{ST}(\mathbb{S})}$ be the presentation for global $\mathbb{S}$-state from Example 4. Given any presentation $\mathcal{P}$, we have:

$$M_{\mathcal{P}_{\mathrm{ST}(\mathbb{S})}\otimes\mathcal{P}} = \mathbf{StateT}\ \mathbb{S}\ M_{\mathcal{P}}$$

Similarly, let $\mathcal{P}_{\mathrm{Rd}(\mathbb{S})}$ be the presentation consisting of the signature $\mathbf{lookup} :: () \to m\ \mathbb{S}$ and the equations:

$$\begin{array}{l} \vdash x \leftarrow \mathbf{lookup}\ (); \equiv \mathbf{return}\ () \\ \quad \mathbf{return}\ () \end{array}$$

$$\begin{array}{l} \vdash x \leftarrow \mathbf{lookup}\ () \qquad z \leftarrow \mathbf{lookup}\ () \\ \quad y \leftarrow \mathbf{lookup}\ () \ \equiv\ \mathbf{return}\ (z, z) \\ \quad \mathbf{return}\ (x, y) \end{array}$$

Then $M_{\mathcal{P}_{\mathrm{Rd}(\mathbb{S})}}$ is the $\mathbb{S}$-reader monad $M_{\mathcal{P}_{\mathrm{Rd}(\mathbb{S})}} = \mathbb{S} \to a$. Moreover, for all presentations $\mathcal{P}$:

$$M_{\mathcal{P}_{\mathrm{Rd}(\mathbb{S})}\otimes\mathcal{P}} = \mathbf{ReaderT}\ \mathbb{S}\ M_{\mathcal{P}}$$

The writer monad transformer can be recovered similarly. □

In terms of the monads induced by the resulting presentations, the sum and tensor operations are both commutative, associative, and the empty presentation (i.e., without any operations and equations) is neutral with respect to these two operations.

Using this algebraic understanding of monad transformer, we reformulate the monad stack generation problem from the introduction. Define formal sum and tensor terms:

$$t ::= x \mid \sum_{i\in I} t_i \mid \bigotimes_{i\in I} t_i \qquad (2 \leqslant |I| < \aleph_0)$$

For reasons that will become apparent in the next section, we call these SP-terms. The input to our stack generation problem is a collection of effects, $x_1, \ldots, x_n$ with the information whether each pair of effects should commute or not. This data is captured precisely by an irreflexive undirected graph whose vertices are labelled $x_1, \ldots, x_n$, where the presence of an edge means the two end-point effects should commute. The output to our reformulated problem is: (1) an SP-term realising this interaction of effects, i.e., the commutativity equations produced by the SP-term are precisely the equations described by the graph; and (2) if given, in addition, a description of which effects have corresponding familiar monad transformers arising as either sum or tensor of presentations, decide whether this combination of presentations generates a monad stack comprising of these transformers and arising through sum and tensor. This reformulation is subtly different than the original monad stack problem, as we will discuss in Section 4.

## 3. SP-graphs and modular decomposition

The essence of our reformulated monad stack problem is graph theoretical, and its linear-time solution is well-known in graph theory. We give a brief summary of the relevant graph-theoretical notions and results. Where possible, we sketch the proofs of the relevant results.

### 3.1 Series-parallel graphs

Let $G = (V, E)$ be an irreflexive undirected graph whose vertices are $V$ and its set of edges is $E$. The *complement* of $G$, is the graph $G^{\complement} \coloneqq (V, \{\{u, v\} | u \neq v, \{u, v\} \notin E\})$. Given any subset of vertices $U \subseteq V$, the *induced full subgraph* is the graph $U|_G \coloneqq (U, \{\{u, v\} \subseteq U | \{u, v\} \in E\})$. Recall that a *graph isomorphism* $\alpha :: G \to G'$ is a bijection $\alpha :: V \to V'$, such that $\{u, v\} \in E \iff \{\alpha(u), \alpha(v)\} \in E'$. When the graph is labelled, we require isomorphisms to preserve the labels. When $G$ is isomorphic to a full subgraph of $G'$, we say that G *embeds into* $G'$, and write $G \hookrightarrow G'$.

Let $(G_i)_{i\in I}$ be a collection of graphs. Their *sum* is the graph $\sum_{i\in I} G_i$ given by the disjoint union of the vertex sets, and suitably renaming the edges. The sum of the collection is also known as their *parallel composition*. The *tensor* of the collection is the graph $\bigotimes_{i\in I} G_i$ given by adding to their sum all the edges between vertices of different components, i.e.:

$$\{\{u, v\} | i \neq j \in I, u \in V_i, v \in V_j\}$$

The tensor is also known as their *serial composition*.

Thus, the SP-terms defined in the previous section denote graphs, where $x$ denotes a graph $[\![x]\!]$ with a single vertex labelled $x$, and compound terms are given homomorphic denotations using sum and tensor. A graph that is isomorphic to $[\![t]\!]$ for some SP-term $t$ is a *series-parallel (SP) graph*.

### 3.2 Cographs

The SP-graphs can be given an alternative description: a *cograph* is a graph that can be constructed from single vertices using sums and graph complements.

**Theorem 2** (folklore)**.** *A graph is an SP-graph if and only if it is a cograph.*

**Proof sketch:** Because $\bigotimes_{i\in I} G_i = \left(\sum_{i\in I} G_i^{\complement}\right)^{\complement}$, the "if" direction follows by induction on SP-terms. For the converse, define a complementation operation on SP-terms by:

$$x^{\complement} \coloneqq x \qquad \sum_{i\in I} t_i^{\complement} \coloneqq \bigotimes_{i\in I} t_i^{\complement} \qquad \bigotimes_{i\in I} t_i^{\complement} \coloneqq \sum_{i\in I} t_i^{\complement}$$

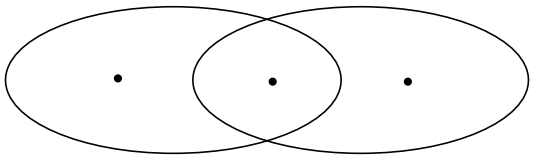


**Figure 4.** overlapping sets

Induction shows that, for all SP-terms $t$, $\left[\!\left[t^{\complement}\right]\!\right] = [\![t]\!]^{\complement}$. By induction on the formation of cographs, every cograph is an SP-graph. ■

The importance of identifying these two classes of graphs lies in the following *forbidden subgraph* characterisation of the cographs:

**Theorem 3** ((Corneil et al. 1981), Theorem 2)**.** *A graph is an SP-graph/cograph if and only if the following graph $P_4$ does not embed into it:*

$$\bullet - \bullet - \bullet - \bullet$$

This theorem gives a polynomial brute-force decision procedure determining whether a given graph is SP: for all distinct quadruples of vertices $v_1$, $v_2$, $v_3$, and $v_4$, verify that the following condition does not hold:

$$\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\} \in E \wedge$$
$$\{v_1, v_3\}, \{v_2, v_4\}, \{v_1, v_4\} \notin E$$

Moreover, if the procedure rejects the graph as SP, it can supply easy-to-check evidence. However, in case the graph is an SP-graph, we would like to know which SP-term denotes it.

### 3.3 Graph modules

Let $G = (V, E)$ be an irreflexive undirected graph. A *module* of $G$ is a set of vertices $U \subseteq V$ such that, for every $v \in V \setminus U$, either for all $u \in U$, $\{u, v\} \in E$, or for all all $u \in U$, $\{u, v\} \notin E$. Thus, a module is a set of vertices that cannot be distinguished by adjacency to any vertex outside the set.

**Example 10.** For every collection of graphs $(G_i)_{i \in I}$, the embedding of each component's vertices $V_i$, for every $i \in I$, is a module of both $\sum_{i \in I} G_i$ and of $\bigotimes_{i \in I} G_i$. □

We say that two sets $A$ and $B$ *overlap* if $A \setminus B$, $A \cap B$, and $B \setminus A$ are non-empty (see Figure 4). A module is *strong* if it does not overlap any other module.

**Example 11.** The empty set, the set of all vertices, and every vertex singleton are strong modules. The set $\{x, y\}$ is not a strong module of the SP-graph $[\![x + y + z]\!]$, as it overlaps the module $\{y, z\}$. The set $\{x, y\}$ is a strong module of $[\![(x + y) \otimes (z + w)]\!]$: any module that contains $y$ and $z$ must also contain $x$. □

Two strong modules are either contained in one another, or are disjoint. A strong module that is not the entire graph is *maximal* if the only strong module containing it is the entire graph. Thus maximal modules either coincide or are disjoint. As every vertex is contained in at least one strong module, the maximal strong modules of any graph with two or more vertices form a partition. The following result, due to Gallai (Gallai 1967), shows that this partition is particularly easy to compute for disconnected graphs:

**Proposition 4.** *Let $G$ be a graph. If $G$ is disconnected, its connected components form a partition of $G$ into maximal strong modules.*

**Proof sketch:** Consider a connected component $C$. Then $C$ is a module: every vertex not in $C$ is not adjacent to $C$'s vertices. Consider any other module $M$. If there are vertices in both $C \cap M$ and $C \setminus M$, then there must be an edge $\{u, v\}$ with $u \in C \cap M$ and $v \in C \setminus M$. As $M$ is a module, $v$ is adjacent to every $M$-vertex. As $C$ is a connected component, $M \subseteq C$, hence $C$ is strong. To show maximality, we need to rule out the case where $C'$ and $C''$ are two other connected components, and $M$ is a strong module such that $C, C' \subseteq M$ and $C'' \cap M = \emptyset$. This case is ruled out by the strength of $M$, as then $C \cup C''$ would be a module overlapping $M$. ■

As a maximal strong module of $G$ is also a maximal strong module of $G^{\complement}$, this proposition allows us to decompose both disconnected graphs and graphs whose complement is disconnected into strong modules.

Using this result, Gallai defines the *modular decomposition* of a graph: if the graph consists of a single vertex, this vertex forms the decomposition, otherwise, the decomposition consists of the partition of the graph into maximal strong modules, and progresses recursively for each such maximal strong module. Each non-leaf module in the decomposition is labelled: if the current subgraph is connected and so is its complement, the node is labelled *prime*. If the subgraph is disconnected, it is labelled *parallel*. If the complement subgraph is disconnected, it is label *series*. These are the only three cases, as no graph is both disconnected and has a disconnected complement. As these labels suggest, we can use the modular decomposition to characterise SP-graphs:

**Theorem 5** ((Möhring 1985))**.** *A graph is SP iff its modular decomposition has no prime nodes. In this case, the modular decomposition represents an SP-term denoting the graph.*

**Proof sketch:** The "only if" direction and the suffix follow by induction over the modular decomposition. To establish the "if" direction, we use term rewriting techniques. The rewriting system induced by the following rules is both terminating and confluent:

$$\sum_{i \in I} t_i \xrightarrow{\exists i_0 \in I : t_{i_0} = \sum_{j \in J_{i_0}} t_{(i_0,j)}} \sum_{i \in (I \setminus \{i_0\}) \cup \{(i_0,j) \mid j \in J_{i_0}\}} t_i$$

$$\bigotimes_{i \in I} t_i \xrightarrow{\exists i_0 \in I : t_{i_0} = \bigotimes_{j \in J_{i_0}} t_{(i_0,j)}} \bigotimes_{i \in (I \setminus \{i_0\}) \cup \{(i_0,j) \mid j \in J_{i_0}\}} t_i$$

Therefore, each SP-term has a unique normal form. Given such a normal form, induction over its subterms shows the connected components of the graph denoted by every subterm $\sum_{i \in I} t_i$ are $\{[\![t_i]\!] | i \in I\}$, and similarly the connected components of the complement of the graph denoted by every subterm $\bigotimes_{i \in I} t_i$ are $\{[\![t_i]\!] | i \in I\}$. Therefore, the modular decomposition of every term in normal form is identical to the normal form, hence has no prime nodes. ■

McConnell and Spinrad (McConnell and Spinrad 1999) give a linear-time algorithm for constructing the modular decomposition of a given graph. This algorithm is beyond the scope of this paper. Fortunately, for our purposes, we can simplify matters using the following two observations. First, once we identify a prime node in the decomposition, we have no need to find the decomposition, and it is sufficient to find the embedding of the forbidden graph $P_4$. Second, the graphs we intend to deal with are small in size, i.e., a few hundred vertices at most, and so even a polynomial super-linear time algorithm is acceptable. Algorithm 1 is structured around the definition of the modular decomposition.

## 4. Generating monad stacks

We now solved the first part of our reformulation of the monad stack generation problem: given a graphical description of our effects' commutativity, any correct SP decomposition algorithm, such as Algorithm 1, would decide whether this combination can be captured through sums and tensors of the given presentations,

**Algorithm 1:** super-linear SP decomposition

**Data**: a non-empty graph $G = (V, E)$
**Result**: an SP-term denoting the graph or a $P_4$-embedding
**if** $V = \{x\}$ **then** return $x$ ;
Find the connected components $(C_i)_{i\in I}$ of $G$, and $(C'_i)_{i\in I'}$ of $G^{\complement}$.
**if** $|I| > 1$ **then**
  **foreach** $i \in I$ **do** recurse on subgraph $G|_{C_i}$ yielding $t_i$;
  **if** *any call returned a $P_4$ embedding* **then** return it too;
  return $\sum_{i\in I} t_i$
**else if** $|I'| > 1$ **then**
  **foreach** $i \in I'$ **do** recurse on subgraph $G|_{C'_i}$ yielding $t_i$;
  **if** *any call returned a $P_4$ embedding* **then** return it too;
  return $\bigotimes_{i\in I'} t_i$
**else**
  Find an embedding of $P_4$ by brute-force

and provide the appropriate SP-term. We now address the second part of the problem, namely monad stack generation.

### 4.1 The algorithm

An *algebraic monad transformer* is a unary function on SP-terms of the form $f(y) = x + y$ or $f(y) = x \otimes y$. We say that the transformer is of *sum*, or *tensor*, *type* respectively, and that it is *determined* by $x$. If every vertex in a graph is associated with a single transformer, we identify transformers with vertices. An *unbound monad stack* is a composition of monad transformers. A *bound monad stack* is an unbound monad stack together with a single SP-variable $v$. A *monad stack* is either an unbound or a bound monad stack. The graph denoted by a bound monad stack $(s(y), v)$ is $[\![s(v)]\!]$. The graph denoted by an unbound monad stack $s(y)$ is $[\![s(\mathrm{id})]\!]$, where id denotes the empty graph. The vast majority of computationally meaningful presentation that gives rise to familiar monad transformers does so using only one of sum and tensor. The notable example is the free presentation with a unary parameterised operation, which combines as a free monad transformer when summed, and as a writer monad when tensored. Consequently, we assume that the vertices of our input graphs are labelled by exactly one of sum, tensor, or none according to the type of familiar monad transformers they determine. We do not foresee any difficulty in handling the more general case.

Our algorithm relies on the following three observations:

**Theorem 6.** *Let $G = (V, E)$ be a graph and $V_+ \cup V_\otimes$ a partition of its vertices. Assume $G$ is denoted by an unbounded monad stack whose transformers of sum type are precisely $V_+$ and whose transformers of tensor type are precisely $V_\otimes$.*

*There exists a partition $C_1, \ldots, C_n$ of $V$ satisfying:*

1. *Each class $C_i$ consists of transformers of the same type, and the sequence $C_1, \ldots, C_n$ alternates in type.*
2. *Every monad stack denoting $G$ can be described by an unbounded monad stack $S_1 \circ \cdots \circ S_n$ where, for all $i = 1, \ldots, n$, the transformers in $S_i$ are a permutation of the transformers in $C_i$.*
3. *The depth of the SP decomposition of $G$ is $n$, and the $i^{\text{th}}$ level of this decomposition is $C_i \cup \{[\![S_{i+1} \circ \cdots \circ S_n(\mathrm{id})]\!]\}$.*

**Proof sketch:** Every bounded monad stack $(S(y), v)$ determining $G$ can be viewed as an unbounded stack $S \circ v$. If $S(y) = v_1 \circ \cdots \circ v_{n_0}(y)$ is an unbounded stack denoting $G$, by defining $C_i^0 := \{v_i\}$ for all $i = 1, \ldots, n_0$ we obtain a partition of $V$ to transformer classes of homogeneous type possessing the invariant that $S(y)$ can be decomposed as in condition 2. Recall the rewrite system over SP-terms from the proof of Theorem 5. The partition can thus be regarded as an SP-term satisfying condition 3, and hence the rewrite rules can be applied to it. Each rewrite merges two adjacent classes $C_i^k, C_{i+1}^k$ with the same type, and maintains the invariants in 2 and 3. A partition is then in normal form if and only if it alternates in type, and thus the normal form, i.e., the SP decomposition, possess properties 1-3. The fact that every monad stack denoting $G$ can be decomposed as in 2 follows, as the rewriting process always yields the SP decomposition of $G$. ■

If a graph $G$ can only be denoted by bounded stacks, there must be exactly one vertex that does not determine any familiar monad transformer. In this case, a similar proof proves the following theorem:

**Theorem 7.** *Let $G = (V, E)$ be a graph and $V_+ \cup V_\otimes$ a partition of $V \setminus \{v\}$ for some $v \in V$. Assume $G$ is denoted by an bounded monad stack whose transformers of sum type are precisely $V_+$ and whose transformers of tensor type are precisely $V_\otimes$.*

*There exists a partition $C_1, \ldots, C_n$ of $V \setminus \{v\}$ satisfying:*

1. *Each class $C_i$ consists of transformers of the same type, and the sequence $C_1, \ldots, C_n$ alternates in type.*
2. *Every monad stack denoting $G$ is bounded monad stack $S_1 \circ \cdots \circ S_n(v)$ where, for all $i = 1, \ldots, n$, the transformers in $S_i$ are a permutation of the transformers in $C_i$.*
3. *The depth of the SP decomposition of $G$ is $n$, and the $i^{\text{th}}$ level of this decomposition is $C_i \cup \{[\![S_{i+1} \circ \cdots \circ S_n(v)]\!]\}$.*

In particular, we obtain the following condition, characterising when our reformulated monad generation problem has a solution:

**Corollary 8.** *A graph $G$ is denoted by a monad stack of transformers of sum type determined by $V_+$ and of transformers of tensor type determined by $V_\otimes$, with $V_+ \cap V_\otimes = \emptyset$, if and only if $G$ is an SP-graph and every level in the SP decomposition has at most one node that is non-leaf or labelled differently than the current node.*

Algorithm 2 solves the stack generation problem. It checks the condition in Corollary 8 level by level, reconstructing the classes of the appropriate decomposition from Theorems 6–7 for each level in the SP decomposition. If we assume a linear SP decomposition algorithm, linear time iteration over the SP decomposition (e.g., by a breadth-first traversal), and constant time access to vertex labels, Algorithm 2 is linear.

**Algorithm 2:** linear stack generation

**Data**: a non-empty graph $G = (V, E)$ whose vertices are labelled by 'sum', 'tensor' or 'none'.
**Result**: a description of all stacks denoting the graph with transformers determined by the labels
**if** *there is more than one vertex labelled 'none'* **then** fail;
**if** *$G$ is not an SP-graph* **then** fail;
**foreach** *level in the SP decomposition of $G$* **do**
  Partition the level into leaf nodes $C_i$ labelled with the type of the current level, and the remaining nodes $C'$;
  **if** $|C'| > 1$ **then** fail;
  **else if** $C' = \{v\}$ *and $v$ is labelled 'none'* **then**
    return the bounded stacks $((C_1, \ldots, C_i), v)$
  **else if** $C' = \{v\}$ **then**
    return the unbounded stacks $(C_1, \ldots, C_i, \{v\})$
  **else if** $C' = \emptyset$ **then**
    return the unbounded stacks $(C_1, \ldots, C_i)$
  {-continue to the next iteration-}

### 4.2 Semantic web-tool

We implemented our algorithms in a minimal web-tool[4]. The tool takes as input a textual description of an effect interaction graph and labels for its vertices. It computes the graphs SP decomposition using Algorithm 1, providing either a $P_4$ counter-example when the graph is not SP, or displaying an SP term denoting the graph. In the latter case, the tool uses Algorithm 2 to generate all monad stacks that denote the input graph based on its labels.

The textual input to the tool comprises of two sections. The first section describes the vertices of the input graph and their labels as determining familiar monad transformer as either sum or tensor. As a convenience, we allow multiple vertices to share the same label. The second section describes the edges of the input graph. Lines of the form $u_1, \ldots, u_m \otimes v_1, \ldots, v_n$ add all possible edges between *distinct* vertices on either side of the tensor. Lines of the form $v_1, \ldots, v_n$ `all commute` add all possible edges between distinct vertices in the list.

The tool was straightforward to implement once we finalised both algorithms. We used OCAML as it has a flexible graph library, OCAMLGRAPH (Conchon et al. 2007), that is fully compatible with a relatively reliable JAVASCRIPT™ backened, js_of_ocaml[5]. However, there is nothing inherent to our implementation requiring OCAML, and we foresee no particular difficulties in implementing it in other languages.

### 4.3 Nuances

The main subtlety with our reformulation of the stack generation problem is that it is more restrictive than the original stack generation problem.

**Example 12.** Consider the following combination of presentations:

$$raise \otimes ND, getChar$$
$$putChar \otimes getChar$$

Computationally, this combination is a candidate for a monad with the following effects. We can read and write (separately) to two files non-interactively. We can raise exceptions. And we can make non-deterministic choices. As raising an exception would terminate the program and we cannot observe how far into the input file we have read, exception raising commutes with both non-deterministic choice and the input. As we read and write from separate files, input should commute with output. Traditionally, input and output do not commute with non-determinism (Hyland et al. 2007), though the traditional analysis focusses on interactive I/O rather than file I/O. Finally, we do not want exceptions to commute with output, as after raising the exception we may still inspect the file for output traces.

The resulting graph is $P_4$:

$$ND - raise - getChar - putChar$$

Thus, this potentially computationally meaningful graph is not an SP-graph, hence our decision procedure would reject it. Consider the case where $ND$ denotes the presentation for non-determinism, namely having the signature $\{\mathbf{choose} :: () \to \mathit{Bool}\}$ and the following three equations:

$$\begin{array}{lcl} x \leftarrow \mathbf{choose}\ () & & y \leftarrow \mathbf{choose} \\ y \leftarrow \mathbf{choose}\ () & \equiv & x \leftarrow \mathbf{choose} \\ \mathbf{return}\ (x, y) & & \mathbf{return}\ (x, y) \end{array}$$

[4] Tool availble at:
`http://www.cl.cam.ac.uk/~ok259/graphtool/`
Code available at:
`https://code.launchpad.net/~ohad-kammar/graphtool/trunk`

[5] See `http://ocsigen.org/js_of_ocaml/`.

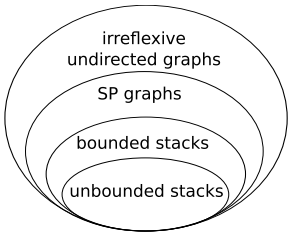



**Figure 5.** graph classes hierarchy

$$\begin{array}{l} x \leftarrow \mathbf{choose}\ () \\ y \leftarrow \mathbf{choose}\ () \\ \mathbf{case}\ (x, y)\ \mathbf{of} \\ \quad (\mathit{True}, \mathit{True}) \to \mathbf{return}\ 1 \\ \quad (\mathit{True}, \mathit{False}) \to \mathbf{return}\ 2 \\ \quad (\_, \quad \mathit{False}) \to \mathbf{return}\ 3 \end{array} \equiv \begin{array}{l} x \leftarrow \mathbf{choose}\ () \\ y \leftarrow \mathbf{choose}\ () \\ \mathbf{case}\ (x, y)\ \mathbf{of} \\ \quad (\mathit{True}, \ \_) \to \mathbf{return}\ 1 \\ \quad (\mathit{False}, \mathit{True}) \to \mathbf{return}\ 2 \\ \quad (\mathit{False}, \mathit{False}) \to \mathbf{return}\ 3 \end{array}$$

$$x \leftarrow \mathbf{choose}\ () \quad \equiv \quad \mathbf{return}\ ()$$
$$\mathbf{return}\ ()$$

In each equivalence, the set of available result types does not change, hence they would have the same non-deterministic behaviour.

From the last equation we can derive the commutativity equation for **choose** and *raise* even in their sum. Therefore, the following combination:

$$raise \otimes getChar$$
$$putChar \otimes getChar$$

Will *also* exhibit the behaviour $raise \otimes ND$. However, this interaction does arise from an SP-term, namely:

$$((raise + putChar) \otimes getChar) + ND \qquad \square$$

The discrepancy in Example 12 occurs because our analysis does not in fact take into account any intrinsic properties of the presentations at hand, but merely their superficial specified commutativity behaviour. Therefore the reformulate problem is subtly different than the original monad stack generation problem.

Example 12 also serves another purpose. Our analysis, extending Hyland et al.'s (Hyland et al. 2006), deals with four sub-classes of graphs (see Figure 5). These are proper inclusions, with potentially computationally meaningful examples separating them. Any stack of monad transformers arising from sum and tensor yields an an unbounded stack. By applying such a stack to any presentation that does not have a corresponding familiar monad transformer, for example, the list monad, we separate the bounded stacks from the unbounded stacks. The SP-graph description in Example 12 separates the bounded stacks from the SP-graphs, as $ND$ does not correspond to a familiar monad transformer. Finally, the $P_4$ graph in Example 12 separates the SP-graphs from the rest of the graphs.

## 5. Conclusion

We showed how, based on the algebraic understanding of effects, monads, and monad transformers, commutativity analysis can efficiently generate monad stacks. Though subtly different, our solution provides a basic tool for semantic investigation of combinations of effects. We hope it will be useful for structuring effectful code in the future. After reformulating the stack generation problem in terms of commutativity equations, we could reuse much of the technology developed for algebraic effects and series-parallel graphs. While not widely familiar, the connection with SP-graphs

VALUE DENOTATIONS:

$$[\![a]\!](\gamma) \coloneqq a \qquad [\![x]\!](\gamma) \coloneqq \pi_x(\gamma)$$

$$[\![(V_1, \ldots, V_n)]\!](\gamma) \coloneqq ([\![V_1]\!](\gamma), \ldots, [\![V_n]\!](\gamma))$$

$$[\![I.i\ V]\!](\gamma) \coloneqq (I, i, [\![V]\!](\gamma))$$

COMPUTATION DENOTATIONS:

$$[\![\mathbf{return}\ V]\!](\gamma) \coloneqq \mathbf{return}([\![V]\!](\gamma))$$

$$[\![x \leftarrow N_1; N_2]\!](\gamma) \coloneqq [\![N_1]\!](\gamma) \gg\!= \lambda x \to [\![N_2]\!](\gamma, x)$$

$$[\![\mathbf{case}\ V\ \mathbf{of}\ I.i_1\ x_1 \to N_1; \ldots; I.\ i_n\ \ x_n \to N_n]\!](\gamma) \coloneqq$$

$$\begin{cases} [\![N_1]\!]((\gamma, x_1) & [\![V]\!](\gamma) = (I, i_1, x_1) \\ \quad\vdots & \\ [\![N_n]\!]((\gamma, x_n) & [\![V]\!](\gamma) = (I, i_n, x_n) \end{cases}$$

$$[\![\mathbf{case}\ V\ \mathbf{of}\ (x_1, \ldots, x_n) \to N]\!](\gamma) \coloneqq [\![N]\!](\gamma, x_1, \ldots, x_n)$$

where $[\![V]\!](\gamma) = (x_1, \ldots, x_n)$

$$[\![\mathbf{op}\ V]\!](\gamma) \coloneqq [\![\mathbf{op}]\!]([\![V]\!](\gamma))$$

**Figure 6.** semantics

previously appeared in the functional programming community (cf. Atkey's thesis (Atkey 2006)).

## A. Equational logic

We supply a more detailed technical account of our equational logic, starting with its semantics. Let $\sigma$ be a signature. Types denote sums and products:

$$[\![()]\!] \coloneqq \{()\} \qquad [\![\emptyset]\!] \coloneqq \emptyset \qquad [\![G]\!] \coloneqq G$$
$$[\![A_1, \ldots, A_n]\!] \coloneqq [\![A_1]\!], \ldots, [\![A_n]\!]$$
$$[\![I.i_1\ A_1 \mid \ \ldots \ \mid I.i_n\ A_n]\!] \coloneqq \sum_{i \in I} A_i$$
$$[\![\Gamma]\!] \coloneqq \prod_{(x::A) \in \Gamma} [\![A]\!]$$

Given a monad $(M, \mathbf{return}, \gg\!=)$ and, for every $\mathbf{op} :: P \to m\ A$ in $\sigma$, an algebraic operation $[\![\mathbf{op}]\!] :: [\![P]\!] \to M\ [\![A]\!]$, the denotational semantics for the term language are given by Figure 6. Value terms $\Gamma \vdash V : A$ denote, given a context $\gamma$, an element $[\![V]\!](\gamma) \in [\![A]\!]$. Similarly, computation terms $\Gamma \vdash N :: m\ A$ denote, given a context $\gamma$, an element $[\![N]\!](\gamma) \in M\ [\![A]\!]$. Crucially, if $A$ and $A'$ are ground types, for every function $f :: [\![A]\!] \to [\![A']\!]$ there exists a term $a :: A \vdash N :: m\ A'$ that denotes $f$ using the identity monad. This denotational semantics satisfies all the equational rules in Figure 7. Consequently, if some given monad and operations satisfy all the axioms of a presentation, they satisfy all equations derived in this equational logic.

Let $\mathcal{P}$ be any presentation. We outline how to construct the monad $M_{\mathcal{P}}$ from Theorem 1. Consider any set $A$. The relation $\equiv$ defined over terms with ground values in $G \coloneqq A$ is an equivalence relation, hence we can quotient the set of ground terms $\vdash N :: m\ A$ by $\equiv$ . This construction extends to a functor. If $f{::}A \to B$ is any set-theoretic function, define *fmap f* by mapping each primitive value $a$ to $f(a)$, and homomorphically preserving the remaining syntactic

REFLEXIVITY: $N \equiv N$

SYMMETRY: $\dfrac{N_1 \equiv N_2}{N_2 \equiv N_1}$

TRANSITIVITY: $\dfrac{N_1 \equiv N_2 \quad N_2 \equiv N_3}{N_1 \equiv N_3}$

BIND-$\beta$: $x \leftarrow \mathbf{return}\ V;\ N \equiv N[V/x]$

BIND-$\eta$: $N \equiv x \leftarrow N;\ \mathbf{return}\ x$

SEQUENCING: $x \leftarrow (y \leftarrow N; N_1);\ N_2 \equiv y \leftarrow N;\ x \leftarrow N_1;\ N_2$

BIND-CONGRUENCE: $\dfrac{N_1 \equiv N_1' \quad N_2 \equiv N_2'}{x \leftarrow N_1; N_2 \equiv x \leftarrow N_1'; N_2'}$

AXIOM: $N \equiv N' \in \mathbf{Ax}$

PRODUCT-$\beta$: $\mathbf{case}\ (V_1, \ldots, V_n)\ \mathbf{of}\ (x_1, \ldots, x_n) \to N \equiv N[V_1/x_1, \ldots, V_n/x_n]$

PRODUCT-$\eta$: $N[V/z] \equiv \mathbf{case}\ V\ \mathbf{of}\ (x_1, \ldots, x_n) \to N[(x_1, \ldots, x_n)/z]$

PRODUCT-CONGRUENCE: $\dfrac{N \equiv N'}{\mathbf{case}\ V\ \mathbf{of}\ (x_1, \ldots, x_n) \to N \equiv \mathbf{case}\ V\ \mathbf{of}\ (x_1, \ldots, x_n) \to N'}$

SUM-$\beta$:

$$\mathbf{case}\ V\ \mathbf{of}\ \{\ I.i_1\ x_1 \to N_1;\ \ldots;\ I.\ i_n\ \ x_n \to N_n\ \} \equiv N_j[V/x_j]$$

SUM-$\eta$

$$N[V/z] \equiv \mathbf{case}\ V\ \mathbf{of}\ \{\ I.i_1\ x_1 \to N[(I.i_1\ x_1)/z];\ \ldots;\ I.\ i_n\ \ x_n \to N[(I.i_n\ x_n)/z]\ \}$$

SUM-CONGRUENCE:

$$\frac{\text{for all } i = 1, \ldots, n\text{: } \Gamma \vdash N_i \equiv N_i'}{\mathbf{case}\ V\ \mathbf{of}\ \{\ I.i_1\ x_1 \to N_1;\ \ldots;\ I.\ i_n\ \ x_n \to N_n\ \} \equiv \mathbf{case}\ V\ \mathbf{of}\ \{\ I.i_1\ x_1 \to N_1';\ \ldots;\ I.\ i_n\ \ x_n \to N_n'\ \}}$$

**Figure 7.** equational logic

constructs. As our logic and axioms never inspect the primitive types, this definition respects the equational theory, hence is a well-defined function *fmap* $f :: M_{\mathcal{P}}\ A \to M_{\mathcal{P}}\ B$. The monadic $\mathbf{return}$ function maps every $a \in A$ to the equivalence class of $\mathbf{return}\ a$.

Defining the monadic bind requires some care. Consider any representative $\vdash N :: m\ A$ of an arbitrary equivalence class in $M_{\mathcal{P}}\ A$, and any function $f :: A \to M_{\mathcal{P}}\ B$. The term $N$ involves finitely many primitive values, say $a_1, \ldots, a_n$. By defining $I$ to be $\coloneqq \{C_1, \ldots, C_n\} \subseteq \mathbb{C}\mathtt{oms}$, we have a bijection $\beta :: \{a_1, \ldots, a_n\} \to I$, and therefore the equivalence class given by *fmap* $\beta([N]_{\equiv}) \in M_{\mathcal{P}}\ I$ has a representative $N' :: m\ I$. Define the term $c :: I \vdash N_f :: m\ B$ as follows:

$$\begin{array}{l}\mathbf{case}\ c\ \mathbf{of}\\ \quad C_1 \to N_1\\ \quad \vdots\\ \quad C_n \to N_n\end{array}$$

where, for all $i = 1, \ldots, n$, $N_i$ is a representative of the equivalence class $f(a_i)$. Let $x$ be a variable not appearing in any of $N_1, \ldots, N_n$. We then define $[N]_{\equiv} \gg\!= f$ as the equivalence class of $x \leftarrow N'; N_f$. The equational logic in Figure 7 ensures the monad laws hold for these definitions.

Finally, for every $\mathbf{op} :: P \to m\ A$ in $\sigma$, and $p \in [\![P]\!]$, there is an appropriate value term $\vdash V_p :: P$ such that $[\![V_p]\!]\,() = p$. Define $[\![\mathbf{op}]\!]\,(p)$ to be the equivalence class of $[\![\mathbf{op}\ V_p]\!]\,()$. This definition yields an algebraic operation $[\![\mathbf{op}]\!]$ for $M_{\mathcal{P}}$.

We can then show that $M_{\mathcal{P}}$ with these operations satisfies all the axioms in $\mathcal{P}$, and that this model satisfies the condition in Theorem 1.

## Acknowledgments

*(Omitted for reviewing purposes.)*

## References

R. Atkey. *Substructural Simple Type Theories for Separation and In-place Update*. PhD thesis, University of Edinburgh, 2006.

S. Conchon, J.-C. Filliâtre, and J. Signoles. Designing a generic graph library using ML functors. In *Trends in Functional Programming (TFP'07)*, New York City, USA, Apr. 2007. URL `http://www.lri.fr/~{}filliatr/ftp/publis/ocamlgraph-tfp-8.pdf`.

D. Corneil, H. Lerchs, and L. Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 3(3):163 – 174, 1981. ISSN 0166-218X. . URL `http://www.sciencedirect.com/science/article/pii/0166218X81900135`.

T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 18(1-2):25–66, 1967. ISSN 0001-5954. . URL `http://dx.doi.org/10.1007/BF02020961`.

J. Gibbons and R. Hinze. Just do it: Simple monadic equational reasoning. *SIGPLAN Not.*, 46(9):2–14, Sept. 2011. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2034574.2034777`.

M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006.

M. Hyland, P. B. Levy, G. Plotkin, and J. Power. Combining algebraic effects with continuations. *Theoretical Computer Science*, 375(13):20 – 40, 2007. ISSN 0304-3975. . URL `http://www.sciencedirect.com/science/article/pii/S0304397506009157`. Festschrift for John C. Reynoldss 70th birthday.

B. Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America*, 50(1): 869–872, 1963.

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. . URL `http://doi.acm.org/10.1145/199448.199528`.

F. Linton. Some aspects of equational categories. In S. Eilenberg, D. Harrison, S. MacLane, and H. Rhrl, editors, *Proceedings of the Conference on Categorical Algebra*, pages 84–94. Springer Berlin Heidelberg, 1966. ISBN 978-3-642-99904-8. . URL `http://dx.doi.org/10.1007/978-3-642-99902-4_3`.

R. M. McConnell and J. P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(13):189 – 241, 1999. ISSN 0012-365X. . URL `http://www.sciencedirect.com/science/article/pii/S0012365X98003197`.

P.-A. Melliès. Segal condition meets computational effects. In *Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science*, LICS '10, pages 150–159, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4114-3. . URL `http://dx.doi.org/10.1109/LICS.2010.46`.

R. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In I. Rival, editor, *Graphs and Order*, volume 147 of *NATO ASI Series*, pages 41–101. Springer Netherlands, 1985. ISBN 978-94-010-8848-0. . URL `http://dx.doi.org/10.1007/978-94-009-5315-4_2`.

G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43366-8. . URL `http://dx.doi.org/10.1007/3-540-45931-6_24`.

G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11:2003, 2003.

W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008. ISSN 0956-7968. . URL `http://dx.doi.org/10.1017/S0956796808006758`.

P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5. URL `http://dl.acm.org/citation.cfm?id=647698.734146`.