Distributive Interaction of Algebraic Effects

Kwok-Ho Cheung

Merton College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2017

Contents

Acknowledgements

As I and no doubt numerous many others have come to realise, completing a DPhil is no mean feat, and I would not be at the finishing line without the help of many people along the way. I would like to thank Jeremy for his patience, encouragement and guidance over the last few years. As he will know well, it has not been an easy ride but I am grateful for his willingness to push me into (occasionally) producing some good work. I am also indebted to Faris, "parachuted" in midway though the DPhil as my second supervisor. His arrival could not have been better timing for me—thank you for the many illuminating discussions over the whiteboard.

Let me also thank Ralf, Sam and Hongseok for their helpful advice as assessors at various junctions of the DPhil; Graham and Sam for agreeing to be my examiners at the viva. Sam deserves a third mention—my interest in the subject can be traced back to an essay I wrote under his guidance during my year at Cambridge. I would be amiss in failing to thank Merton and the University for their financial aid and counselling in helping me get over the final few hurdles. Last but certainly not least, my family for their love and support, and friends in Oxford, London and around the world for keeping me sane.

Abstract

While monadic effects are widespread in modern functional programming, the idea of formulating computational effects as algebraic theories seems a less familiar one to programmers. One appealing feature of such *algebraic* effects is the clear decoupling between specification and implementation (or in more model-theoretic terms, syntax versus semantics). With monads, this distinction is arguably less clear.

But perhaps the most compelling reason for considering algebraic effects is the relative ease by which such effects can be combined. This point is clearly of much relevance to the semantics of programming languages since in much of modern software development, one often deals with multiple interacting effects. As a simple example, we may want a program that not only keeps track of some state across the computation (e.g. a parser consuming a string of text), but also account for the possibility of failure. In Haskell, we can express the combination of "state" and "exception" as itself a monadic type. But it is well known that there is more than one way to combine these two effects, each corresponding to a different composition of the underlying functors. One choice of composition reverts the state to its original value in the event of a failure, whereas another choice of composition does not. Neither can be considered canonical, since both have their use cases. It turns out that under the lens of algebraic effects, both interactions can be understood in terms of straightforward amalgamations of the respective equational laws. Specifically, the latter is given by taking a simple union of the two theories, while the former in addition demands equations for commutativity between each pair of stateful and exceptional

operations. Both of these constructions arise naturally from the categorical structure of *Lawvere theories*—the more abstract formulation of equational theories, as categories.

In this dissertation we seek to further the understanding of combining effects, especially from this more algebraic perspective. Of particular interest is a *distributive tensor* construction on Lawvere theories that does not seem to be very widely known. Similar to the above, this takes a simple union of two theories, but demands additional equations for *distributivity* of operations in one theory over those of the other (e.g. in the same sense that multiplication distributes over addition in any ring-like algebraic structure). There are some clear parallels between this notion and distributive laws of monads (that underlie several of the most common monad transformers). We make some steps towards establishing a more precise relationship, giving examples where the two notions coincide. The distributive tensor plays a leading role in many examples of computational interest—two such applications will be considered here in some depth.

From the observation that various combinatorial search strategies are characterised by two equivalent formulations—as *bunch* monadic types, and as more structured theories of monoids—we give a number of correspondence results. In particular, the bunch type describing a kind of depth-bounded traversal is shown to be models of a distributive tensor. We also consider the free models of this theory, giving rise to a monad, and by imposing symmetry on the monoid operation, obtain another distributive tensor theory matching closely breadth-first traversal. Depth-first traversal is implicit in the list monad, and it is shown that the list monad transformer is exactly a distributive tensor from the theory of monoids. This is in contrast to the equational presentation of the alternative "done right" list transformer, which is clarified: while it too exhibits distributivity of the monoid operation, it does so crucially only in the leftwards direction and not the right.

The second application considers in some detail a derivation of the geometrically convex monad—the combination of probabilistic and nondeterministic choice effects, called *combined choice*—in a relatively simpler setting than the usual domain-theoretic presentation. As such, its characterisation as a distributive tensor is clearer. The results building up to this are then applied to equational reasoning. Under this lens, one is more able to identify an incorrect assumption in various equational axiomatisations of effects (such as probabilistic choice) overlooked in the literature. As it is no easy task to capture the equational properties between probabilistic and nondeterministic choice, a technique is explored for reasoning about such equations *visually* by taking a geometric interpretation of the free models of combined choice, as convex polygons on a plane.

CHAPTER 1

Introduction

A computational effect, also known as *notion of computation*, is a very general term encompassing a wide range of phenomena to be found in the many paradigms and guises of programming. These range from features typically supported in some shape or form within most modern programming languages—I/O, exceptions, state, to name a few—through to sophisticated and less conventional forms of computation, such as logic (or relational), probabilistic and quantum computation. Very rarely does the development of software not involve one or more such *effects* in its design. This can be of great relevance for example, in almost any computer code that interfaces with its external environment in some capacity—whether that be a user, a network or external storage, possibly in the presence of errors or even non-termination. To deal with such complexity requires more machinery than that offered by the evaluation of pure mathematical functions alone[1]. As such, in many instances we can view effects as enhancements to ordinary functions with some desired side-effect, delivered alongside an evaluated result. The way in which this result is computed is in turn determined by the particular effect. More generally speaking, one might think of an effect as providing a "computational structure" over a given type of values, and so an effectful function can be thought of as one that returns structured output, or output in a particular computational context.

The distinction between an ordinary and effectful function is perhaps most transparently seen when framed in the context of a pure functional

[1]As in the Lambda Calculus, which can be seen as the core of most modern programming languages, especially functional languages.

programming language such as Haskell [**22**]. One reason for this is that in many other programming languages, effects are (typically by design) implicit—every executable unit of code has the potential to perform certain effects, e.g. write values into mutable variables, throw exceptions, or interact with the console. By contrast, in Haskell the type system demands the programmer to be more explicit about the type of each function definition, to the extent that we may classify it as either pure (i.e. without effects) or "effectful"[2]. In Haskell terminology, for types a and b the former are functions of type $a \longrightarrow b$, while the latter are of type $a \longrightarrow m\ b$ where m is a type constructor (that is, $m\ b$ is a type parametrised by b) instance of the class Monad, the name originating from category-theoretic monads [**41**]. Indeed, from a categorical perspective, the effectful maps $a \longrightarrow m\ b$ just described can be understood as morphisms in the *Kleisli category*: roughly speaking, if we view the collection of data types and functions as forming the mathematical structure of a category, then the corresponding Kleisli category (for the monad m) consists of effectful functions $a \longrightarrow m\ b$ as its morphisms. For example, functions $a \longrightarrow \mathsf{IO}\ b$ potentially perform I/O, while functions $a \longrightarrow \mathsf{Maybe}\ b$ may possibly return a value Nothing instead of a value in b. In addition to the most common instances like IO and Maybe, the type class mechanism in Haskell allows the programmer to conveniently define one's own Monad instances, thereby extending the array of effects available.

As an aside, it is worth noting that even in the specific setting of functional programming (and in particular, Haskell), one might still find some difficulty in pinning down a precise—if not mathematical—definition of computational effect, for monads are not the only characterisation for it. Applicative functors (or idioms) [**47**] are a more recent discovery, and pervasive (every monad is by definition an applicative functor). Arrows [**23**] are a more powerful notion. Comonads—the dual notion of monads—have also

[2]Strictly speaking, even pure functions in Haskell are not entirely effect-free, for they still have the potential to fail or exhibit non-termination.

seen use. Nevertheless, although we acknowledge that monads are by no means the only abstraction for effects in functional programming, it is fair to say that it is without much doubt the most prominent one, and the one predominantly considered in this dissertation.

The use of monads from category theory in modelling computational effects first came to fruition in its use for structuring the denotational semantics of programming languages, beginning with Moggi [**51**], before it was later adapted into language design as a programming abstraction in its own right for structuring programs themselves [**69**]. The idea is that for each type A in the language, we introduce another type $M(A)$ of computations in A, with M a monad (typically on a cartesian closed category) representing some computational effect. That way, morphisms in the Kleisli category for M provide the semantics of effectful functions. The Haskell functions $a \longrightarrow m\ b$ mentioned above are of course a realisation of this idea. A further observation in this approach is that composition in the Kleisli category (which amounts to the operation $\gg\!=$ in Haskell's `Monad` type class) provides the semantics for "chaining together" effectful functions into bigger computations. In addition to providing a clean, compositional semantics for studying effectful languages, in terms of programming this leads to several practical benefits, such as code reuse and (with support from its **do**-notation) elimination of "boilerplate" code.

In a parallel development in mathematics, history also seems to have favoured category-theoretic monads in the study of universal algebra [**28**], becoming the most popular characterisation of algebraic theories. It was only much later that the ideas of Lawvere [**37**] permeated the field, offering an alternative "categorification" of algebraic theories that—with the obvious benefit of hindsight—seems more direct and natural. Nevertheless, when one restricts attention to "finitary" monads on the category of sets, there is an equivalence with Lawvere theories—a fact established by Linton [**40**]. In more recent times still, Lawvere theories have, as did monads, seen extensive

application in computer science, once again in the denotational semantics of computational effects. This has lead to an ongoing research programme of work carried out by the likes of Power, Plotkin, Hyland and others [**56, 58, 26**]. It is worth noting however that abstract formalisations of algebraic theories are not limited to Lawvere theories and monads alone (although in this dissertation at least, they will be our principal focus). For example, abstract clones from universal algebra have also been studied in connection with computational effects [**64**].

While monadic effects are widespread in modern functional programming, the idea that effects can be thought of as algebraic theories (stemming from the work of Power et al) seems generally a less familiar one to programmers. But arguably, it presents effects in a form many functional programmers readily understand. In its most concrete form, an algebraic theory consists of two parts—a set of operations, from which one is able to construct syntactic terms, and equational laws, each of which identify two particular terms. For example, Haskell's Monoid type class consists of two operations which one might refer to as *multiplication* and *unit*, with laws matching the same concept of monoid in algebra. One is then able to instantiate types that admit such "multiplication with unit" structure (*lists* being the canonical instance), which turns out to be very pervasive in programming. An appeal of this view of computational effect—sometimes called "algebraic effect"—is the very clear separation between specification and implementation (or, syntax versus semantics). With monads, this distinction is somewhat less clear. Recent interest and exploration in algebraic effects, together with the concept of *handlers* [**59**] have resulted in new languages and libraries [**33, 5, 46**], making it an important area of current research in functional programming and effects[3].

[3]While not a specific focus of this dissertation, Appendix A gives a brief account of handlers and selected implementations.

In most realistic software applications (and all but the most simple), one rarely deals with a single specific instance of effect at a time. A function intended to read a path input by the user to load a file into memory, say, will be deemed to require both interaction with the console, and file I/O. In this case, the IO monad in Haskell is in fact "large" enough[4] to accommodate both requirements, but what if in addition we would like this function to throw an exception in the case that the user supplies an invalid path, say? Clearly, a single monad alone is not enough in most nontrivial programming—one also needs the facility to combine different effects together. For example, a programmer may wish to account for the possible failures in computing a result (or have exceptional values), and she may also hope to keep track of some state across this computation. In Haskell, the usual method for combining effects is via monad transformers [**39**], which build a "stack" of monadic effects. In this example, two choices of combined monad are possible—MaybeT State and StateT Maybe—and they have different semantics. In other words, the two effects interact differently. How should we explain this interaction mathematically? As it turns out, under the lens of algebraic theories, such interaction can be understood in terms of equational laws.

Motivated in part by the lack of a clear consensus in tools beyond monad transformers for programming with effects, in this dissertation we seek to further the understanding of combining effects from a theoretical standpoint. For example, we are interested in constructions for combining Lawvere theories [**26**], known as *sum* and *tensor*. The former amounts to taking the (disjoint) union of two theories (both their operations and equations), while the latter in addition imposes commutativity between operations across the two theories. It turns out that the aforementioned monads MaybeT State and StateT Maybe can be explained precisely by sum and tensor, respectively. Of

[4]Though many would be quick to argue that IO is *too* large [**53**], encompassing many otherwise disparate sub-effects.

particular interest will be examples of combination whose interaction is best described equationally by distributivity [**27**] (in the same precise sense that multiplication distributes over addition in any ring-like algebraic structure), which has seen relatively little study thus far. Such interaction underlies the combination of probabilistic and nondeterminstic choice, as well as various forms of search strategy, when considered as algebraic effects. Much of this builds upon the work of Power, Plotkin and others on Lawvere theories of effects, although—perhaps due to the technical demands of its readership and the level of abstraction that they work upon—this body of work can appear both challenging and somewhat disconnected to a more general functional programming audience. Acknowledging this, in this dissertation we tend to work in more modest, concrete settings by some way of contrast. The reader is assumed to know only basic category theory, with references to Haskell code where appropriate.

1.1. Structure

The rest of the dissertation is structured as follows. Chapter 2 introduces the central (and closely connected) notions of *monad* and *algebraic theory*, by way of a motivating example of *state*—a computational effect familiar to most functional programmers. Monads from category theory are then covered in some detail, before the discussion moves to techniques in combining monadic effects, notably monad transformers. The chapter concludes with a brief review of related literature, where known shortcomings of monad transformers have been studied. The long-standing problem of "non-uniform" lifting of operations, for example, is discussed briefly. While monads do not always compose, there has been some research on addressing other approaches (coproduct of monads, in particular), or focusing on specific classes of such (ideal monads). When they do compose however, generalisations have been studied too (iterated distributive laws).

Chapters 3 and 4 introduce notions in universal algebra that will be assumed knowledge in later chapters. Algebraic theories and models are defined, both in terms of the classical presentation, and in more abstract categorical terms. Having defined a Lawvere theory, Chapter 4 contains definitions and examples of various constructions for combining such theories, including sum and tensor. It concludes with an account of the relatively little known distributive tensor, which will be important in the rest of the dissertation.

Both chapters 5 and 6 investigate the distributive tensor further, by considering two case studies of wide computational interest. Chapter 5 reviews Spivey's work on algebraic search strategies (including depth-first and breadth-first traversal), and considers the question of which equational laws one should impose to axiomatise such strategies. It also contains a study of two well known variants of the list (or backtracking) monad transformer. Chapter 6 addresses in some detail a result characterising the combination of probabilistic and nondeterminstic choice effects both in terms of a monad and a (distributive tensor) theory, concluding with an exposition of a diagrammatic reasoning method that results.

1.1.1. Contributions

As mentioned above, Chapters 2 and 3 are background material. The contents of Chapter 4 also begins with a continuation of background material, containing some folklore results from the literature. Notably, Corollary 4.2.3 characterises the two well-known combinations of the state and exception monads—essentially MaybeT State and StateT Maybe—as sum and commutative tensor of their respective theories. These are consequences of more general results about the exception transformer (Theorem 2.3.2) and state transformer (Theorem 4.1.3). Following this are specific contributions towards an (as far as we know) open question found in Section 4.3. There, we give an account of a simple example of a distributive law (of the

list monad over the bag monad) which turns out to coincide with the distributive tensor of the respective theories. We also show that a distributive tensor of the semigroup over unit theory is equivalent to the free *semigroup with zero* monad. This leads to the question of *when* the two notions—distributive tensor and distributive law—exhibit such coincidence generally, and is ongoing future work.

The second part of the dissertation contains contributions to two separate applications of combined effects where the distributive tensor plays a leading role. In Chapter 5, we give a number of correspondence results based on the following observation: that the essential structure of various combinatorial search strategies (such as depth-first and breadth-first) can be captured by two equivalent formulations—as *bunch* monadic types, and as more structured theories of monoids. The latter, specifically, by adding appropriate equations to the theory of monoids MON with a unary operation WRAP, and considering its normal forms. As such, we show that a bunch monad of *forests* arises from taking the sum of MON and WRAP i.e. without adding new equations (Proposition 5.3.1). We give two proofs of this—firstly by considering the free models of the sum-theory. The other proof (Corollary 5.3.4) has a more computational flavour by application of a theorem about the generalised resumptions monad transformer. While the list monad (characterising depth-first traversal) arises by discarding WRAP, other bunch types are shown to satisfy equations given by *adding* interaction between the two theories. In the case of depth-bounded traversal, such values are shown to be models of a distributive tensor of MON over WRAP (Theorem 5.3.7). We consider also the free models of this theory, giving rise to a monad of *fences* (Proposition 5.3.11, Theorem 5.3.13). Furthermore, by imposing symmetry on the monoid operation, we obtain a distributive tensor theory that matches breadth-first traversal closely, and while we show that the bunch type of "matrices" does not satisfy all requirements of being a model of this theory, we propose a refinement of it that is

(Proposition 5.3.16). Again we consider the free models, which gives rise to a monad of *bundles* (Proposition 5.3.20, Theorem 5.3.23).

The remainder of the chapter shifts focus to list computations, where the (`mtl`) list monad transformer is critiqued with respect to its application to practical stream programming. The main contribution here is a novel characterisation of this transformer as a distributive tensor from MON (Theorem 5.4.4). This is in contrast to the equational presentation of the alternative, "done right" list transformer which is clarified: while it too exhibits distributivity of the monoid operation, it does so crucially only in the leftwards direction and not the right.

A contribution of Chapter 6 is a derivation of the *geometrically convex* monad—the combination of probabilistic and nondeterministic choice effects, which we call *combined choice*—in a relatively simpler setting than the usual presentation of it in the (mostly domain theoretic semantics) literature. As such, its characterisation as a distributive tensor is clearer. Although this particular characterisation has been mentioned before in passing [**27**], we do not believe that our main correspondence result (Theorem 6.3.7) is widely known. The definitions and results building up to this are then applied in the remainder of the chapter, where the focus shifts to equational reasoning in Haskell (or a similar functional programming setting). Under this lens, another contribution of this work is the identification of an incorrect assumption in equational axiomatisations of various algebraic effects such as probabilistic choice, that has previously been overlooked [**19, 18**]. Specifically, it was thought that monadic bind $\gg=$ distributes over probabilistic choice, both in the leftward and rightward directions. It turns out the latter is a far more contentious property than the former (which just states algebraicity of the operation), particularly if one assumes that such properties carry over without fuss when combined with other effects. For example, Section 6.4 considers once more the algebraic effect of combined choice, where we have both probabilistic and nondeterministic choice operations. But we

show that rightwards distributivity of $\gg=$ over probabilistic choice implies its commutativity with *any* operation (Corollary 6.4.3), in particular nondeterministic choice. But this contradicts the usual equations of combined choice—in fact we show that in such case, the probabilistic choice operations collapse. Note that we also give a direct equational proof of this result in [**1**].

As it can be difficult to get the equational properties of the interaction between probabilistic and nondeterministic choice right, in the remainder of the chapter we explore a technique for reasoning about such equations visually, by taking a geometric interpretation of the free models of combined choice. That is, we interpret computations of the geometrically convex monad (over three values) as convex polygons on a plane. We illustrate the use of this diagrammatic model by exhibiting an inequality (specifically, non-distributivity of nondeterministic over probabilistic choice, Figure 6.6). Compared to our earlier exposition of the work on diagrammatic reasoning [**1**], we give a more detailed treatment here, and in addition, describe the interpretation of $\gg=$ (Definition 6.4.10).

Part 1

Background

CHAPTER 2

Monadic effects

Many functional programmers will be familiar with the concept of monad, which has seen widespread use both as a tool for modelling effects in programming language semantics, and as a programming abstraction for structuring effectful programs. In this chapter, we review the basic category theory of monads, their Kleisli and Eilenberg-moore categories, and distributive laws. In addition, we cover the concept of monad transformer, a standard tool in Haskell programming for combining monadic effects, before reviewing selected literature on other developments towards the problem of combining monads. In later chapters we will develop a complementary perspective on effects that emphasise their operations and equational laws. In what immediately follows, we provide some motivation.

2.1. Stateful effects

The term *algebraic effect* [**60**] is a relatively recent one, so-called to emphasise the view of an effect as a collection of operations and equational laws, as opposed to a monad. While the latter is more familiar to most functional programmers, the basic ideas of the former are readily grasped from many existing Haskell type classes. For example, consider the following class declaration for stateful effects[1].

[1] MonadState is a *multi-parameter* type class, for it is parameterised in two type variables m and s. The *functional dependency* $m \longrightarrow s$ in the declaration is simply a hint for the Haskell type checker, expressing the constraint that for each m there be no more than one s.

$$\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadState}\ s\ m \mid m \longrightarrow s\ \textbf{where}$$
$$put :: s \longrightarrow m\ ()$$
$$get :: m\ s$$

The MonadState type class captures the class of mutable state effects, for the case where there is just a single mutable value of type s. An instance for this is a type constructor m that is an instance of Monad and supports two methods *get* and *put* with definitions satisfying the following "get-put" laws[2] [**56**]

$$put\ s \gg put\ s' = put\ s' \tag{PP}$$
$$put\ s \gg get = put\ s \gg return\ s \tag{PG}$$
$$get \gg\!= put = return\ () \tag{GP}$$
$$get \gg\!= \lambda s \longrightarrow get \gg\!= \lambda s' \longrightarrow k\ s\ s' = get \gg\!= \lambda s \longrightarrow k\ s\ s \tag{GG}$$

Remark 2.1.1. The get-put laws are often presented as above, expressing how each pair of the two operations interact. However, it turns out that the law (GG) is redundant, for it can be shown to be derivable from the laws (GP) and (PG).

The laws appealingly capture the intuition for how the two stateful operations ought to behave and interact, and provided they hold, we can be reasonably confident that *get* and *put* have been implemented in a sensible way. But they are more than a mere sanity check for the programmer—MonadState describes what is essentially an algebraic theory, much in the same way that the Monoid type class is defined in terms of the operations $mempty :: a$ and $mappend :: a \longrightarrow a \longrightarrow a$ satisfying unit and associativity

[2]Note that such laws are not enforced by (or even expressible in) Haskell—thus they should be seen as proof obligations for the programmer.

laws

$$mappend\ mempty\ x = x$$

$$mappend\ x\ mempty = x$$

$$mappend\ x\ (mappend\ y\ z) = mappend\ (mappend\ x\ y)\ z$$

The difference is that the get-put laws are expressed with monadic operations $\gg\!=$ and *return*. We will present these laws differently (namely, without monadic operations) in what immediately follows.

Let us make this clearer by simplifying the state s to be a single bit

$$\textbf{data}\ \mathsf{Bit} = \mathsf{L} \mid \mathsf{R}$$

where the value of the bit of state[3] is either "left" or "right". We also employ the following translation to give alternative methods *putk* and *getk*

$$\begin{aligned}
&putk\ b\ k\ = put\ b \gg k\\
&getk\ k\ k' = get \gg\!= \lambda b \longrightarrow\\
&\quad \textbf{case}\ b\ \textbf{of}\\
&\qquad \mathsf{L} \longrightarrow k\\
&\qquad \mathsf{R} \longrightarrow k'
\end{aligned}$$

These can be thought of as continuation-passing style variants of the existing methods: *putk* writes b to the bit, continuing with the (monadic) computation k, and *getk* reads the bit, continuing with k if it was left, or k' for right.

Notice these encapsulate the use of the monadic operations, so that we can present the get-put laws in truly algebraic terms, that is, without

[3]Synonymous to the type Bool, but L and R are somewhat more suggestive when the laws are presented with *getk* an infix operator.

monadic binds and such:

$$
\begin{aligned}
putk\ b\ (putk\ b'\ k) &= putk\ b'\ k && b, b' :: \mathsf{Bit} \\
putk\ \mathsf{L}\ (k\ `getk`\ \_) &= putk\ \mathsf{L}\ k \\
putk\ \mathsf{R}\ (\_\ `getk`\ k) &= putk\ \mathsf{R}\ k \\
(putk\ \mathsf{L}\ k)\ `getk`\ (putk\ \mathsf{R}\ k) &= k \\
(k\ `getk`\ \_)\ `getk`\ (\_\ `getk`\ k') &= k\ `getk`\ k'
\end{aligned}
$$

The first law is in fact short for four equations, one for each combination of L,R. The idea is, whatever was written in the first *putk* gets overwritten by the second. The next two equations simply express that whatever is written to the bit determines whether to branch to the left or right computation, so that a read is superfluous. The next law says that to write back what was immediately read is the same as having done neither, and finally the last equation says that two consecutive reads will yield either the "leftmost" or "rightmost" computation.

What we have arrived at is an algebraic theory of state, a simplified instance of what is sometimes referred to as the theory of mnemoids [**56**]. To see the correspondence with monads, it can be shown that the free term algebra construction for this theory yields the familiar state monad[4]

$$\textbf{type}\ \mathsf{BitState}\ a = \mathsf{Bit} \longrightarrow (a, \mathsf{Bit})$$

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Monad}\ \mathsf{BitState}\ \textbf{where} \\
&\quad return\ x = \lambda b \longrightarrow (x, b) \\
&\quad k \gg\!= f \quad = \lambda b \longrightarrow \textbf{let}\ (x, b') = k\ b\ \textbf{in}\ f\ x\ b'
\end{aligned}
$$

[4]Note that strictly speaking, Haskell does not allow **type** aliases to be **instance** definitions—a restriction ignored here for convenience of exposition.

instance MonadState Bit BitState **where**

$$put\ b = \lambda b' \longrightarrow ((), b)$$
$$get\ \ \ = \lambda b \longrightarrow (b, b)$$

for our single Bit of state.

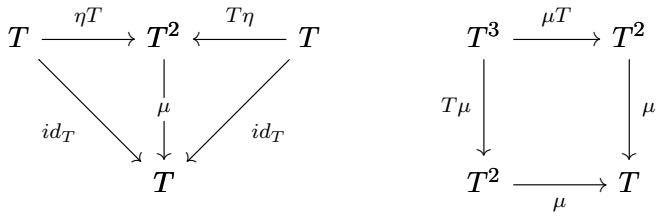
In summary, what we have is a specification of the state monad in purely algebraic terms. As illustrated in this example, this algebraic view of effects emphasises the operations and equational laws giving rise to the monad. Furthermore, it admits very natural mechanisms for combining, thus forming a modular theory of effects. In following chapters, we will study algebraic theories more carefully, before returning to more examples of their application to effects.

2.2. Monads and algebras

Moggi's notions of computation [**51**] (also called *computational effects*) have been very influential in the programming languages community. The breakthrough idea was to model such effects with monads, enriching basic type theory with terms having computational meaning. That is, for an object X of values, one assigns an object TX of values associated with a computational effect, modelled by a type constructor T. As examples, consider *partiality* as an effect $TX = X + \{\bot\}$ (which amounts to Maybe in Haskell) and *nondeterminism* $TX = \mathcal{P}^{\aleph_0}X$, the finite non-empty subsets of X.

Other examples include side-effects, exceptions and interactive I/O. In fact, the use of monads has seen prevalent use not only in structuring denotational semantics, but also in structuring programs themselves. Indeed, since being introduced in Haskell, the ideas have had much influence in many other mainstream programming languages [**21**]. We review this important concept and other relevant theory here. For a comprehensive treatment, see the standard texts, e.g. [**41**].

Definition 2.2.1. A *monad* (T, η, μ) on a category $\mathbf{C}$ consists of an endofunctor $T : \mathbf{C} \to \mathbf{C}$, and natural transformations $\eta : 1_{\mathbf{C}} \to T$, $\mu : T^2 \to T$ such that the following diagrams commute



For a category $\mathbf{C}$ with finite products, a monad on $\mathbf{C}$ is *strong* when it is equipped with a natural transformation

$$t_{A,B} : A \times TB \longrightarrow T(A \times B)$$

satisfying three commutative diagrams (see e.g. [**49**]). Most of the monads considered in this dissertation are strong.

It is well-known that every adjunction gives rise to a monad.

Proposition 2.2.2. *Every adjunction $F \dashv U$ with unit η and counit ϵ gives rise to a monad (T, η, μ) with $T = UF$, $\mu = U\epsilon F$.*

Furthermore, every monad arises from an adjunction. In fact, given a monad T, there are two canonical constructions for recovering an adjunction that gives rise to T—one yielding the *Kleisli* category and the other the *Eilenberg-Moore* category.

Definition 2.2.3. Let $\mathsf{T} = (T, \eta, \mu)$ be a monad on $\mathbf{C}$. The *Kleisli category* $\mathbf{C}_{\mathsf{T}}$ is defined as follows. Objects are just those of $\mathbf{C}$, and a morphism $f : A \to B$ in $\mathbf{C}_{\mathsf{T}}$ is a morphism $f' : A \to TB$ in $\mathbf{C}$. The identity id_A in $\mathbf{C}_{\mathsf{T}}$ is the unit $\eta_A : A \to TA$ in $\mathbf{C}$. For composition, given $f : A \to B$ and $g : B \to C$ in $\mathbf{C}_{\mathsf{T}}$, the composite $g \circ f$ is defined to be the following morphism in $\mathbf{C}$:

$$A \xrightarrow{f'} TB \xrightarrow{Tg'} TTC \xrightarrow{\mu_C} TC$$

Remark 2.2.4. It is sometimes more convenient to consider an equivalent formulation of a monad, called a *Kleisli triple*. This consists of an endomap T on the objects of $\mathbf{C}$, $\eta_A : A \to TA$ for objects A in $\mathbf{C}$ (as for the usual definition of a monad), and $\gg\!=_f : TA \to TB$ for $f : A \to TB$, subject to equations based on the unit and associativity equations of the Kleisli category. The Monad type class in Haskell is based on this formulation[5].

```
class Monad m where
  return :: a ⟶ m a
  (>>=)  :: m a ⟶ (a ⟶ m b) ⟶ m b
```

and the laws

$$return\ x \gg\!= f = f\ x$$

$$m \gg\!= return = m$$

$$(m \gg\!= f) \gg\!= g = m \gg\!= \lambda x \longrightarrow f\ x \gg\!= g$$

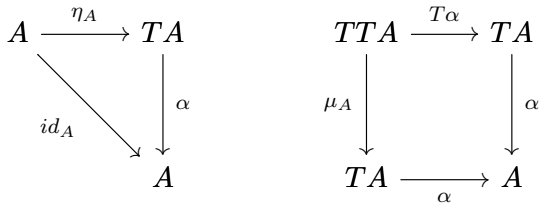
Furthermore, when $\mathbf{C}$ is cartesian closed, we have a *strong* Kleisli triple whenever $\gg\!=$ is a morphism.

The other construction for retrieving an adjunction from a monad is due to Eilenberg and Moore, for which we define the notion of an algebra for a monad.

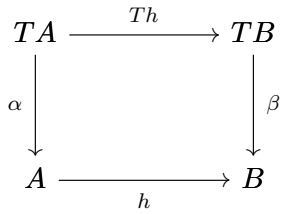
Definition 2.2.5. An *algebra* (A, α) for a monad $\mathsf{T} = (T, \eta, \mu)$ on $\mathbf{C}$, also called a T-algebra, is an object A and a morphism $\alpha : TA \to A$ (in other

[5]It is worth mentioning that as of GHC 7.10, by the Applicative-Monad proposal, instances of Monad are required to be instances of Applicative. See e.g. wiki.haskell.org for details.

words, a T-algebra) that respects the monad structure, i.e.



commute. A T-algebra *homomorphism* $h : (A, \alpha) \to (B, \beta)$ is a morphism $h : A \to B$ such that the square



commutes. T-algebras and homomorphisms form a category $\mathbf{C}^{\mathsf{T}}$ (or sometimes denoted T-**Alg**, especially when the base category is clear), the *Eilenberg-Moore category*.

If we start with an adjunction $F \dashv U : \mathbf{D} \to \mathbf{C}$, by Proposition 2.2.2 we have a monad on $\mathbf{C}$, and hence can construct $\mathbf{C}^{\mathsf{T}}$ for a monad T with functor UF (with associated adjunction $F^{\mathsf{T}} \dashv U^{\mathsf{T}}$). From this, a *comparison functor* $\Phi : \mathbf{D} \to \mathbf{C}^{\mathsf{T}}$ emerges, with $U^{\mathsf{T}} \circ \Phi \cong U$ and $\Phi \circ F = F^{\mathsf{T}}$. In fact, Φ is unique with this property.

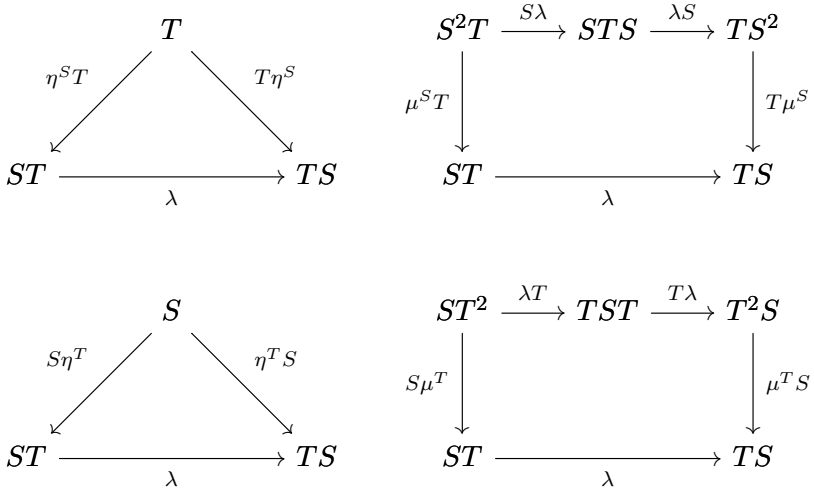
Definition 2.2.6. A functor $U : \mathbf{D} \to \mathbf{C}$ is *monadic* if it has a left adjoint $F \dashv U$ such that the induced comparison functor is an isomorphism of categories $\mathbf{D} \cong \mathbf{C}^{\mathsf{T}}$, for the monad T with functor part UF.

Examples of monadic forgetful functors $\mathbf{D} \to \mathbf{Set}$ include those from "algebraic" categories, such as monoids or groups. Indeed, the notion of monadicity can be thought of as a definition of being "algebraic". A notable non-example is the forgetful functor $U : \mathbf{Pos} \to \mathbf{Set}$ from posets. The left adjoint F is the discrete poset functor, and the monad induced has simply

$UF = id_{\mathbf{Set}}$ the identity as the functor part. Thus it is evident that the Eilenberg-Moore category is just **Set** itself.

Now let $\mathsf{T} = (T, \eta^T, \mu^T)$ and $\mathsf{S} = (S, \eta^S, \mu^S)$ be monads on $\mathbf{C}$. In general, the composite functor TS does not necessarily form a monad, unless we have a so-called distributive law [**8**] between the two monads.

Definition 2.2.7. Let T and S be as above. A *distributive law* of S over T is a natural transformation $\lambda : ST \to TS$ that interacts coherently with the respective monad structures, i.e. the following commute



If λ exists, we do indeed have a monad $\mathsf{TS} = (TS, \eta, \mu)$, now that μ can be expressed as the composite natural transformation

$$TSTS \xrightarrow{T\lambda S} TTSS \xrightarrow{\mu^T \mu^S} TS$$

Fact 2.2.8. The existence of a distributive law λ of S over T is equivalent to a *lift* of T to a monad T' on S-**Alg**, with T' consisting of a functor T' sending

$$(X, SX \xrightarrow{\theta} X) \longmapsto (TX, STX \xrightarrow{\lambda_X} TSX \xrightarrow{T\theta} TX)$$

Furthermore, TS-**Alg** $\cong \mathsf{T}'$-**Alg**.

2.3. Monad transformers

Monad transformers are a standard technique in Haskell for combining monadic effects together.

Definition 2.3.1 ([**39**])**.** Let $\mathbf{Mnd}(\mathbf{C})$ denote the category of monads on a category $\mathbf{C}$. A monad transformer is an function from monads on $\mathbf{C}$ to itself

$$F : |\mathbf{Mnd}(\mathbf{C})| \longrightarrow |\mathbf{Mnd}(\mathbf{C})|$$

with the requirement that for any monad T there is a monad morphism $T \to F(T)$.

By *monad morphism* we mean (as in [**50**]), for strong monads S and T on the same category, a natural transformation $S \to T$ preserving monad structure (in the usual sense). In Haskell, bona fide monad transformers are instances of the MonadTrans type class

```
class MonadTrans t where
  lift :: Monad m ⇒ m a ⟶ t m a
```

with *lift* satisfying the requirements of a monad morphism

$$lift \circ return = return$$

$$lift\ (m \gg\!= f) = lift\ m \gg\!= (lift \circ f)$$

For example, the state monad has a transformer counterpart StateT

```
newtype StateT s m a = StateT {runStateT :: s ⟶ m (a, s)}
```

That is, when m is an instance of Monad, so is StateT s m

```
instance Monad m ⇒ Monad (StateT s m) where
  return x  = StateT (λs ⟶ return (x, s))
  xm >>= f  = StateT (λs ⟶ do
    (x, s') ← runStateT xm s
    runStateT (f x) s')
```

And furthermore, operations in m may always be lifted into the transformed monad in a way that preserves the monadic operations

```
instance MonadTrans (StateT s) where
  lift xm = StateT (λs ⟶ do
    x ← xm
    return (x, s))
```

In a similar fashion, the exceptions monad has a transformer counterpart ErrorT, satisfying analogous structure. It turns out that ErrorT has a mathematical description in terms of a certain coproduct of monads, a subject we explore next. Note that StateT has a somewhat different description that requires further machinery. As such, we return to StateT in Chapter 4.

2.3.1. Exceptions transformer

As discussed in [**26**], if one component of a coproduct is fixed to the exception monad, while the other is parametrised over, this corresponds precisely to the exception monad transformer `ErrorT`.

For a set E (of exceptions), Let Σ_E^* denote the exception monad, with functor $(-) + E$. The notation $(-)^*$ is intentionally suggestive of a *free monad*, over the constant functor $\Sigma_E = - \mapsto E$.

Theorem 2.3.2. *Fix a monad* $\mathsf{T} = (T, \eta, \mu)$*. The sum* $\Sigma_E^* + \mathsf{T}$ *exists and is given by the composite monad* $\mathsf{T}\Sigma_E^*$*, consisting of the functor* $T(- + E)$*.*

It is instructive to outline a proof in here. The argument relies on the existence of a more specialised form of distributive law, namely that of an *endofunctor* S over T. This is very similar to the usual notion of Definition 2.2.7, except S is not required to form a monad, hence the first pair of commutative diagrams are not needed.

Lemma 2.3.3. *For any endofunctor Σ, there is a distributive law of ΣT over T, given by the natural transformation*

$$\Sigma TT \xrightarrow{\Sigma\mu} \Sigma T \xrightarrow{\eta\Sigma T} T\Sigma T$$

Since we always have such a distributive law, one can show that as in Fact 2.2.8, there is always a lift of T to a monad T' on ΣT-**Alg** (the algebras for the *endofunctor* ΣT). Specialising back to exceptions $\Sigma = \Sigma_E$, it is evident that by definition of constant functor, and in turn by monadicity established in Section 3.1.3,

$$\Sigma_E T\text{-}\mathbf{Alg} \cong \Sigma_E\text{-}\mathbf{Alg} \tag{2.1}$$

$$\cong \Sigma_E^*\text{-}\mathbf{Alg} \tag{2.2}$$

Thus the distributive law λ of $\Sigma_E T$ over T yields a monad T' on Σ_E^*-**Alg** (i.e. the Eilenberg-Moore category)—in other words, λ turns out to be a distributive law of Σ_E^* over T, in the usual sense of Definition 2.2.7. As a consequence, we have a composite monad $\mathsf{T}\Sigma_E^*$. Why should this be the sum $\Sigma_E^* + \mathsf{T}$?

To see the connection, note that by Fact 2.2.8 again, algebras for $\mathsf{T}\Sigma_E^*$ are isomorphic to algebras for T'. It is convenient now to take the perspective that T' is a monad on Σ_E-**Alg**, so that a T'-algebra is a Σ_E-algebra (X, θ) together with a homomorphism $\phi : T'(X, \theta) \to (X, \theta)$. By noting that λ (of $\Sigma_E T$ over T) has components

$$\Sigma_E TX = E \xrightarrow{\lambda_X} T\Sigma_E X = TE$$

and by definition of homomorphism and T', we have the following situation

$$\begin{array}{ccc} TE & \xrightarrow{T\theta} & TX \\ {\scriptstyle \lambda_X}\uparrow & & \downarrow{\scriptstyle \phi} \\ E & \xrightarrow[\theta]{} & X \end{array}$$

Now (X, θ) is plainly isomorphic to a Σ_E^*-algebra by 2.1, and it is clear that (X, ϕ) forms a T-algebra. Together we have precisely an algebra (X, θ, ϕ) for the sum $\Sigma_E^* + \mathsf{T}$ of monads [**16**].

Remark 2.3.4. As noted above, Theorem 2.3.2 generalises to any endofunctor Σ, hence to any computational effect that is the free monad over its signature. As well as exceptions, interactive I/O is another notable example of an effect whose theory does not stipulate axioms, and hence fits this framework. In other words, the interactive I/O monad transformer may also be explained as the sum of monads $\Sigma_{I/O}^* + \mathsf{T}$ (where $\Sigma_{I/O}$ denotes the signature of interactive I/O), given by the composite monad $\mathsf{T}\Sigma_{I/O}^*$.

2.4. Combining monads

Despite the popularity of monad transformers, they are not without their limitations. In this section we briefly review some of the literature on this and various developments building on the ideas of monad transformers, as well as the general area of monad composition.

2.4.1. Distributive laws

It is well-known that given two monads S, T, the composite functor TS does not in general form a monad. That is, unless there is a distributive law of monads S over T. But while monads do not always compose in this sense, a wide class of combined computational effects nevertheless arise in this way e.g. the exceptions monad transformer as discussed is defined by composition with $- + E$. Another example from mathematics: in **Set**, the free monoid monad is given by a distributive law of the free semigroup monad S over the "point" monad $T = - + 1$. This shows that (the theory of) a monoid may be "decomposed" as a composite of a semigroup and a point.

So a distributive law is what one needs in order to compose two monads. To compose three monads A, B, C however, it turns out one needs more than just distributive laws, as discussed in [**13**]. Let us say we have

distributive laws B over A, C over A and C over B, giving rise to composite monads AB, AC and BC respectively. Now, given these, it is easy to give (composite) natural transformations $CAB \Rightarrow ABC$ and $BCA \Rightarrow ABC$. Are these distributive laws? In general no, unless A, B and C additionally satisfy a coherence law known as the *Yang-Baxter* equation. In such case, the two distributive laws give rise to the same monad ABC. Furthermore, even when composing more than three monads, the same conditions apply—a distributive law between each pair of monads (in one direction), and similarly Yang-Baxter for each triple of monads. This data is referred to as an *iterated distributive law* [**13**].

2.4.2. Coproduct of monads

Although composing monads via distributive laws may not always be possible, in theory one is always able to take coproducts. However, the general categorical definition of a coproduct of monads is somewhat involved—in particular, being a quotient type (rather than a free type) makes it difficult to implement, say, in Haskell. For one needs a "representative" for each equivalence class, as well as a decision procedure for deciding the equivalence of terms under the appropriate equations. In general the latter is undecidable.

A term of a monad coproduct $S+T$ can be thought of as a "layered" syntactic tree, consisting of S-layers interleaved with T-layers. In the approach taken in [**16**], one restricts attention to *layered* monads—monads equipped with an "inverse unit" η^{-1}. The intuition is that such monads can inspect whether a term is in the image of the unit, i.e. a variable. It turns out that this extra structure is sufficient in computing a representative (or *witness*), without the need for a more general decision procedure. For example, given a term of the free coproduct type, one reduction rule uses η^{-1} to quotient away variable layers; another rule uses the respective monad multiplications μ to collapse adjacent layers.

Coproducts of monads generalise distributive laws—whereas the former considers all possible interleaving layers of S and T, the latter can be seen as just one possible interleaving e.g. TS. There is however a notion of *strong* distributive law (the exceptions monad transformer is one such example). When one has a strong distributive law, the induced composite monad TS and the coproduct $S + T$ coincide [**16**].

Again, when considering a binary coproduct of monads as all (finite) interleavings of layers of the two monads, it was the ability to distinguish between trivial (i.e. variable) and nontrivial layers that proved a key insight. Indeed in later work [**17**], *ideal* monads $T = Id + T_0$ (as introduced by Adamek) were observed to do precisely that. Free monads are examples of ideal monads, as are all the usual computational examples of interest. Given two such monads, their coproduct is the ideal monad of the form $Id+(T_1+T_2)$ i.e. a computation is either a variable layer, a T_1-computation—an interleaving of layers starting with the first monad, or a T_2-computation—an interleaving starting with the second monad. In more recent work still, it was shown that this formula for determining the coproduct of ideal monads extends to an even wider class of *consistent* monads [**2**].

2.4.3. Lifting

The use of binary combinators such as coproduct of monads, sum and tensor of theories, are all forms of *compositional* semantics. Monad transformers on the other hand are unary constructs that take monads and transforms them into "bigger" monads. As such, the latter are also known as a form of *incremental* semantics. The two are by no means entirely distinct from each other. We have seen that the algebraic approach to effects emphasises the presence of operations—and in practice when working with monads and transformers, one also finds that they are equipped with operations (e.g. the get and put operations of the state monad). This however presents a long-standing problem with monad transformers: when a monad

is transformed into a bigger monad, all the associated operations are required to be *lifted* into the bigger monad. Since definitions of such liftings have traditionlly been on an ad hoc (i.e. per case) basis, it is not uncommon to find non-uniform behaviour across related transformers. In particular, one would expect a lifted operation of one monad transformer to behave the same way as the same (or analogous) lifted operation of a more general monad transformer. But as liftings are ad hoc, there is no such guarantee.

To address this problem of *non-unform lifting* requires a careful consideration of the taxonomy of both operations and transformers [**30**]. For example, algebraic operations turn out to have unique liftings. For more general (i.e. non-algebraic) operations, the situation depends on the nature of the transformer that the operation is to be lifted through (see also [**32**] for the lifting theorems generalised to the monoidal category setting). The ideas of this work have been implemented in the Monatron library [**31**] as an alternative monad transformer library. As mentioned, algebraic operations are lifted automatically (via its unique lifting property), and even non-algebraic operations (e.g. handle) can be lifted uniformly by exploiting the properties of the codensity monad. By offering uniform liftings, the library also has the desirable effect of reducing the number of instances a programmer traditionally would write when defining a new transformer, from quadratic to linear.

CHAPTER 3

Algebraic theories and models

In this chapter, we develop in some detail the idea of an algebraic theory. One advantage of this approach in modelling effects is that our intuition about what should be axioms of the computational effect are made explicit, thereby justifying the monad that is to be constructed. Starting concretely from (what are sometimes called) *presentations* of algebraic theories, from there we build intuition for the more abstract notion of Lawvere theory.

3.1. Universal algebra

We will start by reviewing some basic universal algebra. Recall that a *signature* Σ consists of a set of operators (or *operation symbols*), and an assignment to each operator a natural number—its *arity*.

In the classic (set-theoretic) sense, an *algebra* for a signature Σ (also called a Σ-algebra) is a set S and for each operator o (having arity n, say), a function $[\![o]\!] : S^n \to S$. Σ-algebras are denoted as tuples $(S, [\![-]\!])$. A Σ-algebra *homomorphism* from $(S, [\![-]\!]_S)$ to $(T, [\![-]\!]_T)$ is a function $f : S \to T$ that preserves the operations, in the sense that

$$f([\![o]\!]_S(x_1, \ldots, x_n)) = [\![o]\!]_T(f(x_1), \ldots, f(x_n))$$

for each $o \in \Sigma$, and $x_1, \ldots, x_n \in S$. We are interested in equational theories (Σ, E) i.e. a signature together with equations E. An algebra for such a theory is then a Σ-algebra that "satisfies" E, in a sense that will be made precise in what follows.

In the spirit of categorical logic, we can easily generalise these basic definitions, such that a signature (or theory) may be interpreted in general

categories—not necessarily **Set**. To this end, let us fix a category **C** with finite products.

Definition 3.1.1. For a signature Σ, a Σ-*algebra* $(X, [\![-]\!])$ is an object X and for each Σ-operator o with arity n, a morphism $[\![o]\!] : X^n \to X$ in **C**. A Σ-algebra *homomorphism* from $(X, [\![-]\!]_X)$ to $(Y, [\![-]\!]_Y)$ is a morphism $f : X \to Y$ that preserves the Σ-operations, in the sense that the following diagram

$$\begin{array}{ccc} X^n & \xrightarrow{[\![o]\!]_X} & X \\ {\scriptstyle f^n}\downarrow & & \downarrow{\scriptstyle f} \\ Y^n & \xrightarrow[[\![o]\!]_Y]{} & Y \end{array}$$

commutes, for each Σ-operator o having arity n. Σ-algebras and homomorphisms form a category Σ-**Alg**.

Assume that we have a collection of variables $x_1, x_2, \ldots$ at our disposal. Together with the operators of a signature Σ, we have the ingredients to build *terms* with respect to Σ, defined inductively: a variable x is a term; and for terms $t_1, \ldots, t_n$ and an operator o of Σ with arity n, $o(t_1, \ldots, t_n)$ is a term. Now, if one understands morphisms $[\![o]\!]$ of an algebra as "interpreting" operators o, how are terms interpreted? In fact, under the present categorical treatment, we always consider a term t under a "context" of distinct variables $\Gamma = x_1, \ldots, x_k$ rather than in isolation. A *term in context* is a judgment of the form $\Gamma \vdash t$ where all variables occurring in t are among those in Γ. Our notion of algebra can now be seen to extend to terms in the following.

Fact 3.1.2. Given a Σ-algebra $(X, [\![-]\!])$, its interpretation of a Σ-term t in context $\Gamma = x_1, \ldots, x_k$ is a morphism $[\![\Gamma \vdash t]\!] : X^k \to X$ defined inductively on the structure of t:

$[\![\Gamma \vdash x_i]\!] = \pi_i$ i.e. the product projection

$[\![\Gamma \vdash o(t_1, \ldots, t_n)]\!]$ is the composite of the product morphism with $[\![o]\!]$,

$$X^k \xrightarrow{\langle [\![\Gamma \vdash t_i]\!] \rangle_{i=1\ldots n}} X^n \xrightarrow{[\![o]\!]} X$$

where o is a n-ary operator.

Definition 3.1.3. An *algebraic theory* (Σ, E) is a signature Σ together with a set E of equations, where an *equation* $\Gamma \vdash t = u$ is nothing more than a pair of Σ-terms t, u in context Γ.

Given a theory (Σ, E) we will often refer to Σ-algebras as its *interpretations*. We can now formalise the notion of an algebra for a theory as being those interpretations that satisfy E, as in the following.

Definition 3.1.4. For $\Gamma = x_1, \ldots, x_k$, an interpretation $(X, [\![-]\!])$ *satisfies* an equation $\Gamma \vdash t = u$ if the respective interpretations of t, u yield an equal pair of morphisms, i.e.

$$[\![\Gamma \vdash t]\!] = [\![\Gamma \vdash u]\!] : X^k \longrightarrow X$$

A (Σ, E)-*algebra* is an interpretation that satisfies all equations in E. A *homomorphism* of algebras is defined in exactly the same way as for the signature-only case. (Σ, E)-algebras and homomorphisms form a category (Σ, E)-**Alg**, a full subcategory of Σ-**Alg**.

Notice that an interpretation for (Σ, E) may well satisfy equations not in E. Indeed, we would expect that a (Σ, E)-algebra satisfy not only all equations in E, but also those that are derivable from E via the usual rules of equational logic (hence one often makes a distinction between *axioms* in E, and *theorems* derivable from E). This soundness result is embodied in the following.

Theorem 3.1.5. *A (Σ, E)-algebra satisfies any theorem derivable from the axioms E.*

Remark 3.1.6. In classic denotational semantics based on set theory, one often finds in the literature that naked terms are interpreted with respect to an "environment" ρ—a finite partial function from variables to elements of the algebra X. Under this setting, an equation is satisfied by X if the interpretations of the pair of terms are equal elements of X, with respect to all environments. Our definition based on terms in context, on the other hand, is expressed purely in arrow-theoretic terms rather than elements, thus generalises to arbitrary finite-product categories other than **Set**. See [**55**] for a discussion.

3.1.1. Theory of groups

To make the previous discussion in this section somewhat more concrete, let us consider the theory $\mathbb{G} = (\Sigma, E)$ of groups. Recall that in most presentations of groups, Σ consists of a binary operator m ("multiplication"), a unary operator i ("inverse") and a nullary operator / constant symbol u ("unit"). So a Σ-algebra in $\mathbf{C}$ is simply an object X and three morphisms

(1) $[\![m]\!] : X^2 \to X$
(2) $[\![i]\!] : X \to X$
(3) $[\![u]\!] : 1 \to X$

A homomorphism of Σ-algebras is just a morphism f satisfying three copies of the commuting diagram from Definition 3.1.1, one for each of m, i and u. Together they form a category Σ-**Alg**, sometimes called the category of *group structures*.

Now consider the axioms E of group theory, namely the associativity, inverse and unit laws. These can be expressed in terms of the operators and

variables x, y, z as follows

$$x, y, z \vdash m(x, m(y, z)) = m(m(x, y), z) \tag{3.1}$$

$$x \vdash m(x, i(x)) = u \tag{3.2}$$

$$x \vdash m(i(x), x) = u \tag{3.3}$$

$$x \vdash m(x, u) = x \tag{3.4}$$

$$x \vdash m(u, x) = x \tag{3.5}$$

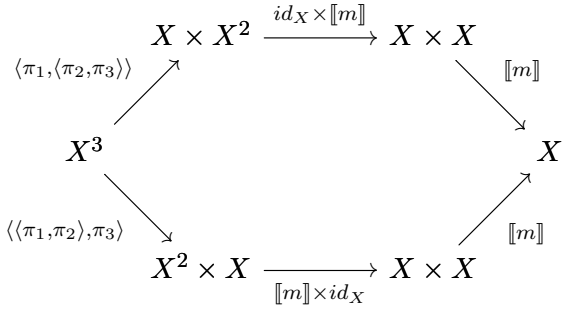
To interpret axiom (3.1), first see that the left-hand-side term is interpreted as the morphism

$$\begin{aligned} [\![m(x, m(y, z))]\!] &= [\![m]\!] \circ \langle \pi_1, [\![m(y, z)]\!] \rangle \\ &= [\![m]\!] \circ \langle \pi_1, [\![m]\!] \circ \langle \pi_2, \pi_3 \rangle \rangle \\ &= [\![m]\!] \circ (id_X \times [\![m]\!]) \circ \langle \pi_1, \langle \pi_2, \pi_3 \rangle \rangle \end{aligned}$$

and by a similar argument, the right-hand-side is interpreted as

$$[\![m(m(x, y), z)]\!] = [\![m]\!] \circ ([\![m]\!] \times id_X) \circ \langle \langle \pi_1, \pi_2 \rangle, \pi_3 \rangle$$

And so for a group structure to satisfy the associativity axiom amounts to these being equal morphisms, thus exhibiting a commutative diagram



The other axioms can be interpreted in a similar way, producing more commutative diagrams. Of course, a group structure that is a $\mathbb{G}$-algebra will satisfy all of these, as well as any theorem of group theory. The category $\mathbb{G}$-**Alg** of algebras for $\mathbb{G}$ in $\mathbf{C}$ is also called the category of *groups in* $\mathbf{C}$.

Example 3.1.7. The category $\mathbb{G}$-**Alg** of groups in **Set** is precisely the usual category **Grp** of groups and group homomorphisms.

Of course, the power of studying algebra in more general categories means that we are not limited merely to set-theoretic models of group theory, or any algebraic theory for that matter. To illustrate further, consider what it means to have a group in the category **Grp** itself. This uses the "Eckmann-Hilton" argument, which we state here.

Proposition 3.1.8. *For any set S equipped with two binary operations $\bullet, \star : S \times S \to S$ with respective units $1^\bullet, 1^\star$, such that the operations commute with each other,*

$$(x_1 \bullet y_1) \star (x_2 \bullet y_2) = (x_1 \star x_2) \bullet (y_1 \star y_2) \tag{3.6}$$

the operations in fact coincide $\bullet = \star$, with unit $1^\bullet = 1^\star$, and is associative and commutative.

Corollary 3.1.9. *The category $\mathbb{G}$-**Alg** of groups in* **Grp** *comprise precisely the abelian groups.*

Proof. Let G be a group in **Grp**. As it is an object of **Grp**, it has the usual structure $(G, \bullet, 1^\bullet, (-)^{-1})$ of a group, and since it is also an *internal* group, there are group homomorphisms $[\![m]\!] : G^2 \to G$, $[\![u]\!] : 1 \to G$, $[\![i]\!] : G \to G$ together satisfying the commutative diagrams described above for E. For clarity, denote $[\![m]\!]$ infix as $\star$, $[\![u]\!]$ as $1^\star$, and note that the group G^2 has multiplication defined pointwise. Now since $\star$ is a group homomorphism, it will preserve multiplication, i.e. precisely the equation (3.6). Hence by Proposition 3.1.8, G is abelian.

Conversely, take any abelian group $A = (A, \diamond, 1^\diamond, (-)^{-1})$. We can define the usual interpreting maps $[\![m]\!], [\![u]\!], [\![i]\!]$ on A as precisely $\diamond, 1^\diamond, (-)^{-1}$ respectively. It is not difficult to check that these are group homomorphisms:

for example, denoting $\diamond^2$ as multiplication of the group A^2 (defined pointwise),

$$[\![m]\!]((a_1, a_2) \diamond^2 (b_1, b_2)) = (a_1 \diamond b_1) \diamond (a_2 \diamond b_2) = (a_1 \diamond a_2) \diamond (b_1 \diamond b_2)$$

where the last equality makes use of associativity and commutativity of $\diamond$. Hence $(A, \diamond, 1^\diamond, (-)^{-1})$ constitutes a group structure in **Grp**. Furthermore, it satisfies all the commutative diagrams for associativity, unit and inverse axioms arising from E (above), by virtue of being a group in the usual sense. Therefore it is a group in **Grp**. □

Remark 3.1.10. Notice that the proof does not make use of the inverse operation of a group. So in fact, by discounting inverses altogether we arrive at a similar result for monoids. Namely, monoids in **Mon** are exactly the commutative monoids.

A monad on a category $\mathbf{C}$ can also be seen as an internal monoid, in the category $\mathbf{C}^\mathbf{C}$ of endofunctors. But note that this is not quite an example of the foregoing, since $\mathbf{C}^\mathbf{C}$ is not being considered as a category with products[1].

3.1.2. Theory of nondeterminism

Let us consider some examples from computational effects, starting with (binary) nondeterminism. Using the tools from Section 3.1 we will derive the finite non-empty subset monad $\mathcal{P}^{\aleph_0}$.

The theory $\mathbb{S}$ of semilattices consists of a binary operator $\sqcap$, and for variables x, y, z the following axioms

$$x \vdash x \sqcap x = x \tag{3.7}$$

$$x, y \vdash x \sqcap y = y \sqcap x \tag{3.8}$$

$$x, y, z \vdash x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z \tag{3.9}$$

namely idempotence, commutativity and associativity.

[1]Indeed, in this setting it is seen as a *monoidal* category

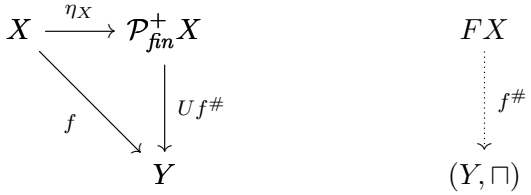
An interpretation in **Set** is a set X and an interpreting function $[\![\sqcap]\!] : X^2 \to X$. As usual, this interpretation extends to terms in context, and if this respects the axioms above then it is a $\mathbb{S}$-algebra. For example, take X to be $\mathcal{P}\mathbb{N}$—the powerset of the natural numbers, with set union $\cup$ for $[\![\sqcap]\!]$. As $\cup$ is idempotent, commutative and associative, it clearly satisfies the axioms. Note also that we could equally have taken set intersection to interpret $\sqcap$, so that $(\mathcal{P}\mathbb{N}, \cap)$ is another $\mathbb{S}$-algebra.

As one might expect, the category $\mathbb{S}$-**Alg** of algebras is the category of semilattices and semilattice homomorphisms. The evident forgetful functor U into **Set** has a left adjoint F, namely the free semilattice functor. This takes a set X to the semilattice with carrier $\mathcal{P}^{\aleph_0}X$ (the set of finite non-empty subsets of X) and union $\cup$ as the semilattice operation. The action on functions $f : X \to Y$ gives the semilattice homomorphism Ff that sends $S \in \mathcal{P}^{\aleph_0}X$ to $\{f(x) \mid x \in S\}$ i.e. the direct image of S.

Let us very briefly justify that we indeed have an adjunction $F \dashv U$. Take any set X. For any semilattice $(Y, \sqcap)$ and function $f : X \to Y$, define a semilattice homomorphism $f^{\#} : FX \to (Y, \sqcap)$ to be the map

$$S \longmapsto \bigsqcap \{f(x) \mid x \in S\}$$

(with $\bigsqcap$ defined in terms of $\sqcap$ in the obvious way). Now define a function $\eta_X : X \to \mathcal{P}^{+}_{fin}X$ as simply the inclusion $x \mapsto \{x\}$, and we have the commutative triangle below left



which can be verified by a simple diagram-chase. Furthermore, it is clear that $f^{\#}$ is uniquely defined with this property, hence FX is free over X with universal arrow η_X. By Proposition 2.2.2, the induced monad has functor part $UF = \mathcal{P}^{\aleph_0}$.

3.1.3. Theory of exceptions

Consider the computational effect of *exceptions*. As an algebraic theory, it comprises a family $(\texttt{raise}_e)_{e \in E}$ of nullary operators `raise`, indexed by a set E of exceptions, with no axioms. Alternatively, we may just as well consider `raise` as a *single* operator, with E as its *coarity*. More generally, the coarity of an operator is an object of the interpreting category (though here we stay in the setting of **Set** for simplicity).

Remark 3.1.11. As an aside, the theory of Section 3.1 generalises to accommodate coarities. For example, an interpretation $(X, [\![-]\!])$ of a theory is an object X and, for each operator $o : n \to C$ (that is, with arity n and coarity C) in the theory, a morphism $[\![o]\!] : C \times X^n \to X$. A homomorphism from $(X, [\![-]\!]_X)$ to $(Y, [\![-]\!]_Y)$ is a morphism $f : X \to Y$ that preserves the operations, i.e. the diagram

$$\begin{array}{ccc} C \times X^n & \xrightarrow{[\![o]\!]_X} & X \\ {\scriptstyle id_C \times f^n}\downarrow & & \downarrow{\scriptstyle f} \\ C \times Y^n & \xrightarrow[[\![o]\!]_Y]{} & Y \end{array}$$

commutes, for each operator $o : n \to C$.

Pursuing a similar analysis to Section 3.1.2, we can determine the monad given by the signature of exceptions. Notice by Remark 3.1.11, any interpretation $[\![\texttt{raise}]\!]$ will have domain E, hence the signature functor Σ_E is simply the constant functor $- \mapsto E$. From here, the derivation is somewhat simpler than for nondeterminism—for since there are no axioms, the monad we seek is precisely the free monad (on **Set**) over Σ_E, which we denote Σ_E^*. It is not difficult to show that it has functor part $- + E$.

3.1.4. Monads from presentations

In the above we have seen examples of monads derived from the theories of semilattices and exceptions, respectively. Here we outline this process in slightly more general terms.

Fix an algebraic theory $\mathbb{T} = (\Sigma, E)$. Now for any set X, let $T(X)$ denote the set of well-bracketed terms (equivalently, abstract syntax trees) t built from operations $\sigma \in \Sigma$ and variables $x \in X$, quotiented by the equivalence generated from the equations E. More precisely, we have equivalence classes $[\,t\,]$ in $T(X)$ generated by the following inductive rules

$$\frac{}{[\,x\,]} \qquad \frac{[t_1]\ \ldots\ [t_n]}{[\sigma(t_1, \ldots, t_n)]}$$

for any operation σ with arity n.

The operations are interpreted syntactically, that is

$$[\![\sigma]\!]([t_1], \ldots, [t_n]) \triangleq [\sigma(t_1, \ldots, t_n)]$$

It is not difficult to check that $T(X)$ with the above interpretations $[\![-]\!]$ is indeed the free algebra over X, and that this construction extends to a functor $\mathbf{Set} \to \mathbb{T}\text{-}\mathbf{Alg}$—indeed, left adjoint to the evident forgetful functor. Thus we have not only a functor $T : \mathbf{Set} \to \mathbf{Set}$ but also a monad, with the following structure

$$\eta_X : X \longrightarrow T(X)$$

$$x \longmapsto [x]$$

$$\mu_X : T^2(X) \longrightarrow T(X)$$

$$[t([t_1], \ldots, [t_k])] \longmapsto [t(t_1, \ldots, t_k)]$$

where an arbitrary $T^2(X)$ term $[\,t\,]$ can be seen as a derived operation (with say, arity k) with "variables" $[t_1], \ldots, [t_k]$ in $T(X)$.

It is reasonably straightforward to check that this monad indeed corresponds to the theory $\mathbb{T}$ by inspection of their respective algebras. Specifically, algebras of the monad T on X are in bijective correspondence with

$\mathbb{T}$-algebras on X. In fact, more can be said of this correspondence—it turns out the monad T is finitary (the functor T is completely determined by its action on *finite* sets), and going in the other direction to the above, we also have that every finitary monad on **Set** gives rise to a corresponding algebraic theory. We refer to e.g. [**61**] for details of this construction (but also see later in Section 3.2.3), only noting that when proceeding as such, it becomes apparent that such monads represent equational theories in a form that is independent of any particular presentation of operations and axioms. This also turns out to be the case when equational theories are considered as *categories*, as in the following.

3.2. Theories categorically

In this section we wish to understand the notion of Lawvere theory. The previous development of interpreting theories in general categories could reasonably be described as *categorical* semantics. The next step in terms of "categorification" is sometimes called *functorial* semantics.

To motivate this let us revisit our example of group theory. Although our presentation of the theory $\mathbb{G}$ of groups in Section 3.1.1 is similar to the usual "textbook" definition of groups, it is by no means unique. For example, a group may also be axiomatised as an alternative theory $\mathbb{G}'$ with a unit and a binary operator d called "double division" (and a single axiom—though the details are not important here) and nothing else. What is the relationship between these two formulations of group theory? We may consider d a *derived* operation of $\mathbb{G}$, with $d(x, y)$ equating to the term-in-context $x, y \vdash m(i(x), i(y))$. There is a similar translation of the operators of $\mathbb{G}$ in terms of $\mathbb{G}'$. Since the two theories are derivable from each other, the two formulations are essentially no different, in the sense that they give rise to the same algebras (namely groups)—the difference is purely syntactic.

Indeed, any term built from the basic operations m, i and u of $\mathbb{G}$ corresponds to a derived operation, thereby implying an infinite number of possible such presentations of group theory. Now there may be good reasons to prefer one formulation over another, but this aside, a natural question to ask is what then is a canonical *syntax-invariant* notion of algebraic theory – that is, one that is independent of any particular presentation? In the study of universal algebra, an *abstract clone* captures precisely this, and in Lawvere's thesis [**37**] this was given a categorical treatment.

3.2.1. Syntactic construction

Intuitively, if we are to represent an algebraic theory in a way that does not favour any particular presentation, we ought to treat all derivable operations with equal status, as it were. In other words, we make no distinction between basic and derived operations—instead treating all possible operations as "basic". Similarly with equations, we take all possible theorems as axioms. We can collect this data into a category of contexts and "context arrows", which we describe below.

Fix a theory $\mathbb{T}$. Recall that Γ denotes a context of variables. Let us say that a *context arrow* from $\Gamma_k = x_1, \ldots, x_k$ to $\Gamma_n = x_1, \ldots, x_n$ is an n-tuple $t = (t_1, \ldots, t_n)$ of terms-in-context $\Gamma_k \vdash t_i$ (for each $i = 1, \ldots, n$) of $\mathbb{T}$. In particular, we can think of context arrows into the single-variable context $\Gamma_k \to x$ as simply the k-ary operations of $\mathbb{T}$. Equality of a pair of context arrows t, t' is defined pointwise: in more detail, $\Gamma_k \vdash (t_1, \ldots, t_n) = (t_1^{'}, \ldots, t_n^{'})$ iff we have equations $\Gamma_k \vdash t_i = t_i^{'}$ in $\mathbb{T}$ for every $i = 1, \ldots, n$. This relation is in fact an equivalence.

Composition of context arrows is defined by substitution of terms, and the identity context morphisms on $x_1, \ldots, x_n$ are simply tuples $(x_1, \ldots, x_n)$. We can now define a category embodying the syntax-invariant content of $\mathbb{T}$ by taking its morphisms to be *equivalence classes* of context arrows.

Definition 3.2.1. For an algebraic theory $\mathbb{T}$, the *syntactic* (or *classifying*) *category* $C_\mathbb{T}$ has as objects contexts Γ of variables. A morphism $\Gamma \to \Gamma'$ is an equivalence class of context arrows from Γ to Γ', for the equivalence relation identifying t, t' just when $\Gamma \vdash t = t'$ in $\mathbb{T}$. Explicitly, a morphism into $\Gamma_n = x_1, \ldots, x_n$ is an equivalence class of judgements

$$[\Gamma \vdash (t_1, \ldots, t_n)] : \Gamma \longrightarrow \Gamma_n$$

Composition is induced by substitution of context arrows, and identities are equivalence classes of identity context arrows.

Example 3.2.2. The syntactic category for the theory of groups will have as morphisms $\Gamma \to x$ all operations we previously considered "basic", as well as all those derivable from them. In particular, multiplication $m(x_1, x_2)$ is a morphism $x_1, x_2 \to x$, and so too is double division $d(x_1, x_2)$. Of course, some derived operations such as the latter have meaningful names, but the vast majority will not! The equations of group theory are encoded into the morphisms: for example, the terms $d(x_1, x_2)$ and $m(i(x_1), i(x_2))$ are equal as context arrows, and therefore part of the same equivalence class. More generally, morphisms $\Gamma_k \to \Gamma_n$ are simply (equivalence classes of) n-tuples of k-ary operations.

Proposition 3.2.3. *$C_\mathbb{T}$ has all finite products.*

Proof. It is not difficult to check that the product of $\Gamma = x_1, \ldots, x_m$ and $\Gamma' = x_1, \ldots, x_n$ is the context $\Gamma \times \Gamma' = x_1, \ldots, x_{m+n}$. The terminal object is the empty context. □

Note that the finite product structure of $C_\mathbb{T}$ yields the property that any context $x_1, \ldots, x_n$ is a finite power n of the single-variable context $x = x_1$.

The syntactic category construction can be seen as a "categorification" of an equational presentation by means of saturating it, in some sense—in doing so, we no longer make a distinction between basic and derived operations, nor do we distinguish between equational axioms and derived

theorems. Put another way, in the syntactic category we take all derived operations and equations as basic. Looking at this category more abstractly, we may observe the essential structure required to define when a category describes, in a syntax-invariant way, an algebraic theory.

Definition 3.2.4. A *Lawvere theory* is a small category $\mathbb{L}$ consisting of objects A^n, one for each natural number $n = 0, 1, \ldots$ with a finite product structure given by $A^m \times A^n = A^{m+n}$ and terminal object $1 = A^0$. In particular every object is a finite power n of the generating object $A = A^1$.

Of course, the notation $A^0, A^1, A^2, \ldots$ for objects is merely suggestive[2], and we could just as well use the natural numbers themselves. This leads to an equivalent definition, more commonly found in the literature: let $\mathbb{N}^{\mathrm{op}}$ denote the opposite category of natural numbers and all functions. A Lawvere theory is a small category $\mathbb{L}$ with finite products, equipped with a (strict) finite-product preserving, identity-on-objects functor

$$\mathbb{N}^{\mathrm{op}} \longrightarrow \mathbb{L}$$

Remark 3.2.5. A Lawvere theory is a *single-sorted* algebraic theory, in reference to the single generating object A in our definition. That said, the notion extends to *multi-sorted* theories, which correspond to general finite product categories (with multiple such generating objects). An example of a presentation giving rise to such: the theory of *random access memory* consists of two sorts M and D (memory locations and data values, respectively), with operations $\texttt{lookup} : M \to D$ and $\texttt{update} : M \times D \to M$. We merely note that multi-sorted effects of this kind also require a different treatment in terms of monads, but will not consider them any further.

While a Lawvere theory generally has many presentations, there is always a maximal one (sometimes called its "internal language"). Thus every Lawvere theory has at least one presentation.

[2]The notation is particularly appealing in light of the fact that products are given by addition.

3.2.2. Functorial semantics

Let us revisit the notion of an algebra for a theory, in this Lawvere setting.

Definition 3.2.6. A *model* of a Lawvere theory $\mathbb{L}$ in a category $\mathbf{C}$ with finite products is a finite-product preserving functor $M : \mathbb{L} \to \mathbf{C}$. A *homomorphism* of models is a natural transformation $\alpha : M \to M'$ between models. Models and homomorphisms form a category $\mathsf{Mod}(\mathbb{L}, \mathbf{C})$.

Remark 3.2.7. For any model M, we can recover our earlier notion of algebra. First observe that by preservation of products, $M(A^n) \cong (MA)^n$, i.e. the action on objects is determined (up to isomorphism) by the action on the generating object A. And similarly by the universal property of products, the action on morphisms $A^n \to A^m$ is determined by the actions on $A^n \to A$, for all n. Therefore, a model in $\mathbf{C}$ amounts to an object $X = MA$, together with all morphisms $X^n \to X$.

It is also interesting to note that the definition of homomorphism of models does not insist on respecting the product structure of $\mathbb{L}$, as one might expect. As it turns out, this would be superfluous, since it is a property that can be shown to be implied, by naturality alone.

In accordance with the definition of model, we claim that the analogue of a $\mathbb{T}$-algebra in Section 3.1 must be a finite-product preserving functor—indeed, one from the syntactic category $C_{\mathbb{T}}$. Let us make this statement more precise, by considering a $\mathbb{T}$-algebra in $C_{\mathbb{T}}$ itself. By definition, this is an interpretation U consisting of the single-variable context x, with (an equivalence class of) terms

$$[\![o]\!] = [x_1, \ldots, x_n \vdash o(x_1, \ldots, x_n)] : x_1, \ldots, x_n \to x$$

for each operator o of arity n in (the signature of) $\mathbb{T}$. A term-in-context is interpreted as its equivalence class. Thus the equations are satisfied, and U is indeed an algebra. In the literature, U is called the *generic algebra* of $\mathbb{T}$,

with a defining property of it being *complete*, in the sense that it satisfies only the equations of $\mathbb{T}$, and nothing more.

Now notice that for any model $M : C_{\mathbb{T}} \to \mathbf{C}$, the action of M on U is also a $\mathbb{T}$-algebra, by virtue of the fact that algebras "travel along" finite-product functors, in the sense of the following.

Fact 3.2.8. Finite-product preserving functors preserve algebras of algebraic theories.

Proof. Let $F : \mathbf{D} \to \mathbf{E}$ be a finite product-preserving functor. Consider an algebra $(X, [\![-]\!])$ in $\mathbf{D}$ for a theory, consisting of morphisms $[\![o]\!] : X^n \to X$ for every operator o of arity n. F sends these to morphisms

$$F(X^n) \cong (FX)^n \xrightarrow{F[\![o]\!]} FX$$

in $\mathbf{E}$ so that $(FX, F[\![-]\!])$ is an interpretation. The analysis for terms is similar: for an equation-in-context $x_1, \ldots, x_k \vdash t = u$, its interpretation in $\mathbf{D}$ as the equal pair of morphisms $[\![t]\!] = [\![u]\!] : X^k \to X$ is sent, via F, to the equal pair of morphisms

$$F(X^k) \cong (FX)^k \xrightarrow{F[\![t]\!]=F[\![u]\!]} FX$$

in $\mathbf{E}$ so that $(FX, F[\![-]\!])$ indeed satisfies all equations of the theory. □

So MU really is a $\mathbb{T}$-algebra in $\mathbf{C}$. Also note it can be shown that a homomorphism $\alpha : M \to M'$ of models $M, M' : C_{\mathbb{T}} \to \mathbf{C}$ gives rise to a $\mathbb{T}$-algebra homomorphism $MU \to M'U$ by taking the component α_U (by naturality – the details omitted here for brevity). Thus the act of evaluating at U induces a functor

$$ev_U : \mathsf{Mod}(C_{\mathbb{T}}, \mathbf{C}) \to \mathbb{T}\text{-}\mathbf{Alg}(\mathbf{C})$$

In fact, it can be shown (see Awodey and Bauer's lecture notes [**3**]) that this functor is a natural equivalence of categories.

Proposition 3.2.9. *For any algebraic theory* $\mathbb{T}$ *and finite-product category* $\mathbf{C}$*, there is an equivalence*

$$\mathsf{Mod}(C_{\mathbb{T}}, \mathbf{C}) \simeq \mathbb{T}\text{-}\mathbf{Alg}(\mathbf{C})$$

natural in $\mathbf{C}$*, between the category of models of* $C_{\mathbb{T}}$ *and the category of* $\mathbb{T}$*-algebras.*

Returning to the claim earlier in this subsection, we see that a $\mathbb{T}$-algebra X really does correspond to a finite-product functor, M_X say, from $C_{\mathbb{T}}$, and furthermore M_X is determined uniquely up to equivalence, with the generic $\mathbb{T}$-algebra U acting as mediator—i.e. $M_X U \cong X$. Thus M_X is said to classify (or, is the classifying functor of) X. The above has shown the precise sense in which, when theories are treated as categories, their algebras correspond to functors.

3.2.3. Correspondence with monads

For completeness we summarise the correspondence between Lawvere theories and monads (due to [**40**]), as we did earlier in Section 3.1.4 with equational presentations.

Given a monad T on $\mathbf{Set}$, let $\mathbf{Kl}_T^{\mathrm{op}}$ denote the opposite of its Kleisli category. Then a Lawvere theory $\mathbb{L}_T$ can be obtained by considering the Kleisli maps between finite sets—more precisely, we take $\mathbb{L}_T$ to be the full subcategory of $\mathbf{Kl}_T^{\mathrm{op}}$ whose objects are those of $\mathbb{N}$. Indeed, its models coincide with the algebras—specifically, we have an equivalence

$$\mathsf{Mod}(\mathbb{L}_T, \mathbf{Set}) \simeq T\text{-}\mathbf{Alg}$$

between the category of models of this Lawvere theory and the Eilenberg-Moore category.

Conversely, given a theory $\mathbb{L}$, Lawvere showed that the "forgetful" functor

$$U : \mathsf{Mod}(\mathbb{L}, \mathbf{Set}) \longrightarrow \mathbf{Set}$$
$$M \longmapsto M(A)$$

has a left adjoint. Linton went further to show that U is monadic—that the induced monad $T_{\mathbb{L}}$ is *equivalent* to $\mathbb{L}$ in the same precise sense as above. The process from $\mathbb{L}$ to a (finitary) monad $T_{\mathbb{L}}$ can be expressed in terms of a coend[3]

$$T_{\mathbb{L}}(X) = \int^{n \in \mathbb{N}} \mathbb{L}(n, 1) \times X^n$$

[3]Ends (and their dual, coends) are treated in many texts, notably Mac Lane [**41**].

CHAPTER 4

Combining algebraic effects

In programming, one typically works with multiple interacting computational effects, whether implicitly, or more explicitly—for example, as in Haskell via monads. But given two monads, it is not always clear how they "combine", if at all. In specific instances such as the exceptions monad transformer (considered in Section 2.3), one obtains a composite monad (equivalently, a distributive law). In general however, composites are not guaranteed to exist. On the other hand, Lawvere theories admit very natural constructions for combining one another. For example, unlike the general case for monads, coproducts are very simple to describe, at least at the equational level. Quite simply, given two theories $\mathbb{T}$ and $\mathbb{S}$, their *sum* $\mathbb{T} + \mathbb{S}$ comprises all operations from both theories (renaming where necessary, to avoid clashing), and similarly all equations from both theories, and nothing more.

Briefly, in this chapter we will also formally define and describe a tensor product on theories, and its relation to commutative monads in concrete terms. We give some examples of both sum and tensor (in particular, by relating to specific instances of monad transformers). Finally, we give one more construction on theories characterised by distributivity, which will play a prominent role in chapters that follow.

While this chapter is largely a continuation of background material, the choice of its presentation deserves a brief mention. As a general theme, we discuss various abstract concepts relating to Lawvere theories in a concrete setting, relating the discussion to equational reasoning in effectful functional programming, where appropriate. As such, the categorical definitions of

commutative tensor (Section 4.1), algebraicity (Section 4.2) and distributive tensor (Section 4.3) are (unlike the more semantics-oriented presentation in the literature) cast in somewhat more familiar programming terms.

The main example of commutative tensor is the explication of the state monad transformer, in Section 4.1. Relating to the commutative tensor, the basic notion of commutativity—both of a theory and of a monad, is also discussed in this section. The latter in particular requires mention of *strength*, and is presented concretely in a Haskell setting. In fact, Haskell functors are strong and thus enriched—a fact that appears to not be widely known to programmers. While the equivalence between commutative monads and commutative theories may be a well known fact to some, for concreteness we also give details of (one direction of) this equivalence.

Section 4.2 introduces *algebraicity*, providing theoretical justification for equations of the form *monadic bind* $\gg=$ *distributes (leftwards) over operations*—often stated as axioms in the literature on equational reasoning of effectful functional programs (e.g. [**19, 18**]). This section also considers the two well known combinations of the state and exception monads (which we call backtracking and persistent state), from the perspective of commutative tensor and sum of their respective theories—a result [**26**] that deserves to be more common knowledge.

We mention specific contributions towards an (as far as we know) open question, found in Section 4.3. There, we give an account of a simple example of a distributive law of monads, which turns out to coincide with the distributive tensor of the respective theories. This leads quite naturally to the question of *when* the two notions exhibit such coincidence generally. For instance, we do not see this agreement in the case of the list monad, which can be built from a distributive law (specifically, of nonempty lists over Maybe) but is not the composite monad arising from the analogous distributive tensor. Indeed, it turns out that the latter is equivalent instead to a different distributive law (again of nonempty lists over Maybe) that gives

the free *semigroup with zero* monad. It remains ongoing work to investigate the precise relationship between the two notions of distributive combination.

4.1. Commutative effects

We give an abstract definition of tensor product, as seen in [**26**]. As the name suggests, together with the initial Lawvere theory (the "empty" theory with no operations nor equations, giving rise to the identity monad) as unit, it forms a symmetric monoidal structure on the category of Lawvere theories.

Definition 4.1.1 ([**26**])**.** Let $f : n \to n'$ and $g : m \to m''$ be maps in Lawvere theories $\mathbb{T}$ and $\mathbb{S}$ respectively. The *tensor product* $\mathbb{T} \otimes \mathbb{S}$ is the Lawvere theory defined by the universal property of having maps of Lawvere theories from $\mathbb{T}$ and $\mathbb{S}$ to $\mathbb{T} \otimes \mathbb{S}$, with commutativity of the diagram

$$\begin{array}{ccc} n \times m & \xrightarrow{n \times g} & n \times m' \\ {\scriptstyle f \times m}\downarrow & & \downarrow{\scriptstyle f \times m'} \\ n' \times m & \xrightarrow[n' \times g]{} & n' \times m' \end{array}$$

This abstract definition deserves some further explication. $\mathbb{T} \otimes \mathbb{S}$ is the theory consisting of all operations and equations of both $\mathbb{T}$ and $\mathbb{S}$ (i.e. their sum), together with commutativity of every operation f in $\mathbb{T}$ with every operation g in $\mathbb{S}$. For this reason, this construction is also called *commutative* tensor[1]. Concretely, for n-ary f and m-ary g as stated, we have the equation[2]

$$f \begin{pmatrix} g(x_{11}, \ldots, x_{1m}) \\ \vdots \\ g(x_{n1}, \ldots, x_{nm}) \end{pmatrix} = g(f \begin{pmatrix} x_{11} \\ \vdots \\ x_{n1} \end{pmatrix}, \ldots, f \begin{pmatrix} x_{1m} \\ \vdots \\ x_{nm} \end{pmatrix}) \tag{4.1}$$

[1]Not to be confused with the notion of a (binary) operation *being* commutative (or *symmetric*), e.g. $x + y = y + x$. Note that some authors also like to call $\otimes$ the *Kronecker product*.

[2]We write the arguments of f vertically, for readability.

Informally, we may think of the top left of the commutative square as a $n \times m$ matrix of arguments

$$\begin{pmatrix} x_{11} & \dots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nm} \end{pmatrix}$$

Then the left (right) hand side of equation (4.1) corresponds to going along (down) and then down (along) the diagram. Note that this is an equation between $n' \times m'$ matrices of values, as determined by the coarities of f and g, although in examples this is typically a singular value i.e. $n' = m' = 1$.

Hence by definition, any operation with arity 0 or 1 automatically commutes with itself, and any two commuting nullary operations must necessarily be equal. We say that a theory $\mathbb{T}$ is *commutative* if every pair of operations f, g in $\mathbb{T}$ satisfies equation (4.1). Another way to express this is the slogan "all operations are homomorphisms", for f (as a map) preserves g (as an operation), and vice versa.

4.1.1. State monad transformer

Recall the algebraic presentation of the computational effect *state*, in Section 2.1. We mentioned there that this algebraic theory, in the terminology of Section 3.1, consists of two operations $\mathit{getk} : 2 \longrightarrow 1$ and $\mathit{putk} : 1 \longrightarrow 2$, subject to the get-put laws. Furthermore, it gives rise to the usual state monad with functor $(S \times -)^S$ (in the case of a single bit of state $S = 2$).

Example 4.1.2. The tensor product is easily described in terms of operators and equations, which we illustrate with an example of our theory of state combined with nondeterminism. The combined theory has all the operators from both (*getk*, *putk* and $\vee$), as well as all axioms from both, and in addition demands commutativity of each operator in one theory with each in the other. That is, is an additional equation

$$x_1, x_2, y_1, y_2 \vdash \mathit{getk}(x_1 \vee y_1, x_2 \vee y_2) = \mathit{getk}(x_1, x_2) \vee \mathit{getk}(y_1, y_2)$$

expressing that to follow a non-deterministic computation determined by a lookup of state equates to a non-deterministic choice of lookup computations. Similarly, there is an equation for commutativity of update with $\vee$:

$$x, y \vdash putk_b(x \vee y) = putk_b(x) \vee putk_b(y)$$

where the subscript b ranges over both values of the state 2.

Again, we may consider this in the more general setting of combining state with an *arbitrary* effect, and moreover, ask what is the equational explanation for the state monad transformer StateT. While the sum $\mathbb{L}_E + \mathbb{L}$ of Lawvere theories—rephrased in terms of a sum of monads—was the right notion for the exceptions transformer (and similarly for interactive I/O, see section 2.3.1), the presence of *equations* in the theory of state distinguishes itself as a different class of effect, hence it does not fit within that same framework. Indeed, the tensor product that turns out to be the notion corresponding to StateT—more precisely, the monad corresponding to $\mathbb{L}_S \otimes \mathbb{L}$, for $\mathbb{L}_S$ denoting the Lawvere theory of state.

Theorem 4.1.3 ([**26**])**.** *Let T be the functor part of a monad corresponding to a Lawvere theory $\mathbb{L}$. The monad induced by the tensor product $\mathbb{L}_S \otimes \mathbb{L}$ is isomorphic to the composite monad consisting of the functor $T(S \times -)^S$.*

We sketch the pertinent ideas of the proof, assuming **Set** as the base category. The argument rests upon a certain characterisation of the tensor product, namely in terms of models discussed in Section 3.2.2.

Fact 4.1.4. For any category $\mathbf{C}$ with finite products, and Lawvere theories $\mathbb{L}$ and $\mathbb{L}'$, there is a coherent equivalence of categories between $\mathsf{Mod}(\mathbb{L} \otimes \mathbb{L}', \mathbf{C})$ and $\mathsf{Mod}(\mathbb{L}, \mathsf{Mod}(\mathbb{L}', \mathbf{C}))$.

As an aside, it is very possible to obtain some intuition for this particular characterisation by recalling the Eckmann-Hilton argument in Section 3.1.1. Take $\mathbb{L}_G$ to be the Lawvere theory of groups. In this Lawvere setting,

$\mathsf{Mod}(\mathbb{L}_G, \mathbf{Set})$ corresponds to the usual set-theoretic groups, i.e. **Grp**, and we know from Corollary 3.1.9 that $\mathsf{Mod}(\mathbb{L}_G, \mathbf{Grp})$ are the abelian groups. Now since Lawvere theories are *semantically invariant* (precisely: equivalent models imply an isomorphism of the underlying theories), together with the above fact, we see that the tensor product $\mathbb{L}_G \otimes \mathbb{L}_G$ is isomorphic to the Lawvere theory of abelian groups.

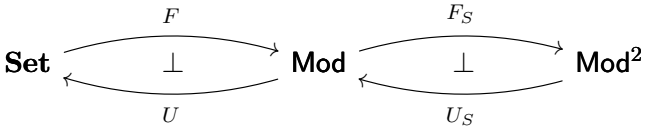
Returning to our proof outline, we can invoke Fact 4.1.4 to observe an equivalence

$$\mathsf{Mod}(\mathbb{L}_S \otimes \mathbb{L}, \mathbf{Set}) \simeq \mathsf{Mod}(\mathbb{L}_S, \mathsf{Mod}(\mathbb{L}, \mathbf{Set}))$$

Let us work with the right hand side. First, observe that there is a forgetful functor $U : \mathsf{Mod}(\mathbb{L}, \mathbf{Set}) \to \mathbf{Set}$ given by evaluation at the generating object. In the case of **Set** as we have here, U is monadic, thus it has a left adjoint F giving rise to a monad with functor $T = UF$. But note also the forgetful functor

$$U_S : \mathsf{Mod}(\mathbb{L}_S, \mathsf{Mod}(\mathbb{L}, \mathbf{Set})) \longrightarrow \mathsf{Mod}(\mathbb{L}, \mathbf{Set})$$

defined in an entirely similar way to U. Again, this has a left adjoint F_S, inducing the state monad (with $U_S F_S = (S \times -)^S$) on $\mathsf{Mod}(\mathbb{L}, \mathbf{Set})$. Schematically, we have the following adjunctions



where for brevity, $\mathsf{Mod}(\mathbb{L}, \mathbf{Set})$ has been shortened to just Mod, and Mod^2 is shorthand for $\mathsf{Mod}(\mathbb{L}_S, \mathsf{Mod}(\mathbb{L}, \mathbf{Set}))$. By composing the adjunctions, $F_S F \dashv U U_S$ induces a monad with functor

$$\begin{aligned} U U_S F_S F &= U(S \times -)^S F \\ &= U(S \times F-)^S \end{aligned}$$

Now recall that left adjoints preserve coproducts, so that $S \times F- \cong F(S \times -)$, and dually as right adjoints preserve products, $U(-)^S \cong (U-)^S$. Taken

together, we have

$$\begin{aligned} U(S \times F-)^S &\cong (UF(S \times -))^S \\ &\cong T(S \times -)^S \end{aligned}$$

Thus we have shown that the monad corresponding to the tensor product $\mathbb{L}_S \otimes \mathbb{L}$ is indeed isomorphic to StateT, as desired.

4.1.2. Commutative monads

A commutative theory can also be formulated abstractly in terms of a monad. First, recall that a monad (T, η, μ) is strong [**36**] is when it is equipped with a natural transformation t called *strength*, and its symmetric counterpart s

$$t_{A,B} : A \times TB \longrightarrow T(A \times B)$$

$$s_{A,B} : TA \times B \longrightarrow T(A \times B)$$

satisfying certain coherence laws. For example, monads in Haskell are known to be strong

$$\begin{aligned} &t :: \mathsf{Monad}\ m \Rightarrow (a, m\ b) \longrightarrow m\ (a, b) \\ &t\ (x, ym) = ym \gg\!= \lambda y \longrightarrow return\ (x, y) \end{aligned}$$

Indeed, functors in Haskell are strong—in other words, $\mathsf{Functor}$ structure alone is sufficient to define strength

$$\begin{aligned} &t :: \mathsf{Functor}\ f \Rightarrow (a, f\ b) \longrightarrow f\ (a, b) \\ &t = uncurry\ rst \\ &\quad \textbf{where} \\ &\qquad rst\ x = fmap\ (\lambda y \longrightarrow (x, y)) \end{aligned}$$

and similarly for its counterpart

$$\begin{aligned} &s :: \mathsf{Functor}\ f \Rightarrow (f\ a, b) \longrightarrow f\ (a, b) \\ &s = uncurry\ (flip\ lst) \end{aligned}$$

where

$lst\ y = fmap\ (\lambda x \longrightarrow (x, y))$

In the setting of a monoidal closed category (and in particular for applications in computer science, a cartesian closed category) V, to give a strong functor T on V amounts to V-enrichment on T [**36**]. In this sense, instances of Functor in Haskell can indeed be seen as enriched, for *fmap* in

class Functor f **where**

$fmap :: (a \longrightarrow b) \longrightarrow f\ a \longrightarrow f\ b$

is considered a function $(a \longrightarrow b) \longrightarrow (f\ a \longrightarrow f\ b)$ between *function types* i.e. (exponential) objects in the category, rather than between sets.

Given strengths, we can define the following composite functions—referred to in [**52**] as (*left* and *right*) *derived prestrengths*

$lpstrength, rpstrength :: \mathsf{Monad}\ m \Rightarrow (m\ a, m\ b) \longrightarrow m\ (a, b)$

$lpstrength\ = (\gg\!= t) \circ s$

$rpstrength = (\gg\!= s) \circ t$

Intuitively, given a pair of computations (xm, ym), e.g. two effectful arguments of a function, the computation *lpstrength* corresponds to evaluation of arguments in left-to-right order, while *rpstrength* corresponds to evaluation right-to-left. Therefore, the equation

$$lpstrength = rpstrength \tag{4.2}$$

represents the situation when the choice of evaluation order (and hence the order in which m-effects are executed) is immaterial. This equation, expressed in monadic **do**-notation, yields the usual Haskell definition of a commutative monad.

Definition 4.1.5. A monad m in Haskell is commutative if we have the following equation between computations in $m\ (a, b)$

$$\begin{array}{lll} \textbf{do} & x \leftarrow xm & \qquad\qquad \textbf{do} \quad y \leftarrow ym \\ & y \leftarrow ym & = \qquad\quad\; x \leftarrow xm \\ & return\ (x, y) & \qquad\qquad\; return\ (x, y) \end{array}$$

for $(xm, ym) :: (m\ a, m\ b)$.

It is worthwhile justifying equation (4.2) in more detailed, concrete terms. Writing $(\gg\!= f)$ as the (equivalent) composite $\mu \circ T(f)$, the left derived prestrength *lpstrength* is in turn the composite map

$$TA \times TB \xrightarrow{s_{A,TB}} T(A \times TB) \xrightarrow{Tt_{A,B}} TT(A \times B) \xrightarrow{\mu_{A\times B}} T(A \times B)$$

Working in the category **Set**, denote $\underrightarrow{a}_n \vdash f$ to mean a n-ary term f in a context of variables $\underrightarrow{a}_n = a_1, \ldots, a_n$ in A. Similarly, let $\underrightarrow{b}_m \vdash g$ be a m-ary term g in a context of variables in B. Consider such a pair of terms

$$\underrightarrow{a}_n \vdash f, \quad \underrightarrow{b}_m \vdash g$$

in $TA \times TB$ (technically *equivalence classes* of such terms) and send it through the composite map above. The action of the "left" strength $s_{A,TB}$ distributes g over $\underrightarrow{a}$, in the sense that it is paired with each variable in $\underrightarrow{a}$ to give a $(A \times TB)$-context to f

$$\begin{array}{l} (a_1,\ \underrightarrow{b}_m \vdash g) \\ \qquad\vdots \qquad\quad \vdash f \\ (a_n,\ \underrightarrow{b}_m \vdash g) \end{array}$$

Then for each of its variables $(a_i, \underrightarrow{b} \vdash g)$, the action of $T(t_{A,B})$ in turn distributes a_i over $\underrightarrow{b}$

$$\begin{array}{l} \Big((a_1, b_1), \ldots, (a_1, b_m) \vdash g_1\Big) \\ \qquad\qquad\vdots \qquad\qquad\quad \vdash f \\ \Big((a_n, b_1), \ldots, (a_n, b_m) \vdash g_n\Big) \end{array}$$

Finally, the monad multiplication $\mu_{A\times B}$ sends the n-ary f to a mn-ary operation which we write $f(g_1,\ldots,g_n)$—in full,

$$\begin{matrix}(a_1,b_1) & \ldots & (a_1,b_m)\\ \vdots & \ddots & \vdots\\ (a_n,b_1) & \ldots & (a_n,b_m)\end{matrix} \quad \vdash f\begin{pmatrix} g_1\\ \vdots\\ g_n\end{pmatrix}$$

Similarly by sending the same pair of terms

$$\underrightarrow{a}_n \vdash f, \quad \underrightarrow{b}_m \vdash g$$

along the right derived prestrength i.e. the composite map

$$TA\times TB \xrightarrow{t_{TA,B}} T(TA\times B) \xrightarrow{Ts_{A,B}} TT(A\times B) \xrightarrow{\mu_{A\times B}} T(A\times B)$$

gives the mn-ary term $g(f_1,\ldots,f_m)$ defined in the analogous way. Thus a *commutative* monad T implies equality between the two mn-ary terms

$$\begin{matrix}(a_1,b_1) & \ldots & (a_1,b_m)\\ \vdots & \ddots & \vdots\\ (a_n,b_1) & \ldots & (a_n,b_m)\end{matrix} \quad \vdash f\begin{pmatrix} g_1\\ \vdots\\ g_n\end{pmatrix} = g(f_1,\ldots,f_m) \tag{4.3}$$

It is clear that by identifying (a_i,b_j) with x_{ij}, the equation (4.3) is tantamount to equation (4.1).

4.2. Sum and tensor examples

In the following, we give examples of algebraic operations in equational form, and present the two distinct combinations of state and exception monads as sum and tensor.

Definition 4.2.1 ([**58**])**.** Let T be a strong monad on a category with finite products. An n-ary *algebraic operation* with coarity m on T is a natural transformation

$$\alpha : (T-)^n \longrightarrow (T-)^m$$

equivalently, m families of maps

$$\alpha_A : (TA)^n \longrightarrow TA$$

with not only the usual naturality in A but also *Kleisli naturality*: for every map $f : A \to TB$ the following diagram commutes

$$\begin{array}{ccc} (TA)^n & \xrightarrow{\langle (\gg\!= f) \circ \pi_i \rangle_{i=1\ldots n}} & (TB)^n \\ {\scriptstyle \alpha_A}\downarrow & & \downarrow{\scriptstyle \alpha_B} \\ TA & \xrightarrow[\gg\!= f]{} & TB \end{array}$$

Note that $\gg\!= f$ is sometimes called the *Kleisli extension* of f.

An algebraic operation for T with arity n and coarity m as above is sometimes denoted $n \to m$ when the monad is clear from context. In addition, it can be presented equivalently as a Kleisli arrow $m \to Tn$ (note the change in direction), sometimes called a *generic effect*.

Example 4.2.2. Consider the following type class

class Monad $m \Rightarrow$ MonadPlus m **where**

$$\begin{aligned} \varepsilon \quad &:: m\ a \\ (\oplus) &:: m\ a \longrightarrow m\ a \longrightarrow m\ a \end{aligned}$$

To say that its operations are algebraic amounts to the statement that $\gg\!=$ distributes leftwards over them.

$$\varepsilon \gg xm = \varepsilon$$

$$(xm\ \oplus\ ym) \gg\!= k = (xm \gg\!= k)\ \oplus\ (ym \gg\!= k)$$

The use of algebraicity laws can be found in practical applications. For example, in the context of monadic parsers [**25**], the zero law above allows some parsers to be simplified, while the distributivity law improves the efficiency of some others.

4.2.1. State

The presentation of state typically seen in Haskell is somewhat different to failure and choice, and is worth mentioning here. For simplicity let us

assume only a single memory location that stores a value of type S. Then we have the following algebraic operations for the state monad. A S-ary *lookup* operation $S \to 1$ and a unary *update* operation $1 \to S$ (note that the latter has coarity S), subject to equational axioms [**56**] (for the case $S = 2$ these were given in Section 2.1 in terms of the operations *getk* and *putk*). For an operational intuition of these operations, consider that *lookup* takes S computations and upon reading the state, determines the one to continue with. Similarly, *update* takes a computation and upon writing to the state, continues with this computation.

Notice however the following problem. In general S may be infinite thus the *lookup* operation is potentially infinitary. For this reason, state is typically presented instead in terms of generic effects. When rendered as such, we recover the familiar Haskell operations in the type class MonadState, together with the usual get-put laws (as stated in Section 2.1).

class Monad $m \Rightarrow$ MonadState s m **where**

$get :: m\ s$

$put :: s \longrightarrow m\ ()$

4.2.2. Backtrackable State

Let us now combine the aforementioned two effects together. To simplify matters a little, we will consider state with just the theory of failure without binary choice, which serves to illustrate adequately the construction.

One way to combine the two is to apply the monad transformer StateT to what is essentially the Maybe monad. This yields the type

$$\mathsf{StateT}\ s\ \mathsf{Maybe}\ a\ \simeq\ s \longrightarrow \mathsf{Maybe}\ (a, s)$$

This interaction and other similar variants is sometimes called *backtrackable* state (also *local* state). In some sense, this monad models transactional

operation—if such a computation fails, the state is rolled back (or "backtracked") to how it was prior. This intuition is all the more clearer when one see this in terms of algebraic theories.

Recall that StateT corresponds to the construction $\mathbb{L}_S \otimes -$ that takes the tensor product of the given effect with the theory $\mathbb{L}_S$ of state. In this case, the combined theory has all the operations ε, *get* and *put*, algebraicity and all the get-put laws, and commutativity of ε with both *put* and *get*, i.e.

$$put\ s \gg \varepsilon = \varepsilon$$
$$get \gg \varepsilon = \varepsilon$$

The first of these is particularly telling. Given a starting state s_0, any update s immediately followed by a failure is the same computation as just failing (thus leaving the state s_0 intact). In other words, the update is discarded.

Let us consider now the full backtracking monad, by adding back the choice operation. In terms of monads, we have a composite monad that is essentially StateT applied to the list monad, that is

$$\mathsf{StateT}\ s\ [\,]\ a\ \simeq\ s \longrightarrow [(a, s)]$$

Now the tensor product theory is in addition to that given in the above, the operation *or*, *(or-alg)* and the monoid laws, and the following

$$get \gg\!= \lambda s \longrightarrow xm\ \oplus\ ym = (get \gg\!= \lambda s \longrightarrow xm)\ \oplus\ (get \gg\!= \lambda s \longrightarrow ym)$$
$$put\ s \gg xm\ \oplus\ ym = (put\ s \gg xm)\ \oplus\ (put\ s \gg ym)$$

The equations express that while the choice of computation may be made at the outset, in either case, the state read (or written to) is common to each. If you will, each branch has its own local copy of the state at the start.

4.2.3. Persistent State

Backtrackable state is by no means the only composite effect one obtains by combining state and failure (or choice, for that matter). Reverting to our

simpler backtracking effect, we may instead choose to apply the transformer-equivalent of Maybe to state, to obtain the following type

$$\mathsf{MaybeT}\ (\mathsf{State}\ s)\ a\ \simeq\ s \longrightarrow (\mathsf{Maybe}\ a, s)$$

We call this interaction *perisistent* state (also *local* state, especially when the previous interaction is referred to as global). Here, there are no transactional guarantees on the state, so that a computation that fails will persist any stateful operations prior to the fail. This behaviour is often desired by the programmer—certainly no less so than backtrackable behaviour.

One might observe that MaybeT can be considered a special case $E = 1$ of the exceptions monad transformer ErrorT, with just a single exception. Recalling that ErrorT corresponds to the construction $\mathbb{L}_E + -$ (for $\mathbb{L}_E$ the theory of exceptions), together with the ensuing discussion, we have the following result (as in [**26**]).

Corollary 4.2.3. *The monad of backtrackable state is given by the tensor product, and the monad of global state by the sum.*

This appears to be evident from what has been discussed, albeit our arguments have relied on the knowledge of certain monad transformers—and in particular their characterisation as either a sum or tensor of Lawvere theories. It can be instructive however, to also see a more explicit (and in some sense, more elementary) justification in terms of syntax trees.

Recall that a stateful computation in A can be seen as a syntax tree, where each node is either a *lookup* or an *update* (i.e. an algebraic operation of state) and leaves are values in A. A lookup is an S-ary operation, hence a node with S branches. Similarly, update nodes have just a single branch. In fact, the axioms of state (essentially the get-put laws) imply that any such computation can be reduced to a normal form—that is, a syntax tree comprising a single lookup followed by an update at each branch applied to a value in A. From this normal form, one recovers precisely the usual state monad $(A \times S)^S$.

Now consider a computation of the sum $\mathbb{L}_S + \mathbb{L}_1$ of the state and failure theories. Since ε is nullary, such operations occur at the leaves of the computation. As the sum imposes no further equations, normal forms are similar to before, differing only at the leaves—i.e. a lookup followed by an update at every branch, with each of these updates preceeding either an A or a ε. This gives rise to the monad $((A + 1) \times S)^S$ as required. For the tensor product $\mathbb{L}_S \otimes \mathbb{L}_1$ note that the situation is similar to the sum, except that further identifications are made, namely the equations

$$put\ s \gg \varepsilon = \varepsilon$$
$$get \gg \varepsilon = \varepsilon$$

as stated earlier. The second equation is a special case of a more general result [**56**] that identifies a computation xm with one that has xm at every branch of a lookup

$$get \gg xm = xm$$

Again, however, it is the first equation that is the more telling: an update followed immediately by failure is identified with simply failure, regardless of the update s

$$put\ s \gg \varepsilon = \varepsilon = put\ s' \gg \varepsilon$$

Thus normal forms are lookups followed by *either* an update and then an A, or (by the above equation) simply failure. In other words, we have the monad $((A \times S) + 1)^S$ as required.

4.3. Distributive tensor

We give the abstract definition of the key notion of distributive tensor as in [**27**], followed by a more concrete explanation.

Definition 4.3.1 ([**27**])**.** Let $f : (n + 1) \to 1$ and $g : m \to 1$ be operations in Lawvere theories $\mathbb{T}$ and $\mathbb{S}$ respectively. The *distributive tensor* of $\mathbb{T}$ over $\mathbb{S}$, denoted $\mathbb{T} \triangleright \mathbb{S}$, is the Lawvere theory defined by the universal property of

having maps of Lawvere theories from $\mathbb{T}$ and $\mathbb{S}$ to $\mathbb{T} \triangleright \mathbb{S}$, with commutativity of the diagram

$$\begin{array}{ccc} n+m & \xrightarrow{(f \circ (id_n \times \pi_i))_{i \in m}} & m \\ {\scriptstyle id_n \times g}\downarrow & & \downarrow{\scriptstyle g} \\ n+1 & \xrightarrow[f]{} & 1 \end{array}$$

together with commutativity of all other n variants of the diagram given by varying the choice of an element of $n+1$ in the bottom-left to any of its n predecessors.

It is worth unpacking this definition. $\mathbb{T} \triangleright \mathbb{S}$ is the theory consisting of all operations and equations of both $\mathbb{T}$ and $\mathbb{S}$ (i.e. their sum), together with distributivity of every operation f in $\mathbb{T}$ over every operation g in $\mathbb{S}$. Explicitly, for $(n+1)$-ary f and m-ary g as stated, we have distributivity in each argument of f as shown[3]

$$f(g\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}, x_2, \ldots, x_{n+1}) = g\begin{pmatrix} f(y_1, x_2, \ldots, x_{n+1}) \\ \vdots \\ f(y_m, x_2, \ldots, x_{n+1}) \end{pmatrix} \qquad \text{Eq. 1}$$

$$\vdots \qquad\qquad \vdots$$

$$f(x_1, \ldots, x_n, g\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}) = g\begin{pmatrix} f(x_1, \ldots, x_n, y_1) \\ \vdots \\ f(x_1, \ldots, x_n, y_m) \end{pmatrix} \qquad \text{Eq. } (n+1)$$

Eq. 1 states that f distributes over g in the first argument of f. For example, in the simple (but typical in examples) case $n = 1$, we have multiplication distributing over addition in the first argument

$$(y_1 + y_2) \times x = (y_1 \times x) + (y_2 \times x)$$

[3]We write the arguments of g vertically merely to aid readability

which we also refer to by saying that multiplication distributes over addition *leftwards*. Of course, we also have distributivity in the other position (or *rightwards*)

$$x \times (y_1 + y_2) = (x \times y_1) + (x \times y_2)$$

So in general, maximal distributivity entails as many such equations as there are arguments of f. Note in particular that Eq. $(n + 1)$ corresponds to the particular instance of the commutative diagram above, while Eqs. 1 to n correspond to other variants of the diagram. It is also worth pointing out that distributivity is a separate notion to commutativity (Definition 4.1.1), which is a stronger assertion. For example, multiplication generally does not commute with addition

$$(x_1 \times x_2) + (y_1 \times y_2) \neq (x_1 + y_1) \times (x_2 + y_2)$$

By definition, nullary operations (if they exist in $\mathbb{T}$) cannot distribute. We write $\mathbb{T} \lhd\rhd \mathbb{S}$ to denote a *bidirectional* distributive tensor, that is their sum with distributivity equations from both $\mathbb{T} \rhd \mathbb{S}$ and $\mathbb{S} \rhd \mathbb{T}$. We will see examples of effects combined by distributive tensor both here and in chapters that follow.

Remark 4.3.2. In the study of computational effects it can be useful to generalise Lawvere theories along two dimensions; to different arities e.g. countably infinite sets, and adding V-enrichment to homs (notably in the category $V = \omega CPO$ or similar, to allow modelling of recursion). This leads to general countable Lawvere V-theories. A *discrete* Lawvere V-theory [**27**] is a mild restriction of this notion that guarantees the existence of distributive tensors in this setting. For our purposes, we work with ordinary (finitary) Lawvere theories unless otherwise stated, where existence of distributive tensors can be safely assumed.

Example 4.3.3. We denote the following to be a theory of nondeterministic choice consisting of a single binary operation required to be symmetric,

idempotent and associative

$$\textsc{nondet} \triangleq (\{\sqcap\}, \text{semilattice laws})$$

In the domain of two-player games and communication protocols, one often finds a system that may be seen very naturally as a combined theory of two interacting (nondeterministic) choices. This interaction is typically bidirectional distributivity i.e. $\textsc{nondet} \lhd\rhd \textsc{nondet}$. The signature of this theory is the disjoint union of NONDET with itself, which we distinguish with $\sqcap$ and $\sqcap'$. Thus we have $\sqcap$ distributing over $\sqcap'$

$$x \sqcap (y \sqcap' z) = (x \sqcap y) \sqcap' (x \sqcap z)$$
$$(x \sqcap' y) \sqcap z = (x \sqcap z) \sqcap' (y \sqcap z)$$

and also similarly, $\sqcap'$ distributing over $\sqcap$. Chapter 6 discusses in some detail the distributive tensor of *probabilistic* choice over nondeterministic choice.

4.3.1. Distributive laws and semirings

Let Mon and Ab denote the free monoid and abelian group monads, respectively. The canonical example of a distributive law of monads [**8**] is that of Mon over Ab, giving rise to the monad of free rings. Let us simplify this example further slightly (in particular, we will not concern ourselves with group inverse structure), making it relevant to data types in programming. Let us declare a Haskell type class to specify monadic types equipped with monoidal structure[4]

```
class Monad m ⇒ MonadProd m where
  (∗) :: m a ⟶ m a ⟶ m a
  1   :: m a
```

[4]Essentially the same as the standard MonadPlus type class in Haskell, but our choice of naming for this example is more suggestive.

with the implicit requirements that $*$ is associative, with 1 as its unit.

$$x * (y * z) = (x * y) * z$$

$$x * 1 = x$$

$$1 * x = x$$

Furthermore, let us declare a type class of monads with *commutative monoid* structure

class MonadSum $m \Rightarrow$ MonadSum m **where**

$(+) :: m\ a \longrightarrow m\ a \longrightarrow m\ a$

$0 \quad :: m\ a$

with analogous laws to MonadProd ($+$ is associative with 0 its unit) as well as

$$x + y = y + x$$

Now it is well known that the list data type, together with list append and empty list, forms the canonical instance of MonadProd

instance MonadProd $[a]$ **where**

$(*) = (+\!\!+)$

$1 \quad = [\,]$

and similarly, the type of *bags* is the canonical instance of MonadSum, with 0 given by the empty bag and $+$ by bag union. There is a distributive law of lists over bags given by "multiplying out", for example

$$(a + b) * (c + d) \longmapsto (a * c) + (a * d) + (b * c) + (b * d)$$

Informally, given a list of bags, compute every possible list that can be constructed by choosing an element from each bag, with the ith element of each list coming from the ith bag.

This gives rise to "bag of lists" as a composite monad. Indeed, this is precisely the monad of free *semirings*, in which multiplication distributes

over addition. Axiomatically, as well as both the above specifications which we could express

class (MonadSum m, MonadProd m) $\Rightarrow$ MonadSemiring m **where**

we also have additional interaction laws

$$x * (y + z) = (x * y) + (x * z)$$

$$(x + y) * z = (x * z) + (y * z)$$

$$x * 0 = 0$$

$$0 * x = 0$$

In fact, the theory of semirings is precisely the distributive tensor of monoids over commutative monoids—notice that distributivity of multiplication over 0 corresponds to annihilation laws. Thus the semiring (or "bag of lists") monad is an example of combined effect arising from both a distributive law and a (coinciding) distributive tensor.

4.3.2. Splitting the monoid

From the above example, it is clear that there is much common ground between the two notions of distributive law (of monads) and distributive tensor (of theories). Given that a semiring may be "broken down" as an appropriate combination of monoid and commutative monoid, one might wonder about the extent of which such decompositions can be made generally (at least, among the most standard) of algebraic theories. To distil this further, let us consider the theory of monoids in more detail. At first sight, it may appear that a monoid is so simple a structure that it cannot be further separated. But note that the following fact has long been known (e.g. at least since [**70**])

Proposition 4.3.4. *Let* $\mathbb{U}$ *be a theory comprising a binary operation* $*$ *with unit* 1. *The commutative tensor* $\mathbb{U} \otimes \mathbb{U}$ *is the theory of a symmetric monoid* $(*, 1)$.

PROOF. By the Eckmann-Hilton argument, see Proposition 3.1.8. □

That is, starting with a unital (but not necessarily associative) operation, imposing commutativity with itself gives the required associativity—and indeed along with that, symmetry.

However, it would seem (equally, if not more) natural to assume a monoid be assembled from some suitable combination of a semigroup $\mathbb{S} = (*)$ with a constant 1—in contrast to the above, in this situation we *start* with associativity and look to constructing the unit laws

$$x * 1 = x = 1 * x$$

But the "usual" combinations on theories cannot hope to do this—the commutative tensor $\mathbb{S} \otimes 1$ is only able to express that 1 is its own unit

$$1 * 1 = 1$$

while the distributive tensor $\mathbb{S} \triangleright 1$, as per the semiring example, yields annihilation laws

$$x * 1 = 1 = 1 * x$$

thus adorning the semigroup $*$ with *zero* structure 1. While this is clearly different, we shall return to this theory shortly.

In the case of monoids, it turns out that a formulation in terms of monads is a more fruitful approach. Free semigroups give rise to a data type of non-empty lists[5], and a constant operation amounts to exactly the Maybe monad (where Nothing represents the constant). There is a distributive law of NonEmpty over Maybe, given by the following Haskell program [6]

```
monoid :: NonEmpty (Maybe a) ⟶ Maybe (NonEmpty a)
monoid = nonEmpty ∘ map fromJust ∘ N.filter isJust
```

[5]As of GHC 8.0.1, base has a NonEmpty list data type in Data.List.NonEmpty

[6]Since there is potential for name-clashing of functions for the usual list data type with analogous functions for NonEmpty, one typically imports the latter qualified. In particular, *filter* and *map* are imported *as* N to disambiguate from the usual ones.

Given a non-empty list of values (possibly containing Nothings), the idea is that N.*filter isJust* discards every occurrence of Nothing, and then we simply yield the resulting non-empty list. If we started with a (non-empty) list consisting only of Nothings, the function returns Nothing. This map is known to be a distributive law (e.g. it appears as an example in [**13**]). Indeed, the composite monad Maybe ∘ NonEmpty it gives rise to is precisely the free monoid monad—i.e. the usual list monad.

But this is not by any means the only distributive law of NonEmpty over Maybe. Consider the following function

$$
\begin{aligned}
&\mathit{semigroupZero} :: \mathsf{NonEmpty}\ (\mathsf{Maybe}\ a) \longrightarrow \mathsf{Maybe}\ (\mathsf{NonEmpty}\ a)\\
&\mathit{semigroupZero}\ \mathit{ms}\\
&\quad \mid \mathit{all}\ \mathit{isJust}\ \mathit{ms} = \mathsf{Just}\ (\mathsf{N}.\mathit{map}\ \mathit{fromJust}\ \mathit{ms})\\
&\quad \mid \mathit{otherwise}\quad\ = \mathsf{Nothing}
\end{aligned}
$$

Unlike the *monoid* function, *semigroupZero* is of a somewhat more "destructive" nature—it returns Nothing if there is at least one Nothing in the input list *ms*. It turns out this map is also a distributive law, thus giving rise to a distinct composite monad Maybe ∘ NonEmpty to the monoid above. Furthermore, this coincides with the distributive tensor $\mathbb{S} \triangleright 1$ earlier, of semigroups over a constant. The annihilation behaviour is clear in the monad structure, specifically in the monad multiplication (equivalently, $\gg=$) where for a given list of lists *xss*, its concatenation (or "flattening") is restricted to the case where each sublist in *xss* is non-empty, otherwise collapsing to empty. Informally,

---

import Data.Maybe (*fromJust*, *isJust*)

import Data.List.NonEmpty (NonEmpty (..), *nonEmpty*)

import *qualified* Data.List.NonEmpty *as* N (*filter*, *map*)

$$
\begin{aligned}
&\mathit{mult} :: [[a]] \longrightarrow [a] \\
&\mathit{mult}\ \mathit{xss} \\
&\quad \mid \mathit{any}\ \mathit{null}\ \mathit{xss} = [\,] \\
&\quad \mid \mathit{otherwise} \quad\ = \mathit{concat}\ \mathit{xss}
\end{aligned}
$$

This "canonically composite" monad, coinciding with $\mathbb{S} \triangleright 1$, can therefore be seen as the free *semigroup with zero* monad.

The foregoing examples, while simple, illustrate an obvious but as yet still not entirely clear connection between distributive laws and tensors. Indeed, while distributive tensors always exist (subject to Remark 4.3.2), a distributive law of the corresponding monads may not (e.g. probabilistic and nondeterministic choice—see Chapter 6). But when they do, a canonical choice [**42, 43**], is given by distributive tensor. It remains future work to elucidate precisely this relationship.

Part 2

Applications

CHAPTER 5

Searching with monoids

In this applications chapter, we consider several monadic types representing various combinatorial search strategies, such as depth-first and breadth-first search. When viewed in terms of algebraic specifications, it turns out they can can be characterised neatly in terms of sum and distributive tensors of theories.

Sections 5.1 and 5.2 discuss the background material [**62**] on search strategies as characterised by bunch monads. In particular, Section 5.1 starts by introducing the more familiar backtracking (MonadPlus) monads, as those having monoidal operations. We stress the importance of the equational laws—not only of the monoid laws, but algebraicity of those operations. For example, the list monad is backtracking, but Maybe is not, for its monoid operation acts as a "catch" exception handler known (and which we confirm) to be non-algebraic. Indeed, the list monad supports backtracking with a *depth-first* traversal. In Section 5.1 this is stated more precisely, as a monad morphism from a *forest*-shaped structure of the search space. In fact, a key insight in [**62**] is that other search strategies such as breadth-first and depth-bounded traversal can be formalised in an analogous fashion (in terms of the monadic types Matrix and DBound, respectively), if backtracking monads are adorned with an operation *wrap* to account for the "cost" of traversing depth-wise into a search forest. A backtracking monad with this structure is a bunch monad.

We outline the contributions in Section 5.3, which contain a number of correspondence results based on the following observation: that the essential structure of the various search strategies mentioned above can be

captured by adding appropriate equations to the theory of monoids MON with a unary operation WRAP, and considering its normal forms. As such, we show that the monad of forests arises from taking the sum of MON and WRAP i.e. without adding new equations. We give two proofs of this—firstly by considering the free models of the sum-theory, from which it quickly follows that the traversals considered thus far are well-behaved monad morphisms, in the sense that bunch structure is preserved. The other proof has a more computational flavour, by application of a theorem about the generalised resumptions monad transformer. While the list monad (characterising depth-first traversal) arises by discarding WRAP, other bunch types are shown to satisfy equations given by *adding* interaction between the two theories. In the case of DBound values, these are shown to be models of a distributive tensor of MON over WRAP. We consider also the free models of this theory, giving rise to a monad of *fences*. Furthermore, by imposing symmetry to the monoid operation, we obtain a distributive tensor theory that matches breadth-first traversal closely, and while we show that Matrix does not satisfy all requirements of being a model of this theory, we propose a refinement of it that is. Again we consider the free models, which gives rise to a monad of *bundles*. Compared with DBound and Matrix—both no doubt defined with practicality of programming in mind, the new bunch monads for fences and bundles should be understood as encapsulating the essential structure of depth-bounded and breadth-first traversals, respectively.

In Section 5.4, the focus shifts back to list computations, where the (`mtl`) list monad transformer is critiqued with respect to its application to practical stream programming. The main contribution in this section is a novel characterisation of this transformer as a distributive tensor from MON. This is in contrast to the equational presentation of the alternative, "done right" list transformer, which we clarify. While it too exhibits distributivity of the monoid operation, it does so crucially only in the leftwards direction and not the right.

5.1. Backtracking monads

The ability to search for *multiple* solutions is a distinctive feature of logic programming (e.g. in languages such as Prolog), and one that is not an obviously immediate feature of functional programming. However, it has long been observed [**68**] that list computations provide this facility. So in a lazy functional language like Haskell, a Kleisli map $a \to [b]$ is a function that returns multiple values of type β, or the empty list in case of no solution. Note that such list computations are not necessarily finite—laziness is crucial here, to deliver solutions "as you go along", rather than the whole list at the end of a computation (which would of course be hopeless in the infinite case)

Recall that a nonempty (lazy) list comprises a pair of data, namely a value at the head, together with the tail—an as-yet-unevaluated list computation which may be thought of as the *choices* that the computation can "backtrack" to, to explore next. Indeed, the list monad is the canonical example of a *backtracking monad*. Such monads are often associated with instances of the MonadPlus type class in Haskell.

```
class Monad m ⇒ MonadPlus m where
  ε  :: m a
  (⊕) :: m a ⟶ m a ⟶ m a
```

That is, instances of the Monad type class equipped with a monoidal structure—a binary operation $\oplus$ and a nullary ε, with the expectation that $\oplus$ be associative with ε as a unit. The idea is that ε represents a computation that failed to find any solutions, and $\oplus$ combines alternatives together. In the particular example of the list monad

```
instance MonadPlus [] where
  ε  = []
  (⊕) = (++)
```

and of course, $+\!\!+$ is indeed associative with $[\,]$ as unit. Accordingly, there is a monad transformer ListT that adds backtracking search as an effect. The equational theory of ListT will duly be discussed in Section 5.4.

5.1.1. Algebraicity and backtracking

The Maybe monad is another instance of MonadPlus. In the following, the $\oplus$ operation may be thought of as a simple form of exception handling.

```
instance MonadPlus Maybe where
  ε                = Nothing
  Just x    ⊕ _    = Just x
  Nothing   ⊕ ym   = ym
```

It is easy to see that $\oplus$ and ε for Maybe do indeed satisfy the monoid laws. Despite this however, the definition of $\oplus$ makes it clear that Maybe computations do not backtrack! Consider a MonadPlus computation

$$return\ 1\ \oplus\ return\ 2\ \oplus\ \ldots\ \gg\!= \lambda n \longrightarrow\ \ldots$$

In the list monad, each choice in $[1, 2, \ldots]$ is explored in turn. But in the Maybe monad, the computation to the left of the $\gg\!=$ becomes

$$\begin{aligned} &\mathsf{Just}\ 1\ \oplus\ \mathsf{Just}\ 2\ \oplus\ \ldots \\ =\ &\mathsf{Just}\ 1 \end{aligned}$$

by definition of $\oplus$. In other words, only the first choice is explored—the rest are discarded.

Another way to see this is by considering the equational laws more carefully. Usually when equipping operations to monads, it is a reasonable requirement for such operations to be *algebraic* [**58**], in the sense that $\gg\!=$ distributes leftwards over them. Specifically for MonadPlus, this amounts to

the following (family of) equations

$$\varepsilon \gg\!= f = \varepsilon \tag{5.1}$$

$$(xm \oplus ym) \gg\!= f = (xm \gg\!= f) \oplus (ym \gg\!= f) \tag{5.2}$$

for every Kleisli map f.

Remark 5.1.1. Ignoring the monoid laws of MonadPlus temporarily, the algebraicity equations (such as (5.1) and (5.2) above) states a property common to all *free* monads. Namely, a term $xm \gg\!= f$ given by applying a *substitution* f to the variables of xm, preserves the essential structure of xm.

In fact, a simple counterexample shows that $\oplus$ *is not algebraic for the Maybe monad*, despite being a (seemingly bona fide) instance of MonadPlus. For consider the map

$f\ 1 = \mathsf{Just}\ ()$
$f\ 0 = \mathsf{Nothing}$

Then we have

$$\begin{aligned}(\mathsf{Just}\ 0 \oplus \mathsf{Just}\ 1) \gg\!= f &= \mathsf{Just}\ 0 \gg\!= f \\ &= \mathsf{Nothing} \\ &\neq\ \mathsf{Just}\ () \\ &= \mathsf{Nothing} \oplus \mathsf{Just}\ () \\ &= (\mathsf{Just}\ 0 \gg\!= f) \oplus (\mathsf{Just}\ 1 \gg\!= f)\end{aligned}$$

This gives an intuitive characterisation of algebraicity of $\oplus$ for MonadPlus monads—namely, ability to backtrack. While list computations possess this, we have seen that Maybe does not. Indeed, its multiplication $\oplus$ should be seen instead as a form of exception handling. More generally, such non-algebraic *handlers* [**5**] are discussed briefly in Appendix A .

5.1.2. Searching in the list monad

List computations (and more generally, MonadPlus computations *algebraic* in the above sense) contain the structure necessary for backtracking search. Consider the following program that searches for factors[1].

$$
\begin{aligned}
&\mathit{factor} :: \mathsf{Int} \longrightarrow [(\mathsf{Int}, \mathsf{Int})] \\
&\mathit{factor}\ n = [(x, y) \mid x \leftarrow [1\,..\,n], \\
&\qquad\qquad\qquad\qquad y \leftarrow [1\,..\,n], \\
&\qquad\qquad\qquad\qquad x * y \equiv n]
\end{aligned}
$$

Remark 5.1.2. For convenience this program uses Haskell's list comprehensions, but the point is that it could equally well be written purely in terms of $\oplus$, ε, $\gg\!=$ and *return*, with the benefit of being applicable to any MonadPlus instance—not just for lists. For example, the guard can be expressed in terms of the function

$$
\begin{aligned}
&\mathit{test} :: \mathsf{MonadPlus}\ m \Rightarrow \mathsf{Bool} \longrightarrow m\ () \\
&\mathit{test}\ p = \textbf{if}\ p\ \textbf{then}\ \mathit{return}\ ()\ \textbf{else}\ \varepsilon
\end{aligned}
$$

and the generators more abstractly in terms of the function

$$
\begin{aligned}
&\mathit{choices} :: \mathsf{MonadPlus}\ m \Rightarrow [a] \longrightarrow m\ a \\
&\mathit{choices} = \mathit{foldr}\ (\oplus)\ \varepsilon \circ \mathit{map}\ \mathit{return}
\end{aligned}
$$

Then the program can be written (using Haskell's **do**-notation for succinctness)

$$
\begin{aligned}
&\mathit{factor} :: \mathsf{MonadPlus}\ m \Rightarrow \mathsf{Int} \longrightarrow m\ (\mathsf{Int}, \mathsf{Int}) \\
&\mathit{factor}\ n = \textbf{do} \\
&\quad x \leftarrow \mathit{choices}\ [1\,..\,n] \\
&\quad y \leftarrow \mathit{choices}\ [1\,..\,n]
\end{aligned}
$$

[1]The bounds $[1..n]$ can of course be tighter, but this is not a crucial point here.

$$
\begin{array}{l}
\mathit{test}\ (x * y \equiv n) \\
\mathit{return}\ (x, y)
\end{array}
$$

In this program, the nature of the search becomes apparent if we modify it slightly so that the choices over the y variable are unbounded

$$
\begin{array}{lll}
\mathit{factor}'\ n = [(x, y) \mid & x \leftarrow [1\,..\,n], \\
& y \leftarrow [1\,..], \\
& x * y \equiv n]
\end{array}
$$

Now, running this on 12 say, computes the first solution $(1, 12)$ but $(2, 6)$ and all others will never be reached as x is stuck at 1 while y increases indefinitely. That is to say, choices in list computations are traversed in *depth-first* order (in contrast to say, the "fairer" diagonalising list comprehensions of the language Miranda [**66**]).

5.2. Search strategies

We can view the search space of a list computation as a tree—or more generally, a *forest* of possible solutions. We consider a type of forests, as finite lists of trees.

$$
\begin{array}{l}
\textbf{type}\ \mathsf{Forest}\ a = [\mathsf{Tree}\ a] \quad \text{-- finite lists} \\
\textbf{data}\ \mathsf{Tree}\ a \quad = \mathsf{Leaf}\ a \\
\qquad\qquad\quad \mid\ \mathsf{Fork}\ (\mathsf{Forest}\ a)
\end{array}
$$

Thus a forest is either empty [], or a leaf with a (possibly empty) sibling subforest, or a tree with a (possibly empty) sibling subforest. It is not difficult to check that forests have monadic structure[2].

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Monad}\ \mathsf{Forest}\ \textbf{where} \\
\quad \mathit{return}\ x \qquad\qquad = [\mathsf{Leaf}\ x]
\end{array}
$$

[2]Haskell does not allow **type** aliases to be **instance** definitions—a restriction we ignore here, for convenience of exposition.

$$
\begin{aligned}
[\,] \gg\!= \_ &= [\,] \\
(\mathsf{Leaf}\ x : ts) \gg\!= f &= f\ x +\!\!+ ts \gg\!= f \\
(\mathsf{Fork}\ ts : us) \gg\!= f &= (\mathsf{Fork}\ ts \gg\!= f) : us \gg\!= f
\end{aligned}
$$

They also have backtracking structure, similar to that of lists.

instance MonadPlus Forest **where**

$$
\begin{aligned}
\varepsilon &= [\,] \\
(\oplus) &= (+\!\!+)
\end{aligned}
$$

Then we see explicitly depth-first traversal as a function on forests.

$$
\begin{aligned}
&dft :: \mathsf{Forest}\ a \longrightarrow [a] \\
&dft\ [\,] = [\,] \\
&dft\ (\mathsf{Leaf}\ x : ts) = x : dft\ ts \\
&dft\ (\mathsf{Fork}\ ts : us) = dft\ ts +\!\!+ dft\ us
\end{aligned}
$$

In fact, this function is a MonadPlus morphism. First it is a monad morphism, meaning that monadic structure is preserved

$$dft \circ return = return \tag{5.3}$$

$$dft\ (ts \gg\!= f) = dft\ ts \gg\!= dft \circ f \tag{5.4}$$

and it is a MonadPlus morphism in that backtracking structure is preserved

$$dft\ \varepsilon = \varepsilon \tag{5.5}$$

$$dft\ (ts \oplus us) = dft\ ts \oplus dft\ us \tag{5.6}$$

Notice in particular equations (5.4) and (5.6) exhibit *dft* as an example of a "compositional" traversal, in the sense that traversing a composite forest is the same as adjoining together the traversals of the subforests.

In many cases however, depth-first search is undesirable (such as when the search space is infinite, as in the factoring example) and another search strategy may be better-suited e.g. breadth-first search. A natural question

is whether other such search strategies exhibit this desirable compositional quality.

5.2.1. Breadth-first search

As discussed above, in a list computation, successive elements represent answers searched in depth-first order. Now consider a somewhat more elaborate type that matches more closely the forest structure of the search space, that of lists of (finite) bags (assuming an appropriate Bag type). We will call such a structure a *matrix*. The idea is that successive bags in a matrix represent solutions found at successive levels of the forest.

$$\textbf{type}\ \mathsf{Matrix}\ a = [\mathsf{Bag}\ a]$$

Note that this is a different composite data type to the one considered in Section 4.3.1, of *bags of lists*. Before discussing matrices further, let us fix notation and suitable combinators on the Bag type[3]. We use bag literal notation $\wr x, y, ... \int$ to denote bags of elements. We will also assume the existence of the following functions—bag union, concatenation, map and fold—each with the obvious intended meaning.

$$\begin{aligned}
(\uplus) \quad &:: \mathsf{Bag}\ a \longrightarrow \mathsf{Bag}\ a \longrightarrow \mathsf{Bag}\ a \\
\mathit{bagconcat} \quad &:: \mathsf{Bag}\ (\mathsf{Bag}\ a) \longrightarrow \mathsf{Bag}\ a \\
\mathit{bagmap} \quad &:: (a \longrightarrow b) \longrightarrow \mathsf{Bag}\ a \longrightarrow \mathsf{Bag}\ b \\
\mathit{bagfold} \quad &:: (a \longrightarrow a \longrightarrow a) \longrightarrow a \longrightarrow \mathsf{Bag}\ a \longrightarrow a
\end{aligned}$$

where the first argument to *bagfold* should be a *commutative* binary operator, so that the order in which elements are folded is not significant. Now

[3]It is not uncommon in Haskell to represent bags using the standard list data type, e.g. **type** Bag $a = [a]$, and indeed we may assume this implementation in what follows with little harm. While convenient however, we emphasise the (mathematical) requirement for bags, since we would like Matrix to form a monad (which lists of *bags* is), but it is widely conjectured that lists of *lists* fails this requirement.

it can be shown [**62**] that matrices form backtracking monads, with the instance definitions that follow. First, matrices have backtracking structure

```
instance MonadPlus Matrix where
  ε     = []
  (⊕) = lzw (⊎)
```

where *lzw* is a "long" variant of the *zipWith* function, somewhat similar to the standard *zipWith* except that any remainder from the argument lists is preserved rather than discarded (hence the necessity for the type of the argument function to be $a \longrightarrow a \longrightarrow a$, rather than the more general $a \longrightarrow b \longrightarrow c$). Its role is to "lift" bag union from Bag a to [Bag a].

```
lzw :: (a ⟶ a ⟶ a) ⟶ [a] ⟶ [a] ⟶ [a]
lzw _ [] ys             = ys
lzw _ xs []             = xs
lzw f (x : xs) (y : ys) = f x y : lzw f xs ys
```

These can in turn be used to define the monadic structure of matrices

```
instance Monad Matrix where
  return x      = [ ⟅ x ⟆ ]
  [] >>= _      = []
  (b : bs) >>= f = join (bagmap f b) ⊕ ⟅ ⟆ :(bs >>= f)
    where
      join = bagfold (⊕) ε
```

In fact, we may refine the definition of $\gg\!=$ a little further, when the function $⟅ ⟆ :\_$ will be called *wrap* for reasons that will become clear in Section 5.2.3. Crucially, we will also see that a matrix represents a *breadth-first* traversal of a forest.

5.2.2. Depth-bounded search

Depth-bounded search is similar to depth-first search, except that the search is made finite by traversing no deeper into the search-forest than a given depth bound. Motivated by this, we employ a type of stateful functions, threading a bound through computations. Answers are found, tagged with the remaining depth bound[4].

```
type DBound a = Int ⟶ [(a, Int)]
              (≅ StateT Int [] a)
```

Similar in nature to certain parser types [**25**], this "backtrackable state" type is isomorphic to a particular application of the StateT transformer. As such, it forms a monad in the expected way.

```
instance Monad DBound where
  return x n  = [(x, n)]
  (p >>= f) n = [(y, s) | (x, r) ← p n,
                          (y, s) ← f x r]
```

And indeed, it is another example of a backtracking monad.

```
instance MonadPlus DBound where
  ε _           = []
  (p1 ⊕ p2) n = p1 n ++ p2 n
```

As with matrices, we shall see in Section 5.2.3 that DBound functions represent *depth-bounded* traversals of forests.

5.2.3. Bunch monads

We have seen that a list represents a depth-first traversal of a forest, given by a MonadPlus morphism. Can this be generalised to other types of search? We address this question in this section. First, we introduce one

[4]As an analogy, one might imagine a scuba diver at work in the sea in search of oceanic artifacts, but with limited (and depleting) oxygen in the tank.

further piece of structure that will become vital in keeping track of branching levels.

Definition 5.2.1. A backtracking monad (i.e. a MonadPlus instance as described above) is a *bunch monad* if it additionally supports an operation *wrap*. In Haskell, this can be expressed as a type class.

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

As with equations (5.1) and (5.2), *wrap* is expected to be an algebraic operation

$$wrap\ xm \gg\!= f = wrap\ (xm \gg\!= f) \tag{5.7}$$

Notice there are no additional equational laws imposed, other than the pre-existing monoid laws of MonadPlus. We will discuss equational laws in more detail in Section 5.3. For now, we note that all the monads described so far are bunch instances, as discussed in the following. Intuitively, we think of *wrap*ping a forest *ts* into a tree, by bringing together the trees *ts* under a new Fork.

```
instance Bunch Forest where
  wrap ts = [Fork ts]
```

With this in mind, for each of the other monads we ask how augmenting a forest in this way affects its traversal. For example, a depth-first traversal of *wrap ts* gives the same list of answers as that of *ts*—the extra fork makes no difference to (depth-first) traversal. This gives a very simple definition of *wrap* for the Bunch instance of list computations

```
instance Bunch [] where
  wrap = id   -- answers unchanged by wrapping!
```

Of course in general, the presence of the extra fork does affect traversals, and indeed this what the *wrap* operation tries to capture. In particular, if

bs is a Matrix resulting from a breadth-first traversal of a forest, wrapping the forest into a tree introduces a new level to search from, corresponding to the root of the tree. But since it is the root, there are of course no answers to be found there, represented by the empty bag. And since the structure of the rest of the tree remains the same (it is, after all, exactly the original unwrapped forest), its breadth-first traversal is still bs—each bag is merely "shifted down" a level.

```
instance Bunch Matrix where
  wrap bs = ⟅ ∫ :bs
```

The intuition for a depth-bounded traversal of a wrapped forest is very similar. Again, there are no answers to be found at the new root node, and answers within depth $n > 0$ are simply answers within depth $n - 1$ of the original forest.

```
instance Bunch DBound where
  wrap p = q
    where
      q 0 = [ ]
      q n = p (n − 1)
```

Equipped with bunch structure, it is then possible to express each search strategy as a generic function.

```
traversal :: Bunch m ⇒ Forest a ⟶ m a
traversal [ ]              = ε
traversal (Leaf x : ts)    = return x ⊕ traversal ts
traversal (Fork ts : us)   = wrap (traversal ts) ⊕ traversal us
```

By specialising *traversal* to bunch monads [ ], Matrix and DBound we respectively obtain depth-first, breadth-first and depth-bounded traversals. The

crucial role that *wrap* plays in enabling this is in respecting forks, i.e. distinguishing between a traversal of a minimal Tree (i.e. a Leaf), and one of a composite Tree (i.e. a Fork).

5.3. Combinations of theories

We have seen several examples of backtracking search strategy, or bunches. One may ask what equational laws each of these satisfy, in addition to the monoid laws of MonadPlus. We will see that such additional laws can be viewed as *interactions* between the backtracking operations and *wrap*.

In the following, denote WRAP for the signature comprising solely the operation *wrap*, and MON the theory of monoids.

$$\text{WRAP} \triangleq \{\mathit{wrap}\}$$

$$\text{MON} \triangleq (\{\varepsilon, \oplus\}, \text{monoid laws})$$

It immediately follows that the sum of theories WRAP + MON is what we also call the theory of bunches—namely, adjoining *wrap* to the theory of monoids with no additional equations. If we think of forests as the starting point of an "empty" search strategy, the following correspondence (indicating the lack of any extra interactions) should be of little surprise.

Proposition 5.3.1. *The monad* Forest *is presented by the theory of bunches, that is the sum of theories* WRAP + MON.

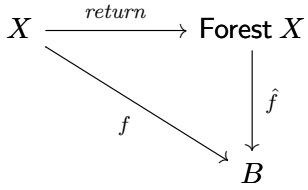
Before offering a proof, let us first recall some standard facts about algebraic theories and models, which should also serve to clarify the claim. Let $\mathbb{T}$ be an algebraic theory, and denote $\mathsf{Mod}(\mathbb{T})$ its category of models and homomorphisms (over the category **Set**). There is a forgetful functor $\mathsf{Mod}(\mathbb{T}) \to \mathbf{Set}$ that discards the interpretations of the operations, keeping only the carrier. This functor is monadic, so that its left adjoint—which for any set X, constructs the free $\mathbb{T}$-model over X—induces a monad $\mathscr{T}$ on **Set** such that its algebras are equivalent to the $\mathbb{T}$-models, i.e. $\mathbf{Set}^{\mathscr{T}} \simeq \mathsf{Mod}(\mathbb{T})$.

Thus to establish such an equivalence between theory and monad in Proposition 5.3.1, amounts to showing that Forest indeed forms the free models of WRAP + MON.

PROOF. Let X be a set, and $(B, \oplus, \varepsilon, \mathit{wrap})$ an arbitrary model of the theory. In particular we have already seen in Section 5.2 that forests form models of the form $(\mathsf{Forest}\ X, +\!\!+, [\,], [\mathsf{Fork}-])$. Now for any function $f : X \to B$, we seek a unique homomorphism between these models

$$\hat{f} : (\mathsf{Forest}\ X, +\!\!+, [\,], [\mathsf{Fork}-]) \longrightarrow (B, \oplus, \varepsilon, \mathit{wrap})$$

such that the following triangle commutes



(5.8)

where *return* is the unit of the monad Forest as defined in Section 5.2.

Take $\hat{f}$ to be the following function

$$\begin{aligned}
\hat{f}\ [\,] &= \varepsilon \\
\hat{f}\ (\mathsf{Leaf}\ x : ts) &= f\ x \oplus \hat{f}\ ts \\
\hat{f}\ (\mathsf{Fork}\ ts : us) &= \mathit{wrap}\ (\hat{f}\ ts) \oplus \hat{f}\ us
\end{aligned}$$

To check that $\hat{f}$ is a homomorphism, first note from the first equation

$$\hat{f}\ [\,] = \varepsilon$$

that the unit is preserved. Similarly by taking us to be empty $[\,]$, the third equation implies that *wrap* is preserved.

$$\hat{f}\ [\mathsf{Fork}\ ts] = \mathit{wrap}\ (\hat{f}\ ts)$$

Finally, to see that multiplication is preserved, a straightforward induction on the argument xs in $xs +\!\!+ us$ suffices. For the base case $xs \equiv [\,]$

$$\hat{f}\ ([\,] +\!\!+ us) = \hat{f}\ us = \varepsilon \oplus \hat{f}\ us = \hat{f}\ [\,] \oplus \hat{f}\ us$$

For the case $xs \equiv \mathsf{Leaf}\ x : ts$,

$$\begin{aligned}
\hat{f}\ (\mathsf{Leaf}\ x : ts +\!\!+ us) &= f\ x \oplus \hat{f}\ (ts +\!\!+ us) && \text{Defn } \hat{f} \\
&= f\ x \oplus \hat{f}\ ts \oplus \hat{f}\ us && \text{IH} \\
&= \hat{f}\ (\mathsf{Leaf}\ x : ts) \oplus \hat{f}\ us && \text{Defn } \hat{f}
\end{aligned}$$

Finally for $xs \equiv \mathsf{Fork}\ ts : vs$,

$$\begin{aligned}
\hat{f}\ (\mathsf{Fork}\ ts : vs +\!\!+ us) &= wrap\ (\hat{f}\ ts) \oplus \hat{f}\ (vs +\!\!+ us) && \text{Defn } \hat{f} \\
&= wrap\ (\hat{f}\ ts) \oplus \hat{f}\ vs \oplus \hat{f}\ us && \text{IH} \\
&= \hat{f}\ (\mathsf{Fork}\ ts : vs) \oplus \hat{f}\ us && \text{Defn } \hat{f}
\end{aligned}$$

Hence by induction, $\hat{f}\ (xs +\!\!+ us) = \hat{f}\ xs \oplus \hat{f}\ us$.

That the triangle commutes is then the simple matter of a diagram-chase

$$\hat{f}\ (return\ x) = \hat{f}\ [\mathsf{Leaf}\ x] = f\ x$$

Lastly, for uniqueness let $h : \mathsf{Forest}\ X \longrightarrow B$ be an arbitrary homomorphism of models. Then by necessity,

$$h\ [\,] = \varepsilon \tag{5.9}$$

$$h\ [\mathsf{Fork}\ ts] = wrap\ (h\ ts) \tag{5.10}$$

$$h\ (ts +\!\!+ us) = h\ ts \oplus h\ us \tag{5.11}$$

Furthermore, assume that h (in place of $\hat{f}$) also makes the triangle commute

$$h\ [\mathsf{Leaf}\ x] = f\ x \tag{5.12}$$

Notice that equation (5.9) agrees with the definition of $\hat{f}$. Let us derive the rest of the definition of h by reasoning equationally as follows

$$\begin{aligned}
h\ (\mathsf{Leaf}\ x : ts) &= h\ ([\mathsf{Leaf}\ x] +\!\!+ ts) \\
&= h\ [\mathsf{Leaf}\ x] \oplus h\ ts && \text{by (5.11)} \\
&= f\ x \oplus h\ ts && \text{by (5.12)}
\end{aligned}$$

$$\begin{aligned} h\ (\mathsf{Fork}\ ts : us) &= h\ ([\mathsf{Fork}\ ts] +\!\!+ us) \\ &= h\ [\mathsf{Fork}\ ts] \oplus h\ us && \text{by (5.11)} \\ &= wrap\ (h\ ts) \oplus h\ us && \text{by (5.10)} \end{aligned}$$

to conclude $h = \hat{f}$. □

Corollary 5.3.2. *The function traversal (Section 5.2.3) is a morphism of bunch monads.*

Proof. In the proof of Proposition 5.3.1, take f to be the unit (or *return*) of an arbitrary bunch monad. Then its unique extension $\hat{f}$ is exactly the function *traversal*. Since it is a homomorphism of models, that bunch structure is preserved is immediate. To see that it is a monad morphism, notice that the commutative triangle (5.8) says exactly that *return* is preserved. It remains to check that $\gg\!=$ is preserved, that is

$$traversal\ (ts \gg\!= f) = traversal\ ts \gg\!= traversal \circ f \tag{5.13}$$

For $ts \equiv [\,]$, we have

$$\begin{aligned} traversal\ ([\,] \gg\!= f) &= traversal\ [\,] && \text{Defn } \gg\!= \\ &= \varepsilon && \text{Defn } traversal \\ &= \varepsilon \gg\!= traversal \circ f && \text{algebraicity (5.1)} \\ &= traversal\ [\,] \gg\!= traversal \circ f && \text{Defn } traversal \end{aligned}$$

For $ts \equiv \mathsf{Leaf}\ x : us$,

$$\begin{aligned} &traversal\ ((\mathsf{Leaf}\ x : us) \gg\!= f) \\ &= traversal\ (f\ x +\!\!+ us \gg\!= f) && \text{Defn } \gg\!= \\ &= traversal\ (f\ x) \oplus traversal\ (us \gg\!= f) && \text{homomorphism} \\ &= (return\ x \gg\!= traversal \circ f) \oplus (traversal\ us \gg\!= traversal \circ f) && \text{IH} \\ &= return\ x \oplus traversal\ us \gg\!= traversal \circ f && \text{algebraicity (5.2)} \\ &= traversal\ (\mathsf{Leaf}\ x : us) \gg\!= traversal \circ f && \text{Defn } traversal \end{aligned}$$

For $ts \equiv \mathsf{Fork}\ us : vs$,

$$
\begin{aligned}
& \mathit{traversal}\ ((\mathsf{Fork}\ us : vs) \gg\!= f) \\
&= \mathit{traversal}\ (\mathsf{Fork}\ (us \gg\!= f) : vs \gg\!= f) && \text{Defn } \gg\!= \\
&= \mathit{wrap}\ (\mathit{traversal}\ (us \gg\!= f)) \oplus \mathit{traversal}\ (vs \gg\!= f) && \text{Defn } \mathit{traversal} \\
&= \mathit{wrap}\ (\mathit{traversal}\ us \gg\!= \mathit{traversal} \circ f) \\
&\quad \oplus\ (\mathit{traversal}\ vs \gg\!= \mathit{traversal} \circ f) && \text{IH } \times 2 \\
&= (\mathit{wrap}\ (\mathit{traversal}\ us) \gg\!= \mathit{traversal} \circ f) \\
&\quad \oplus\ (\mathit{traversal}\ vs \gg\!= \mathit{traversal} \circ f) && \text{algebraicity (5.7)} \\
&= \mathit{wrap}\ (\mathit{traversal}\ us) \oplus \mathit{traversal}\ vs \gg\!= \mathit{traversal} \circ f && \text{algebraicity (5.2)} \\
&= \mathit{traversal}\ (\mathsf{Fork}\ us : vs) \gg\!= \mathit{traversal} \circ f && \text{Defn } \mathit{traversal}
\end{aligned}
$$

Hence by induction, equation (5.13) holds. □

The proof of Proposition 5.3.1 can of course be adapted for any algebraic theory—it does not, after all, rely on the theory being a sum (for example). In fact, WRAP + MON is a sum of a very particular form, namely of a theory with a *signature*. With this observation, we may draw upon a result that gives an alternative, shorter proof that offers a different computational narrative on the matter. We explore this next.

5.3.1. Forests as Resumptions Monads

Cenciarelli and Moggi [**11**] introduced the generalised resumptions monad transformer, which has also come to be known by various other guises, such as the "coroutine" transformer and the FreeT transformer (especially among Haskell programmers). The name FreeT is closely related to the construction Free in Haskell

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Free}\ \Sigma\ a = \mathsf{Var}\ a \\
&\qquad\qquad\quad \mid\ \mathsf{Comp}\ (\Sigma\ (\mathsf{Free}\ \Sigma\ a))
\end{aligned}
$$

The type constructor Free Σ forms a Monad for any Σ that is a Functor—indeed as the name suggests, it is the free monad on Σ, in the categorical sense.

```
instance Functor Σ ⇒ Monad (Free Σ) where
  return          = Var
  (Var x) >>= f   = f x
  (Comp y) >>= f  = Comp (fmap (>>=f) y)
```

The transformer FreeT Σ on the other hand, sends a monad t to the monad $\mu z.t(- + \Sigma z)$, or more verbosely

```
type FreeT Σ t a = t (FreeF Σ a (FreeT Σ t a))
data FreeF Σ a z = V a
                 | C (Σ z)
```

As is usual for a monad transformer, there is a MonadTrans instance defining how underlying t-computations are lifted into the "larger" monad

```
instance MonadTrans (FreeT Σ) where
  lift :: Monad t ⇒ t a ⟶ FreeT Σ t a
  lift = liftM V
```

As expected, one may recover the monad Free Σ by instantiating the transformer to the identity monad, i.e. FreeT Σ Id.

The following theorem is due to Hyland, Plotkin and Power [**26**].

Theorem 5.3.3. *The generalised resumptions monad* FreeT Σ t *is given by the coproduct of monads* Free $\Sigma + t$. *Equivalently, by considering* Σ *as a signature and* t *presented by a theory* $\mathbb{T}$, *we have a sum of theories* $\Sigma + \mathbb{T}$.

Thus an alternative proof of Proposition 5.3.1 is given by specialising the theorem, as recorded in the following.

Corollary 5.3.4. *The* Forest *monad is given by the coproduct of monads* Free Id + [] *(where* Id *is the identity functor,* [] *the list monad), equivalently the sum of theories* WRAP + MON *(c.f. Proposition 5.3.1).*

PROOF. By definition, the monad of resumptions on lists is the type

$$\mathsf{FreeT}\ \mathsf{Id}\ []\ a \cong [\mathsf{FreeF}\ \mathsf{Id}\ a\ (\mathsf{FreeT}\ \mathsf{Id}\ []\ a)]$$

That this is isomorphic to Forest a is easily seen by the following (invertible) function

$$\begin{aligned}
&\mathit{forest2resum} :: \mathsf{Forest}\ a \longrightarrow [\mathsf{FreeF}\ \mathsf{Id}\ a\ (\mathsf{FreeT}\ \mathsf{Id}\ []\ a)] \\
&\mathit{forest2resum}\ [] &&= [] \\
&\mathit{forest2resum}\ (\mathsf{Leaf}\ x : ts) &&= \mathsf{V}\ x : \mathit{forest2resum}\ ts \\
&\mathit{forest2resum}\ (\mathsf{Fork}\ ts : us) &&= \mathsf{C}\ ts : \mathit{forest2resum}\ us
\end{aligned}$$

Hence by Theorem 5.3.3, the monad Forest may be decomposed as a coproduct Free Id + [].

Finally, since the Id functor describes precisely a signature consisting of a sole unary operation (= WRAP), and MON presents the list monad, this coproduct is equivalently a sum of theories WRAP + MON. □

To interpret this result in the context of computation, it is insightful to view resumptions computations in the FreeT Id t monad as *trampoline* computations (in reference to [**15**]), so-called because each "bounce" of the computation—that is, each uninterrupted run of t-effects—may be interspersed with pauses. Indeed, resumptions computations can be seen as a form of coroutine. The idea is that at a pause, control is yielded from the computation, possibly to be resumed later.

With a suitable combinator on trampolines, it becomes possible to interleave such computations (possibly of different underlying effects) together concurrently. Note however that coroutines of this form do not facilitate communication between each other. While they can be run concurrently, they are essentially independent of each other. Under this view, we see that

Forest computations can be seen as list (i.e. backtracking search) computations, interspersed with pauses or "delays".

Example 5.3.5. As another application of Theorem 5.3.3, consider effectful strategy trees [**34**]. These turn out be to a more powerful form of coroutine, supporting a request-response protocol of communication.

5.3.2. Revisiting depth-bounded

In [**62**] (Section 5), it is demonstrated that by adding appropriate equations to WRAP + MON, Forest terms are quotiented into normal forms closely resembling the various bunch types (Matrix, DBound) defined earlier. In what follows, we consider these equations carefully, seeking precise characterisations in terms of combining theories.

Consider a typical Forest term, to be used in examples that follow.

$$\begin{array}{l}[\mathsf{Fork}\ [\mathsf{Leaf}\ 1, \mathsf{Leaf}\ 2, \mathsf{Leaf}\ 3],\\ \ \mathsf{Leaf}\ 4,\\ \ \mathsf{Fork}\ [\mathsf{Fork}\ [\mathsf{Leaf}\ 5, \mathsf{Leaf}\ 6],\\ \qquad \mathsf{Leaf}\ 7]]\end{array}$$

By adding to the theory of bunches distributivity of *wrap* over $\oplus$

$$wrap\ (ts \oplus us) = wrap\ ts \oplus wrap\ us \tag{5.14}$$

our example term (in a more compact, suggestive notation) becomes identified with one that has its branching structure "forgotten", as though the branches have been pulled apart.

$$\begin{array}{c} w(1 \oplus 2 \oplus 3) \oplus 4 \oplus w(w(5 \oplus 6) \oplus 7) \\ \Big\| \ {\scriptstyle (5.14)} \\ w(1) \oplus w(2) \oplus w(3) \oplus 4 \oplus w^2(5) \oplus w^2(6) \oplus w(7) \end{array} \tag{5.15}$$

The term shows the left-to-right ordering of the solutions, as well as the depth at which each is found, suggesting a correspondence with the DBound

type of functions defined earlier. Indeed, we may represent this term as follows (e.g. by applying *traversal* from earlier to the example term)

$$
\begin{aligned}
&p :: \mathsf{DBound\ Int} \\
&p\ 0 = [(4,0)] \\
&p\ 1 = [(1,0),(2,0),(3,0),(4,1),(7,0)] \\
&p\ 2 = [(1,1),(2,1),(3,1),(4,2),(5,0),(6,0),(7,1)] \\
&p\ n = [(1,n-1),(2,n-1),(3,n-1),(4,n),(5,n-2),(6,n-2),(7,n-1)]
\end{aligned}
$$

To complement equation (5.14), it would appear entirely reasonable to also have distributivity of *wrap* over ε

$$wrap\ \varepsilon = \varepsilon \tag{5.16}$$

Remark 5.3.6. While equation (5.16) is not stipulated in [**62**], accepting it leads to a slightly simpler theory that conveniently quotients away empty trees. To illustrate, let us augment our example term with the tree $\mathsf{Fork}\ [\mathsf{Fork}\ [\,], \mathsf{Fork}\ [\,]]$ (underlined in the below).

$$
\begin{array}{c}
w(1 \oplus 2 \oplus 3) \oplus 4 \oplus w(w(5 \oplus 6) \oplus 7) \oplus \underline{w(w(\varepsilon) \oplus w(\varepsilon))} \\
\Big\| \\
w(1) \oplus w(2) \oplus w(3) \oplus 4 \oplus w^2(5) \oplus w^2(6) \oplus w(7)
\end{array}
$$

By both of the equations (5.14) and (5.16), this is "multiplied out" to the same normalised term as before i.e. the underlined subterm disappears. *Without* equation (5.16) however, one finds in addition a proliferation of empty trees $w^2(\varepsilon) \oplus w^2(\varepsilon)$.

Theorem 5.3.7. *The theories* WRAP, MON, *together with equations* (5.14) *and* (5.16) *give a distributive tensor* WRAP $\triangleright$ MON.

The following confirms that there is indeed a correspondence between this theory and DBound.

Proposition 5.3.8. DBound *computations form a model of the theory* WRAP▷ MON.

PROOF. That DBound satisfies the monoid laws is clear, since it is a MonadPlus instance (Section 5.2.2). In addition,

$$wrap\ \varepsilon = wrap\ (\lambda n \longrightarrow [\,]) = (\lambda n \longrightarrow [\,]) = \varepsilon$$

and since

$$\begin{aligned} wrap\ (p \oplus q)\ 0 &= [\,] \\ &= wrap\ p\ 0 +\!\!+ wrap\ q\ 0 \\ &= (wrap\ p \oplus wrap\ q)\ 0 \end{aligned}$$

$$\begin{aligned} wrap\ (p \oplus q)\ n &= p\ (n-1) +\!\!+ q\ (n-1) \\ &= wrap\ p\ n +\!\!+ wrap\ q\ n \\ &= (wrap\ p \oplus wrap\ q)\ n \end{aligned}$$

we have $wrap\ (p \oplus q) = wrap\ p \oplus wrap\ q$. □

5.3.3. Fences as Free Models

While DBound computations form a model of the theory WRAP ▷ MON, they are not the *free* models, and therefore we do not have an equivalence between the theory and the *monad* DBound (at least, not in the same way as we did for Forest in Proposition 5.3.1). Indeed, the normal form in (5.15) suggests a certain Fence structure

```
type Fence a = [Stick a]
data Stick a = Tip a
             | Segment (Stick a)
```

Notice that while p encodes extra stateful information, (5.15) has a much more direct representation as a Fence term

```
[Segment (Tip 1),
 Segment (Tip 2),
 Segment (Tip 3),
 Tip 4,
 Segment (Segment (Tip 5)),
 Segment (Segment (Tip 6)),
 Segment (Tip 7)]
```

Fence has the following monadic structure

```
instance Monad Fence where
  return x              = [Tip x]
  [] >>= _              = []
  (Tip x : ss) >>= f    = f x ++ ss >>= f
  (Segment s : ss) >>= f = map Segment ([s] >>= f) ++ ss >>= f
```

It has, respectively, MonadPlus and Bunch structure

```
instance MonadPlus Fence where
  ε     = []
  (⊕) = (++)

instance Bunch Fence where
  wrap = map Segment
```

To see that the above definitions are sensible, let us begin by checking that the operations $\oplus$, ε and *wrap* defined above are indeed algebraic as required.

Lemma 5.3.9. *The following equations hold*

$$[] \gg\!= f = [] \tag{5.17}$$

$$ss +\!\!+ ts \gg\!= f = ss \gg\!= f +\!\!+ ts \gg\!= f \tag{5.18}$$

$$map\ \mathsf{Segment}\ ss \gg\!= f = map\ \mathsf{Segment}\ (ss \gg\!= f) \tag{5.19}$$

PROOF. (5.17) is immediate, by definition of $\gg\!=$.

For (5.18), an induction on ss suffices. We show the case $ss \equiv \mathsf{Segment}\ s : ss'$ (the other cases are similar, and are no different to the analogous proof for Forest).

$$
\begin{aligned}
&\mathsf{Segment}\ s : ss' +\!\!+ ts \gg\!= f \\
&= map\ \mathsf{Segment}\ ([s] \gg\!= f) +\!\!+ (ss' +\!\!+ ts) \gg\!= f && \text{Defn } \gg\!= \\
&= map\ \mathsf{Segment}\ ([s] \gg\!= f) +\!\!+ ss' \gg\!= f +\!\!+ ts \gg\!= f && \text{IH} \\
&= \mathsf{Segment}\ s : ss' \gg\!= f +\!\!+ ts \gg\!= f && \text{Defn } \gg\!=
\end{aligned}
$$

For (5.19) we again proceed by induction on ss. For $ss \equiv [\,]$, both sides of

$$map\ \mathsf{Segment}\ [\,] \gg\!= f = map\ \mathsf{Segment}\ ([\,] \gg\!= f)$$

equate to $[\,]$, by definition of $\gg\!=$ and the fact that $map\ \_\ [\,] = [\,]$. For $ss \equiv \mathsf{Tip}\ x : ss'$,

$$
\begin{aligned}
&map\ \mathsf{Segment}\ (\mathsf{Tip}\ x : ss') \gg\!= f \\
&= \mathsf{Segment}\ (\mathsf{Tip}\ x) : map\ \mathsf{Segment}\ ss' \gg\!= f && \text{Property } map \\
&= map\ \mathsf{Segment}\ ([\mathsf{Tip}\ x] \gg\!= f) +\!\!+ map\ \mathsf{Segment}\ ss' \gg\!= f && \text{Defn } \gg\!= \\
&= map\ \mathsf{Segment}\ ([\mathsf{Tip}\ x] \gg\!= f) +\!\!+ map\ \mathsf{Segment}\ (ss' \gg\!= f) && \text{IH} \\
&= map\ \mathsf{Segment}\ ([\mathsf{Tip}\ x] \gg\!= f +\!\!+ ss' \gg\!= f) && \text{Property } map \\
&= map\ \mathsf{Segment}\ ([\mathsf{Tip}\ x] +\!\!+ ss' \gg\!= f) = map\ \mathsf{Segment}\ (\mathsf{Tip}\ x : ss' \gg\!= f) && \text{Eq (5.18)}
\end{aligned}
$$

Finally for $ss \equiv \mathsf{Segment}\ s : ss'$,

$$
\begin{aligned}
& map\ \mathsf{Segment}\ (\mathsf{Segment}\ s : ss') \gg\!= f \\
&= \mathsf{Segment}\ (\mathsf{Segment}\ s) : map\ \mathsf{Segment}\ ss' \gg\!= f && \text{Property } map \\
&= map\ \mathsf{Segment}\ ([\mathsf{Segment}\ s] \gg\!= f) +\!\!+ map\ \mathsf{Segment}\ ss' \gg\!= f && \text{Defn } \gg\!= \\
&= map\ \mathsf{Segment}\ ([\mathsf{Segment}\ s] \gg\!= f) +\!\!+ map\ \mathsf{Segment}\ (ss' \gg\!= f) && \text{IH} \\
&= map\ \mathsf{Segment}\ ([\mathsf{Segment}\ s] \gg\!= f +\!\!+ ss' \gg\!= f) && \text{Property } map \\
&= map\ \mathsf{Segment}\ ([\mathsf{Segment}\ s] +\!\!+ ss' \gg\!= f) && \text{Eq (5.18)} \\
&= map\ \mathsf{Segment}\ (\mathsf{Segment}\ s : ss' \gg\!= f)
\end{aligned}
$$

□

Let us further establish two special cases of the monad laws, which shall prove useful in the respective laws of Proposition 5.3.11.

Lemma 5.3.10. *The following equations hold*

$$[s] \gg\!= return = [s] \tag{5.20}$$

$$([s] \gg\!= f) \gg\!= g = [s] \gg\!= \lambda x \longrightarrow f\ x \gg\!= g \tag{5.21}$$

Proof. By induction on s. For (5.20), case $s \equiv \mathsf{Tip}\ x$,

$$[\mathsf{Tip}\ x] \gg\!= return = return\ x = [\mathsf{Tip}\ x]$$

by the monadic definitions. For $s \equiv \mathsf{Segment}\ s'$,

$$
\begin{aligned}
[\mathsf{Segment}\ s'] \gg\!= return &= map\ \mathsf{Segment}\ ([s'''] \gg\!= return) && \text{Defn } \gg\!= \\
&= map\ \mathsf{Segment}\ [s'] && \text{IH} \\
&= [\mathsf{Segment}\ s']
\end{aligned}
$$

For (5.21), case $s \equiv \mathsf{Tip}\ x$,

$$\begin{aligned}
([\mathsf{Tip}\ x] \gg\!= f) \gg\!= g &= f\ x \gg\!= g && \text{Defn } \gg\!= \\
&= \mathit{return}\ x \gg\!= \lambda y \longrightarrow f\ y \gg\!= g && \text{Eq (5.22)} \\
&= [\mathsf{Tip}\ x] \gg\!= \lambda y \longrightarrow f\ y \gg\!= g && \text{Defn } \mathit{return}
\end{aligned}$$

For $s \equiv \mathsf{Segment}\ s'$,

$$\begin{aligned}
([\mathsf{Segment}\ s'] \gg\!= f) \gg\!= g &= \mathit{map}\ \mathsf{Segment}\ ([s'] \gg\!= f) \gg\!= g && \text{Defn } \gg\!= \\
&= \mathit{map}\ \mathsf{Segment}\ (([s'] \gg\!= f) \gg\!= g) && \text{Eq (5.19)} \\
&= \mathit{map}\ \mathsf{Segment}\ ([s'] \gg\!= \lambda x \longrightarrow f\ x \gg\!= g) && \text{IH} \\
&= \mathit{map}\ \mathsf{Segment}\ [s'] \gg\!= \lambda x \longrightarrow f\ x \gg\!= g && \text{Eq (5.19)} \\
&= [\mathsf{Segment}\ s'] \gg\!= \lambda x \longrightarrow f\ x \gg\!= g
\end{aligned}$$

□

Proposition 5.3.11. *The monad laws hold*

$$\mathit{return}\ x \gg\!= f = f\ x \tag{5.22}$$

$$xm \gg\!= \mathit{return} = xm \tag{5.23}$$

$$(xm \gg\!= f) \gg\!= g = xm \gg\!= \lambda x \longrightarrow f\ x \gg\!= g \tag{5.24}$$

Proof. For (5.22), we have by definition

$$[\mathsf{Tip}\ x] \gg\!= f = f\ x$$

as required.

For (5.23), we proceed by induction on xm. For $xm \equiv [\,]$, by definition

$$[\,] \gg\!= \mathit{return} = [\,]$$

For $xm \equiv \mathsf{Tip}\ x : ss$,

$$\begin{aligned}
&\mathsf{Tip}\ x : ss \gg\!= return = return\ x +\!\!+ ss \gg\!= return && \text{Defn } \gg\!= \\
&\qquad = [\mathsf{Tip}\ x] +\!\!+ ss && \text{Defn } return\text{, IH} \\
&\qquad = \mathsf{Tip}\ x : ss
\end{aligned}$$

For $xm \equiv \mathsf{Segment}\ s : ss$,

$$\begin{aligned}
&\mathsf{Segment}\ s : ss \gg\!= return \\
&= map\ \mathsf{Segment}\ ([s] \gg\!= return) +\!\!+ ss \gg\!= return && \text{Defn } \gg\!= \\
&= map\ \mathsf{Segment}\ [s] +\!\!+ ss && \text{Eq (5.20), IH} \\
&= [\mathsf{Segment}\ s] +\!\!+ ss = \mathsf{Segment}\ s : ss
\end{aligned}$$

Finally for (5.24), we proceed by induction on xm. For $xm \equiv [\,]$, by definition of $\gg\!=$ both sides of

$$([\,] \gg\!= f) \gg\!= g = [\,] \gg\!= \lambda x \longrightarrow f\ x \gg\!= g$$

equate to $[\,]$. For $xm \equiv \mathsf{Tip}\ x : ss$,

$$\begin{aligned}
&(\mathsf{Tip}\ x : ss \gg\!= f) \gg\!= g \\
&= (f\ x +\!\!+ ss \gg\!= f) \gg\!= g && \text{Defn } \gg\!= \\
&= f\ x \gg\!= g +\!\!+ (ss \gg\!= f) \gg\!= g && \text{Eq (5.18)} \\
&= return\ x \gg\!= \lambda y \longrightarrow f\ y \gg\!= g +\!\!+ ss \gg\!= \lambda y \longrightarrow f\ y \gg\!= g && \text{Eq (5.22), IH} \\
&= (return\ x +\!\!+ ss) \gg\!= \lambda y \longrightarrow f\ y \gg\!= g && \text{Eq (5.18)} \\
&= \mathsf{Tip}\ x : ss \gg\!= \lambda y \longrightarrow f\ y \gg\!= g && \text{Defn } return
\end{aligned}$$

For $xm \equiv \mathsf{Segment}\ s : ss$,

$$
\begin{aligned}
&(\mathsf{Segment}\ s : ss \gg\!= f) \gg\!= g \\
&= (map\ \mathsf{Segment}\ ([s] \gg\!= f) +\!\!+ ss \gg\!= f) \gg\!= g && \text{Defn } \gg\!= \\
&= map\ \mathsf{Segment}\ ([s] \gg\!= f) \gg\!= g +\!\!+ (ss \gg\!= f) \gg\!= g && \text{Eq (5.18)} \\
&= map\ \mathsf{Segment}\ (([s] \gg\!= f) \gg\!= g) +\!\!+ ss \gg\!= \lambda x \longrightarrow f\ x \gg\!= g && \text{Eq (5.19), IH} \\
&= map\ \mathsf{Segment}\ ([s] \gg\!= \lambda x \longrightarrow f\ x \gg\!= g) +\!\!+ ss \gg\!= \lambda x \longrightarrow f\ x \gg\!= g && \text{Eq (5.21)} \\
&= map\ \mathsf{Segment}\ [s] \gg\!= \lambda x \longrightarrow f\ x \gg\!= g +\!\!+ ss \gg\!= \lambda x \longrightarrow f\ x \gg\!= g && \text{Eq (5.19)} \\
&= [\mathsf{Segment}\ s] \gg\!= \lambda x \longrightarrow f\ x \gg\!= g +\!\!+ ss \gg\!= \lambda x \longrightarrow f\ x \gg\!= g \\
&= ([\mathsf{Segment}\ s] +\!\!+ ss) \gg\!= \lambda x \longrightarrow f\ x \gg\!= g && \text{Eq (5.18)} \\
&= \mathsf{Segment}\ s : ss \gg\!= \lambda x \longrightarrow f\ x \gg\!= g
\end{aligned}
$$

□

Recall the distributive tensor theory in Theorem 5.3.7. Let us check that Fence computations—together with the definitions of $\oplus$, ε and *wrap*—form its models, before establishing that they are indeed the free such models.

Proposition 5.3.12. (Fence $a, +\!\!+, [\,], map$ Segment) *is a model of the theory* WRAP $\triangleright$ MON*, for any* a*.*

Proof. $(+\!\!+, [\,])$ form a monoid as usual, so it remains to check the distributivity laws (5.16) and (5.14). And indeed,

$$map\ \mathsf{Segment}\ [\,] = [\,]$$

$$map\ \mathsf{Segment}\ (ss +\!\!+ ss') = map\ \mathsf{Segment}\ ss +\!\!+ map\ \mathsf{Segment}\ ss'$$

by properties of *map*. □

Theorem 5.3.13. *The monad* Fence *is presented by the theory* WRAP $\triangleright$ MON*. That is, their algebras and models (respectively) are equivalent to each other*

$$\mathsf{Fence}\text{-}\mathbf{Alg} \simeq \mathsf{Mod}(\text{WRAP} \triangleright \text{MON})$$

PROOF. Let $(b, \oplus, \varepsilon, \mathit{wrap})$ be an arbitrary model. By Proposition 5.3.12, for any a, $(\mathsf{Fence}\ a, +\!\!+, [\,], \mathit{map}\ \mathsf{Segment})$ is one such. For any function $f :: a \longrightarrow b$, we seek a unique homomorphism

$$\hat{f} :: (\mathsf{Fence}\ a, +\!\!+, [\,], \mathit{map}\ \mathsf{Segment}) \longrightarrow (b, \oplus, \varepsilon, \mathit{wrap})$$

such that

$$f = \hat{f} \circ \mathit{return} \tag{5.25}$$

where as usual, $\mathit{return} :: a \longrightarrow \mathsf{Fence}\ a$ is the unit as defined earlier.

To this end, define $\hat{f}$ to be the following

$$\begin{aligned}
&\hat{f}\ [\,] &&= \varepsilon \\
&\hat{f}\ (\mathsf{Tip}\ x : \mathit{ss}) &&= f\ x \oplus \hat{f}\ \mathit{ss} \\
&\hat{f}\ (\mathsf{Segment}\ s : \mathit{ss}) &&= \mathit{wrap}\ (\hat{f}\ [s]) \oplus \hat{f}\ \mathit{ss}
\end{aligned}$$

Clearly (5.25) is satisfied by this definition

$$\hat{f}\ (\mathit{return}\ x) = \hat{f}\ [\mathsf{Tip}\ x] = f\ x$$

To see that it defines a homomorphism, let us check that the operations are preserved, that is

$$\hat{f}\ [\,] = \varepsilon \tag{5.26}$$

$$\hat{f}\ (\mathit{ss} +\!\!+ \mathit{ss}') = \hat{f}\ \mathit{ss} \oplus \hat{f}\ \mathit{ss}' \tag{5.27}$$

$$\hat{f} \circ \mathit{map}\ \mathsf{Segment} = \mathit{wrap} \circ \hat{f} \tag{5.28}$$

(5.26) is immediate. For (5.27), an induction on ss suffices. The cases $\mathit{ss} \equiv [\,]$ and $\mathit{ss} \equiv \mathsf{Tip}\ \_ : \_$ are similar to the analogous cases in Proposition 5.3.1, so we merely exhibit the case $\mathit{ss} \equiv \mathsf{Segment}\ s : \mathit{ts}$.

$$\begin{aligned}
\hat{f}\ (\mathsf{Segment}\ s : \mathit{ts} +\!\!+ \mathit{ss}') &= \mathit{wrap}\ (\hat{f}\ [s]) \oplus \hat{f}\ (\mathit{ts} +\!\!+ \mathit{ss}') && \text{Defn } \hat{f} \\
&= \mathit{wrap}\ (\hat{f}\ [s]) \oplus \hat{f}\ \mathit{ts} \oplus \hat{f}\ \mathit{ss}') && \text{IH} \\
&= \hat{f}\ (\mathsf{Segment}\ s : \mathit{ts}) \oplus \hat{f}\ \mathit{ss}' && \text{Defn } \hat{f}
\end{aligned}$$

Finally, (5.28) is again by induction. For the case $[\,]$, both sides of

$$\hat{f}\ (map\ \mathsf{Segment}\ [\,]) = wrap\ (\hat{f}\ [\,])$$

equate to ε, by definition of $\hat{f}$ and distributivity of $wrap$ over ε. For the case $\mathsf{Tip}\ x : ss$,

$$\begin{aligned}
&\hat{f}\ (map\ \mathsf{Segment}\ (\mathsf{Tip}\ x : ss)) \\
&= \hat{f}\ (\mathsf{Segment}\ (\mathsf{Tip}\ x) : map\ \mathsf{Segment}\ ss) \\
&= wrap\ (\hat{f}\ [\mathsf{Tip}\ x]) \oplus \hat{f}\ (map\ \mathsf{Segment}\ ss) && \text{Defn } \hat{f} \\
&= wrap\ (f\ x) \oplus wrap\ (\hat{f}\ ss) && \text{Defn } \hat{f}\text{, IH} \\
&= wrap\ (f\ x \oplus \hat{f}\ ss) && \text{Eq (5.14)} \\
&= wrap\ (\hat{f}\ (\mathsf{Tip}\ x : ss)) && \text{Defn } \hat{f}
\end{aligned}$$

For the case $\mathsf{Segment}\ s : ss$,

$$\begin{aligned}
&\hat{f}\ (map\ \mathsf{Segment}\ (\mathsf{Segment}\ s : ss)) \\
&= \hat{f}\ (\mathsf{Segment}\ (\mathsf{Segment}\ s) : map\ \mathsf{Segment}\ ss) \\
&= wrap\ (\hat{f}\ [\mathsf{Segment}\ s]) \oplus \hat{f}\ (map\ \mathsf{Segment}\ ss) && \text{Defn } \hat{f} \\
&= wrap\ (wrap\ (\hat{f}\ [s])) \oplus wrap\ (\hat{f}\ ss) && \text{Defn } \hat{f}\text{, IH} \\
&= wrap\ (wrap\ (\hat{f}\ [s]) \oplus (\hat{f}\ ss)) && \text{Eq (5.14)} \\
&= wrap\ (\hat{f}\ (\mathsf{Segment}\ s : ss)) && \text{Defn } \hat{f}
\end{aligned}$$

For uniqueness, assume we have $g :: \mathsf{Fence}\ a \longrightarrow b$ with

$$g\ [\,] = \varepsilon \tag{5.29}$$

$$g\ (ss +\!\!+ ts) = g\ ss \oplus g\ ts \tag{5.30}$$

$$g \circ map\ \mathsf{Segment} = wrap \circ g \tag{5.31}$$

$$f = g \circ return \tag{5.32}$$

Then it follows that $g = \hat{f}$—for example,

$$\begin{aligned} g\ (\mathsf{Segment}\ s : ss) &= g\ ([\mathsf{Segment}\ s] +\!\!+ ss) \\ &= g\ [\mathsf{Segment}\ s] \oplus g\ ss && \text{Eq (5.30)} \\ &= g\ (map\ \mathsf{Segment}\ [s] \oplus g\ ss) \\ &= wrap\ (g\ [s]) \oplus g\ ss && \text{Eq (5.31)} \end{aligned}$$

The definition of g in the cases $[\,]$ and $\mathsf{Tip}\ \_ : \_$ are as in the proof of Proposition 5.3.1.

Thus we have established that $(\mathsf{Fence}\ a, +\!\!+, [\,], map\ \mathsf{Segment})$ is the free model of WRAP ▷ MON over a. Put another way, the construction

$$a \longmapsto (\mathsf{Fence}\ a, +\!\!+, [\,], map\ \mathsf{Segment})$$

defines the left adjoint to the evident forgetful functor (that takes a model and keeps only its carrier) which by virtue of its monadicity, implies exactly that the algebras of the induced Fence monad are equivalent to models of the theory, as claimed. □

5.3.4. Revisiting breadth-first

If in addition to equations (5.14) and (5.16) we also require that $\oplus$ is commutative

$$ts \oplus us = us \oplus ts \tag{5.33}$$

we may then conveniently shuffle $w^n(x)$ terms at will, and use equation (5.14) to factor such terms of the same depth together.

$$\begin{array}{c} w(1) \oplus w(2) \oplus w(3) \oplus 4 \oplus w^2(5) \oplus w^2(6) \oplus w(7) \\ \Big\| \quad {\scriptstyle (5.14)+(5.33)} \\ 4 \oplus w(1 \oplus 2 \oplus 3 \oplus 7) \oplus w^2(5 \oplus 6) \end{array}$$

As noted ([**62**]), solutions are grouped at each level of the search tree in an unordered fashion, suggesting a close correspondence with the Matrix

type. Indeed, we may obtain a term *bags* (e.g. by applying *traversal* to the original forest)

$$
\begin{aligned}
&\mathit{bags} :: \mathsf{Matrix\ Int}\\
&\mathit{bags} = [\ \wr\ 4\ \int,\\
&\qquad\quad \wr\ 1,2,3,7\ \int,\\
&\qquad\quad \wr\ 5,6\ \int\ ]
\end{aligned}
$$

Adding equation (5.33) of course gives a theory of *commutative* monoids

$$\textsc{cmon} \triangleq (\{\oplus, \varepsilon\}, \text{commutative monoid laws})$$

As with Theorem 5.3.7, taking all the equations together yields (for breadth-first search) a distributive tensor.

Theorem 5.3.14. *The theories* WRAP, CMON, *together with equations* (5.14) *and* (5.16) *give a distributive tensor* $\textsc{wrap} \triangleright \textsc{cmon}$.

Despite appearances however, there is not a correspondence between this theory and Matrix, for the latter does not form its models. Indeed, the law that breaks is (5.16) distributivity of *wrap* over ε

$$\mathit{wrap}\ \varepsilon = \mathit{wrap}\ [\,] = [\wr\int] \neq [\,] = \varepsilon$$

One approach in repairing this is to consider a modification of the Matrix type to *necessarily infinite* streams of finite bags

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Stream}\ a = [a] \quad \text{-- always-infinite lists}\\
&\textbf{type}\ \mathsf{InfMatrix}\ a = \mathsf{Stream}\ (\mathsf{Bag}\ a)
\end{aligned}
$$

Note that while InfMatrix computations are infinite streams of (finite) bags, we insist on restricting to "finitely supported" such computations—i.e. streams that eventually all end with $\wr\int$, after a finite number of finite bags.

Proposition 5.3.15. InfMatrix *a forms a monad.*

PROOF. InfMatrix a is isomorphic to the type of functions Nat$\longrightarrow$Bag a, for Nat the type of natural numbers. This may also be expressed as ReaderT Nat Bag a, i.e. an instance of the ReaderT transformer

```
type ReaderT r m a = r ⟶ m a
```

which is well-known to form a monad for any Monad m. □

Unlike Matrix, InfMatrix supports a bunch structure that models the distributive tensor theory.

```
instance MonadPlus InfMatrix where
  ε     = repeat ⟅ ⟆
  (⊕) = zipWith (⊎)
```

The ε operation is given by an infinite stream of empty bags, while the $\oplus$ operation is simpler than before—since we no longer need concern ourselves with different-length argument lists (as computations are necessarily of infinite length), the usual *zipWith* suffices for lifting bag union. *wrap* is as before.

```
instance Bunch InfMatrix where
  wrap bs = ⟅ ⟆ :bs
```

Proposition 5.3.16. InfMatrix a *(equipped with the structure above) is a model of the theory* WRAP ▷ CMON*, for any* a*.*

PROOF. InfMatrix a forms a commutative monoid, since for any bs::InfMatrix a,

$$zipWith\ (\uplus)\ bs\ [⟅⟆, ⟅ ⟆ ..] = zipWith\ (\uplus)\ [⟅⟆, ⟅ ⟆ ..]\ bs = bs$$

follows from ⟅⟆ being the unit of $\uplus$. Similarly, *zipWith* preserves associativity and commutativity of $\uplus$.

For distributivity (5.14),

$$\begin{aligned} wrap\ (bs\ \oplus\ bs') &= \wr\,\int : zipWith\ (\uplus)\ bs\ bs' \\ &= (\wr\,\int\ \uplus\ \wr\,\int) : zipWith\ (\uplus)\ bs\ bs' \\ &= zipWith\ (\uplus)\ (\wr\,\int : bs)\ (\wr\,\int : bs') \\ &= wrap\ bs \oplus wrap\ bs' \end{aligned}$$

Crucially, distributivity (5.16) goes through

$$\begin{aligned} wrap\ \varepsilon &= wrap\ [\wr\int, \wr\,\int\,..] \\ &= \wr\,\int : [\wr\int, \wr\,\int\,..] \\ &= [\wr\int, \wr\,\int\,..] \\ &= \varepsilon \end{aligned}$$

□

5.3.5. Bundles as Free Models

As with Section 5.3.3, we consider the monad corresponding to the theory WRAP▷CMON. Again, observing normal forms suggest a similar structure to Fence, except that (due to (5.33)) the ordering of sticks is not significant. This leads to the idea of a "bundle"—a *bag* of sticks.

```
type Bundle a = Bag (Stick a)
```

The MonadPlus and Bunch instances are defined in a completely analogous way to Fence, using bag (rather than list) combinators.

```
instance MonadPlus Bundle where
  ε     = ⟅ ⟆
  (⊕) = (⊎)

instance Bunch Bundle where
  wrap = bagmap Segment
```

The following confirms that the above is well-defined.

Proposition 5.3.17. (Bundle $a, \uplus, \wr\int, \mathit{bagmap}$ Segment) *is a model of the theory* WRAP $\triangleright$ CMON *for any* a.

PROOF. $(\uplus, \wr\int)$ indeed forms a commutative monoid structure on Bundle a, and distributivity holds

$$\mathit{bagmap}\ \mathsf{Segment}\ \ \wr\ \int = \wr\int$$

$$\mathit{bagmap}\ \mathsf{Segment}\ (ss\ \uplus\ ts) = \mathit{bagmap}\ \mathsf{Segment}\ ss\ \uplus\ \mathit{bagmap}\ \mathsf{Segment}\ ts$$

by properties of *bagmap*. □

Its monadic structure is defined in a very similar way to Fence, as follows[5]

```
instance Monad Bundle where
  return x = ⁅ Tip x ⁆
  ss >>= f  = bagconcat (bagmap f' ss)
    where
      f' (Tip x)       = f x
      f' (Segment s) = bagmap Segment (⁅ s ⁆ >>= f)
```

In fact, for the purposes of reusing previous results for the Fence data type, we will find it convenient in the following to consider bundles (i.e. values of type Bundle a) as equivalence classes of fences (values of type Fence a) with respect to permutation. In more detail, we denote $\sim$ to be the following equivalence relation on fences ss, ss'

$$ss \sim ss' \quad \text{iff} \quad ss' = \sigma(ss)$$

for some permutation σ, so that the equivalence class $[ss]$ comprises the (representative) fence ss and all orderings of its sticks.

[5]The differences largely follow from the fact that a bag computation cannot be deconstructed into the patterns empty [] and nonempty _ : _. For the latter in particular, there is no notion of a "first" element.

Remark 5.3.18. Of course, the square bracket notation has already been used for list literals. To avoid the potential for confusion, where necessary in the remainder of this section we use bold square brackets to denote list literals, e.g. $[\![\ \mathsf{Tip}\ 1, \mathsf{Tip}\ 2\ ]\!]$ is the same equivalence class as $[\![\ \mathsf{Tip}\ 2, \mathsf{Tip}\ 1\ ]\!]$.

The equivalence $\sim$ extends to Kleisli maps in the expected way. For maps $f, f' :: a \longrightarrow \mathsf{Fence}\ b$

$$f \sim_{\to} f' \quad \text{iff} \quad f(x) \sim f'(x)$$

for all x in a. Similarly, $\sim$ extends to *lists of fences* in the following way. Let us denote $(xs)_n$ for the nth element of the list xs. Then for $fs, fs' :: [\mathsf{Fence}\ a]$ of the same length,

$$fs \sim_{[]} fs' \quad \text{iff} \quad (fs)_i \sim (fs')_i$$

for each index i over the lists. That is, fs and fs' are element-wise equivalent with respect to $\sim$.

The list functions *map*, *concat* and $(+\!\!+)$ are well-behaved with respect to the equivalence, as recorded in the following lemma.

Lemma 5.3.19. *The equivalence $\sim$ is preserved by map, concat and append, in the following sense.*

$$ss \sim ss' \implies map\ f\ ss \sim map\ f\ ss' \tag{5.34}$$

$$f \sim_{\to} f' \implies map\ f\ ss \sim_{[]} map\ f'\ ss \tag{5.35}$$

$$fs \sim_{[]} fs' \implies concat\ fs \sim concat\ fs' \tag{5.36}$$

$$ss_1 \sim ss_2 \implies ss_1 +\!\!+ ss \sim ss_2 +\!\!+ ss \tag{5.37}$$

and similarly in the other argument of $+\!\!+$.

Proof. For (5.34), let $ss' = \sigma(ss)$. We require $map\ f\ ss' = \sigma'(map\ f\ ss)$ for some σ'. But since *map* preserves the order of its elements,

$$map\ f\ ss' = map\ f\ \sigma(ss) = \sigma(map\ f\ ss)$$

The others follow a similar argument. □

To help disambiguate, in the following we denote $\gg\!=_B$, $return_B$ to be the monadic operations defined above for Bundle, and $\gg\!=_F$, $return_F$ for those defined for Fence in Section 5.3.3.

Proposition 5.3.20. (Bundle, $\gg\!=_B, return_B$) *satisfies the monad laws.*

Proof. From here on we consider bundles as equivalence classes of fences. In particular, rather than take the definition

$$return_B\ x = \wr\ \mathsf{Tip}\ x\ \int$$

as in the earlier (Bag-based) Bundle, we instead define $return_B$ explicitly on equivalence classes of fences, in terms of $return_F$

$$\begin{aligned} return_B\ x &\triangleq [return_F\ x] \\ &= [[\ \mathsf{Tip}\ x\ ]] \end{aligned}$$

Similarly, monadic bind is defined "pointwise"

$$[ss]\ \gg\!=_B\ [f]_{\to} \triangleq [ss\ \gg\!=_F\ f]$$

We check that $\gg\!=_B$ is well-defined in the sense that it respects the equivalence $\sim$ in both positions: for $ss_1 \sim ss_2$ in $[ss]$ and $f_1 \sim_{\to} f_2$ in $[f]_{\to}$

$$ss_1\ \gg\!=_F\ f\ \sim\ ss_2\ \gg\!=_F\ f \tag{5.38}$$

$$ss\ \gg\!=_F\ f_1\ \sim\ ss\ \gg\!=_F\ f_2 \tag{5.39}$$

For relation (5.38) it is helpful to observe that $\gg\!=_F$ can be equally expressed as a *concat* after *map*

$$ss\ \gg\!=_F\ f = concat\ (map\ f'\ ss)$$

for some $f' :: \mathsf{Stick}\ a \longrightarrow \mathsf{Fence}\ a$. Now recall that every stick is of the form $\mathsf{Segment}^n\ (\mathsf{Tip}\ x)$ i.e. some number $n \geqslant 0$ of Segment constructors followed by a Tip. Then f' can be expressed succinctly as the map

$$\mathsf{Segment}^n\ (\mathsf{Tip}\ x) \longmapsto (map\ \mathsf{Segment})^n\ (f\ x)$$

where we use the notation $g^n\ y$ to mean n applications of g to the argument y. Since $ss_1 \sim ss_2$ and the fact that $(map\ f')$ preserves $\sim$ (by Lemma 5.3.19(5.34)), we have

$$map\ f'\ ss_1 \sim_{[]} map\ f'\ ss_2$$

And since *concat* also preserves $\sim$ (Lemma 5.3.19(5.36)),

$$concat\ (map\ f'\ ss_1) \sim concat\ (map\ f'\ ss_2)$$

we indeed have the relation (5.38).

For (5.39) we again consider $\gg\!=_F$ as a *concat-map* above. In particular

$$\begin{aligned}
& f_1 \sim_{\to} f_2 && \\
\Longrightarrow\ & f_1\ x \sim f_2\ x && \text{for any } x \\
\Longrightarrow\ & (map\ \mathsf{Segment})^n\ (f_1\ x) && \\
\sim\ & (map\ \mathsf{Segment})^n\ (f_2\ x) && \text{by (5.34), } n \geqslant 0 \\
\Longrightarrow\ & f_1' \sim_{\to} f_2' && \text{Defn } f' \\
\Longrightarrow\ & map\ f_1'\ ss \sim_{[]} map\ f_2'\ ss && \text{by (5.35)} \\
\Longrightarrow\ & concat\ (map\ f_1')\ ss \sim concat\ (map\ f_2')\ ss && \text{by (5.36)}
\end{aligned}$$

Thus (5.39) holds.

Having established that $\gg\!=_B$ is well-defined, it follows that the monad laws are satisfied since $\gg\!=_B$, $return_B$ are defined in terms of $\gg\!=_F$, $return_F$ respectively, and the latter indeed satisfy the laws by Proposition 5.3.11.

For example, the monadic associativity law

$$\begin{aligned}
&([ss] \gg\!=_B [f]_\to) \gg\!=_B [g]_\to \\
&= [ss \gg\!=_F f] \gg\!=_B [g]_\to \\
&= [(ss \gg\!=_F f) \gg\!=_F g] \\
&= [ss \gg\!=_F (\lambda x \longrightarrow f\ x \gg\!=_F g)] && \text{Prop 5.3.11} \\
&= [ss] \gg\!=_B [\lambda x \longrightarrow f\ x \gg\!=_F g]_\to \\
&= [ss] \gg\!=_B \lambda x \longrightarrow [f\ x \gg\!=_F g] && (*) \\
&= [ss] \gg\!=_B \lambda x \longrightarrow [f\ x] \gg\!=_B [g]_\to
\end{aligned}$$

holds as required, where $(*)$ follows from the equivalence $\sim_\to$ defined over maps into fences: any such class of maps $[f]_\to$ can be understood as a family of ($\sim$-equivalence classes of) fences $[f\ x]$, indexed by x. In other words, a function $\lambda x \longrightarrow [f\ x]$. The monadic unit laws can be shown with similar reasoning. □

Finally, we work towards establishing a correspondence between Bundle and the theory WRAP ▷ CMON. Before that, as we did with $return_B$ and $\gg\!=_B$, we give alternative "pointwise" interpretations of the operations of the theory in terms of Fence-based interpretations (from Section 5.3.3). That is,

$$\begin{aligned}
\varepsilon_B &\triangleq [\varepsilon_F] &&= [[]] \\
[ss] \oplus_B [ss'] &\triangleq [ss \oplus_F ss'] &&= [ss +\!\!+ ss'] \\
wrap_B\ [ss] &\triangleq [wrap_F\ ss] &&= [map\ \mathsf{Segment}\ ss]
\end{aligned}$$

Lemma 5.3.21. *The operations $\varepsilon_B, \oplus_B, wrap_B$ are well-defined, in the sense that they respect the equivalence $\sim$.*

PROOF. ε_B trivially respects $\sim$. For $wrap_B$, let $ss \sim ss'$ be arbitrary (but equivalent) fences. Then

$$\begin{aligned} wrap_F\ ss &= map\ \mathsf{Segment}\ ss \\ &\sim map\ \mathsf{Segment}\ ss' \qquad \text{by Lemma 5.3.19 (5.34)} \\ &= wrap_F\ ss' \end{aligned}$$

For $ss_1 \sim ss_2$ and ${ss'}_1 \sim {ss'}_2$ we need compatibility with $\sim$ in both arguments of $\oplus_F$

$$ss_1 \oplus_F ss' \ \sim\ ss_2 \oplus_F ss' \tag{5.40}$$

$$ss \oplus_F {ss'}_1 \ \sim\ ss \oplus_F {ss'}_2 \tag{5.41}$$

For (5.40), since list append preserves $\sim$ by Lemma 5.3.19(5.37)

$$\begin{aligned} ss_1 \oplus_F ss' &= ss_1 +\!\!+ ss' \\ &\sim ss_2 +\!\!+ ss' = ss_2 \oplus_F ss' \end{aligned}$$

and similarly for (5.41) in the other argument. □

The definitions above indeed form a model of WRAP ▷ CMON, the analogous result to Proposition 5.3.17.

Lemma 5.3.22. *Bundles (as $\sim$-equivalence classes of fences) equipped with operations $(\varepsilon_B, \oplus_B, wrap_B)$ are models of* WRAP ▷ CMON.

PROOF. Since fences are models of WRAP ▷ MON by Proposition 5.3.12, it follows that bundles are too by the definitions of $(\varepsilon_B, \oplus_B, wrap_B)$. To see that they are furthermore models of WRAP ▷ CMON, it remains to check symmetry of $\oplus_B$. Indeed,

$$\begin{aligned} [ss] \oplus_B [ss'] &= [ss \oplus_F ss'] \\ &= [ss' \oplus_F ss] \\ &= [ss'] \oplus_B [ss] \end{aligned}$$

where the second equality follows from the fact that $ss +\!\!+ ss' \sim ss' +\!\!+ ss$. □

Theorem 5.3.23. *The monad* Bundle *is presented by the theory* WRAP $\triangleright$ CMON. *That is, their algebras and models (respectively) are equivalent to each other*

$$\mathsf{Bundle}\text{-}\mathbf{Alg} \simeq \mathsf{Mod}(\textsc{wrap} \triangleright \textsc{cmon})$$

PROOF. Once again we seek to reuse a prior result about fences—in this case, Theorem 5.3.13. Let $(M, \oplus, \varepsilon, \mathit{wrap})$ be a model of WRAP $\triangleright$ CMON. To establish that bundles in Lemma 5.3.22 are the *free* such models, for any map f into M we require a unique homomorphism $\widehat{f_B}$ from bundles into M such that

$$f = \widehat{f_B} \circ \mathit{return}_B \tag{5.42}$$

Uniqueness: since $\widehat{f_B}$ is required to preserve ε, its action on the empty fence $[[\,]] = \varepsilon_B$ is determined

$$\widehat{f_B}\ [[\,]] = \varepsilon$$

By equation (5.42), its action on "tips" $[[\ \mathsf{Tip}\ x\ ]] = \mathit{return}_B(x)$ is determined

$$\widehat{f_B}\ [[\ \mathsf{Tip}\ x\ ]] = f\ x$$

By preservation of *wrap*, so too is its action on sticks of length $n > 0$

$$\widehat{f_B}\ [[\ \mathsf{Segment}^n\ (\mathsf{Tip}\ x)\ ]] = \mathit{wrap}^n\ (f\ x)$$

since every stick $[[\ \mathsf{Segment}^n\ (\mathsf{Tip}\ x)\ ]] = [(\mathit{map}\ \mathsf{Segment})^n\ [\ \mathsf{Tip}\ x\ ]] = [\mathit{wrap}_F^n\ [\ \mathsf{Tip}\ x\ ]] = \mathit{wrap}_B^n\ [[\ \mathsf{Tip}\ x\ ]]$. Finally, note that (an equivalence class of) fences $[ss]$ can be expressed as a $\oplus_B$ product of singletons, i.e. sticks. Thus by preservation of $\oplus$, the action on $[ss]$ is given by (the product of) actions on the individual sticks. For example, the action on two-stick fences is given by

$$\widehat{f_B}\ [[\ s, s'\ ]] = \widehat{f_B}\ [[\ s\ ]]\ \oplus\ \widehat{f_B}\ [[\ s'\ ]]$$

Hence $\widehat{f_B}$ is uniquely determined.

Existence: for an equivalence class of fences $[\,ss\,]$ we define $\widehat{f_B}$ as follows

$$\widehat{f_B}\ [\,ss\,] \triangleq \widehat{f_F}\ (ss)$$

where $\widehat{f_F}$ is as defined in the proof of Theorem 5.3.13[6] for fences

$$\begin{aligned}
&\widehat{f_F}\ [\,] && = \varepsilon \\
&\widehat{f_F}\ (\mathsf{Tip}\ x : ss) && = f\ x\ \oplus\ \widehat{f_F}\ ss \\
&\widehat{f_F}\ (\mathsf{Segment}\ s : ss) && = \mathit{wrap}\ (\widehat{f_F}\ [\ s\ ])\ \oplus\ \widehat{f_F}\ ss
\end{aligned}$$

but with the understanding that its codomain M is not only a model of WRAP▷MON but also WRAP▷CMON, so that $\oplus$ is symmetric. To see that $\widehat{f_B}$ is well-defined, we require $\widehat{f_F}\ ss = \widehat{f_F}\ ss'$ for any $ss \sim ss'$. For the empty fence $ss \equiv ss' \equiv [\,]$ this trivially holds. For the nonempty case, let ss' be a permutation σ of the sticks $s_1, \ldots, s_k$ in ss. Then

$$\begin{aligned}
\widehat{f_F}\ ss &= \widehat{f_F}\ [s_1, \ldots, s_k] \\
&= \widehat{f_F}\ [s_1]\ \oplus \ldots \oplus\ \widehat{f_F}\ [s_k] && \text{Defn } \widehat{f_F} \\
&= \widehat{f_F}\ [s_{\sigma(1)}]\ \oplus \ldots \oplus\ \widehat{f_F}\ [s_{\sigma(k)}] && \text{symm } \oplus \\
&= \widehat{f_F}\ [s_{\sigma(1)}, \ldots, s_{\sigma(k)}] && \text{Defn } \widehat{f_F} \\
&= \widehat{f_F}\ ss'
\end{aligned}$$

as required. Note that when M is considered as a model of WRAP▷MON, $\widehat{f_F}$ is a homomorphism of such models (as shown in the proof of Theorem 5.3.13) and from this it follows that $\widehat{f_B}$ is a homomorphism of WRAP▷CMON models. For example,

$$\begin{aligned}
\widehat{f_B}\ [\,ss\,]\ \oplus_B\ \widehat{f_B}\ [\,ss'\,] &= \widehat{f_F}\ (ss \oplus_F ss') \\
&= \widehat{f_F}\ ss\ \oplus\ \widehat{f_F}\ ss' && \widehat{f_F} \text{ homomorphism} \\
&= \widehat{f_B}\ [\,ss\,]\ \oplus\ \widehat{f_B}\ [\,ss'\,]
\end{aligned}$$

[6] There we called it $\hat{f}$.

Preservation of ε and *wrap* is similar. Finally, equation (5.42) is satisfied

$$\widehat{f_B}\ (return_B\ x) = \widehat{f_F}\ (return_F\ x) = f\ x$$

□

5.4. List transformer

In Section 5.1 we discussed the data type of lists in Haskell as offering a monadic (indeed, MonadPlus) effect of backtracking search. But while many of the effects in earlier chapters have natural counterparts as monad transformers (e.g. StateT for stateful effects, and so on), the `mtl` list transformer in Haskell was known for some time to be somewhat unsatisfactory. We shall discuss the reasons, together with the "done right" version of the transformer, from the perspective of combining theories distributively.

The `mtl` list transformer, which from here on we will call ListT_0, is given by composition with the list monad

$$\textbf{newtype}\ \mathsf{ListT}_0\ m\ a = \mathsf{ListT}_0\ \{\mathit{unListT}_0 :: m\ [a]\}$$
$$\textbf{deriving}\ \mathsf{Functor}$$

Since it is defined as a composition, we expect that it forms a monad precisely when there is a distributive law of the list monad over m, canonically defined (in Haskell syntax) as the following function[7]

$$\begin{aligned}
&\mathit{sequence} :: \mathsf{Monad}\ m \Rightarrow [m\ a] \longrightarrow m\ [a]\\
&\mathit{sequence}\ [\,] \qquad\qquad = \mathit{return}\ [\,]\\
&\mathit{sequence}\ (\mathit{xm} : \mathit{xms}) = \textbf{do}\\
&\quad x \leftarrow \mathit{xm}\\
&\quad \mathit{xs} \leftarrow \mathit{sequence}\ \mathit{xms}\\
&\quad \mathit{return}\ (x : \mathit{xs})
\end{aligned}$$

But it turns out that *sequence* fails to be a distributive law, unless (as discussed in [**52**]) m is a *commutative monad*. Recall from Definition 4.1.5

[7] A more generic *sequence* function is defined in the Haskell module `Data.Traversable`.

that a monad m in Haskell is commutative when we have the following equation between computations

$$\begin{array}{lll} \textbf{do} & x \leftarrow xm & \\ & y \leftarrow ym & = \\ & return\ (x, y) & \end{array} \qquad \begin{array}{lll} \textbf{do} & y \leftarrow ym & \\ & x \leftarrow xm & \\ & return\ (x, y) & \end{array}$$

In short, effects of m are independent of the order in which they are executed. Equivalently, the obvious Monad instance definition[8] for $\mathsf{ListT}_0\ m$ does not form a monad when m itself is not a commutative monad.

```
instance Monad m ⇒ Monad (ListT0 m) where
  return x   = ListT0 (return [x])
  xsm >>= f = ListT0 $
    fmap concat $ unListT0 xsm >>= mapM (unListT0 ∘ f)
```

For example, when stateful effects are combined with ListT_0, we have the non-monadic type

$$\mathsf{ListT}_0\ (\mathsf{State}\ s)\ a\ \cong\ s \longrightarrow ([a], s)$$

unlike the more widespread StateT combination

$$\mathsf{StateT}\ s\ [\,]\ a\ \cong\ s \longrightarrow [(a, s)]$$

of "backtracking state". While the latter can be thought of as presenting, upon a state transition, a choice of values each paired with its associated new state, the former presents the choice of values only with a single terminal state. More generally speaking, ListT_0 can be seen as a type of a list associated with the overall effect obtained by traversing each of its elements. Indeed, given that there are in general multiple ways of enumerating a list, one should not expect the accumulated effect to be the same for each—unless of course, the effect is commutative.

[8] *mapM f* is equivalent to *sequence* ∘ *map f*.

This implies a further practical limitation of ListT_0, for it does not offer an idiomatic means of programming with effectful streams. After all, a ListT_0 computation represents a whole list together with an overall effect all in one go—there is no facility for consuming this computation in a more "piecemeal", per-element fashion.

While the reasons above have lead many to dismiss ListT_0 in favour of (different variants of) ListT "done right", we show that when seen from a perspective of equational theories, it is nevertheless (perhaps contrary to what some may believe) not without mathematical basis.

5.4.1. Combining with backtracking

In what follows, we prove that ListT_0 may be characterised as a distributive tensor. To motivate, let us consider an example of combining backtracking with another very simple effect, via distributive tensor.

As in Section 5.3, we denote MON for the theory of monoids. We introduce an additional theory IDEM comprising a single unary idempotent operation.

$$\text{MON} \triangleq (\{\varepsilon, \oplus\}, \text{monoid laws})$$
$$\text{IDEM} \triangleq (\{|\cdot|\}, \text{idempotence})$$

Note the choice of name $|\cdot|$, suggestive of taking the absolute value, a function well-known to behave idempotently

$$||x|| = |x|$$

Terms in the free models of IDEM are easy to specify—they consist of the generator variables x and "$|x|$" values, as described by the following Haskell data type

```
data Idem a = Var a | Abs a
            deriving Functor
```

Its Monad instance embodies the idea that a variable may become an absolute value, but an absolute value always remains as such.

```
instance Monad Idem where
  return = Var
  Var x >>= f = f x
  Abs x >>= f = case f x of
    Var y  ⟶ Abs y
    absval ⟶ absval
```

The following is clear, since any unary operation commutes with itself.

Lemma 5.4.1. IDEM *is a commutative theory. Equivalently,* Idem *is a commutative monad.*

Consider now the theory MON ▷ IDEM combined by distributive tensor. That is, the theory obtained by union of the above theories MON + IDEM, together with distributivity of $\oplus$ over $|\cdot|$

$$|x| \oplus y = |x \oplus y| \tag{5.43}$$

$$x \oplus |y| = |x \oplus y| \tag{5.44}$$

Note of course that the nullary ε does not distribute.

Lemma 5.4.2. $\oplus$ *commutes with* $|\cdot|$ *in* MON ▷ IDEM.

Proof. By equations (5.44), (5.43) and idempotence,

$$|x| \oplus |y| = ||x| \oplus y| = ||x \oplus y|| = |x \oplus y|$$

□

Now let us consider the normal forms of the theory. Treating the distributivity equations and Lemma 5.4.2 as rewrite rules (in the sense of rewriting theory)

$$|x| \oplus y \longrightarrow |x \oplus y|$$

$$x \oplus |y| \longrightarrow |x \oplus y|$$

$$|x| \oplus |y| \longrightarrow |x \oplus y|$$

It is clear that an Abs subterm extends to its adjacent subterm (regardless of whether it too is an Abs). These rules describe a map

$$[\mathsf{Idem}\ a] \longrightarrow \mathsf{Idem}\ [a]$$

that "remembers" the presence of any Abs occurring in the given list. Indeed, it is precisely *sequence* (specialising m to Idem), or more concretely

$$\begin{aligned}
&\mathit{sequence}\ [\,] &&= \mathsf{Var}\ [\,] \\
&\mathit{sequence}\ (\mathsf{Var}\ x : \mathit{is}) &&= \mathit{fmap}\ (x{:})\ (\mathit{sequence}\ \mathit{is}) \\
&\mathit{sequence}\ (\mathsf{Abs}\ x : \mathit{is}) &&= \mathsf{Abs}\ (x : \mathit{map}\ \mathit{unIdem}\ \mathit{is})
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{unIdem}\ (\mathsf{Var}\ y) &&= y \\
&\mathit{unIdem}\ (\mathsf{Abs}\ y) &&= y
\end{aligned}$$

Hence by Lemma 5.4.1, it follows that *sequence* is a distributive law with $\mathsf{Idem}\ [a]$ a monad, and this is of course isomorphic to $\mathsf{ListT}_0\ \mathsf{Idem}\ a$.

Remark 5.4.3 (Cut)**.** By varying one of the distributivity equations, it is possible to obtain a theory of backtracking in conjunction with an operator reminiscent of "cut" in the logic programming language Prolog[9]. Specifically, rather than have $\oplus$ distribute leftwards over $|\cdot|$ (5.43)—which we interpret here as cut—*discard* the choice to the right of the cut

$$|x| \oplus y \longrightarrow |x|$$

[9]Personal communication with Maciej Pirog, Algebra of Programming meeting, Oxford.

Thus pruning off any further choices y. While this example is not known to be any of the "natural" combinations of theories, it is nevertheless obtained as a composite monad $\mathsf{Idem}\ [a]$, though a different one to our discussion above. In particular, its monad structure is given by an alternative distributive law to *sequence*.

This illustrates an interesting dichotomy—while constructions on theories account for many computationally interesting examples of combined effects, distributive laws are often the more "flexible" notion [**12**]—but only of course, when the monads compose, whereas theories can in general always be combined in one way or another.

The foregoing example alludes to the following general characterisation of ListT_0.

Theorem 5.4.4. *When $\mathbb{T}$ is a commutative theory (equivalently, a commutative monad T), the transformed monad $\mathsf{ListT}_0\ T$ is equivalent to the distributive tensor of theories $\textsc{mon} \triangleright \mathbb{T}$.*

To establish this, we proceed in a similar fashion to other correspondence results in this chapter, namely by exhibiting the left adjoint to the forgetful functor from models of $\textsc{mon} \triangleright \mathbb{T}$, in turn appealing to monadicity.

First let us fix some notation. Let $\mathbb{T} = (\Sigma, E)$ be a commutative theory as stated, with signature Σ and equations E. Equivalently, it is a commutative monad T. For a set X, $T(X)$ constructs the free models of $\mathbb{T}$, consisting of formal Σ-terms quotiented by E. As usual, Σ-terms are well-bracketed expressions (or if one prefers, *syntax trees*) built from "variables" $x \in X$ and operations $\sigma \in \Sigma$. Thus we understand $T(X)$ to comprise *equivalence classes*[10] of such terms, identified with respect to equivalence generated by E. Finally, let X^* denote the free monoid, or *lists* of elements in X. We

[10]We will find it convenient to omit the use of the usual equivalence class brackets $[-]$. Thus we should read "$t \in T(X)$" as implicitly being $[t] \in T(X)$.

occasionally use Haskell-like notation for common list operations, such as append ($+\!\!+$) and *map*. Let us now give a "syntactic" model of $\textsc{mon} \triangleright \mathbb{T}$.

Definition 5.4.5. Let X be a set. For any list $xs \in X^*$ and n-ary $\sigma \in \Sigma$, the set $T(X^*)$ is generated by the following inductive rules

$$\frac{}{xs} \qquad \frac{t_1 \ \ldots \ t_n}{\sigma(t_1, \ldots, t_n)}$$

We also give interpretations of the operations. For $\sigma \in \Sigma$, interpretation (as well as its extension to terms i.e. *derived* operations) is defined syntactically in the usual manner, thereby satisfying E. For the monoid operations, we interpret the unit ϵ as the empty list, and the multiplication as follows by pattern matching

$$\begin{aligned} xs \oplus ys &= xs +\!\!+ ys \\ xs \oplus \sigma(t_1, \ldots, t_n) &= \sigma(xs \oplus t_1, \ldots, xs \oplus t_n) \\ \sigma(t_1, \ldots, t_n) \oplus t &= \sigma(t_1 \oplus t, \ldots, t_n \oplus t) \end{aligned}$$

where in the base case we append the variable lists xs, ys. We will refer to these three equations as $\oplus$-1, $\oplus$-2, $\oplus$-3 respectively.

Lemma 5.4.6. *The above data* $(TX^*, \oplus, \epsilon, \sigma)$ *is a model of* $\textsc{mon} \triangleright \mathbb{T}$.

Proof. First we check the monoid laws. The unit law $t \oplus \epsilon = t$ is by induction on t. For $t \equiv xs$,

$$xs \oplus \epsilon = xs +\!\!+ \epsilon = xs$$

For $t \equiv \sigma(t_1, \ldots, t_n)$,

$$\begin{aligned} \sigma(t_1, \ldots, t_n) \oplus \epsilon &= \sigma(t_1 \oplus \epsilon, \ldots, t_n \oplus \epsilon) && \oplus\text{-3} \\ &= \sigma(t_1, \ldots, t_n) && \text{IH} \end{aligned}$$

The other unit law $\epsilon \oplus t = t$ is proved analogously, using $\oplus$-2 and induction hypothesis. Associativity $t \oplus (u \oplus v) = (t \oplus u) \oplus v$ is by induction on t. For

$t \equiv \sigma(t_1, \ldots, t_n)$,

$$\begin{aligned}
\sigma(t_1, \ldots, t_n) \oplus (u \oplus v) &= \sigma(t_1 \oplus (u \oplus v), \ldots, t_n \oplus (u \oplus v)) && \oplus\text{-3} \\
&= \sigma((t_1 \oplus u) \oplus v, \ldots, (t_n \oplus u) \oplus v) && \text{IH} \\
&= \sigma(t_1 \oplus u, \ldots, t_n \oplus u) \oplus v && \oplus\text{-3} \\
&= (\sigma(t_1, \ldots, t_n) \oplus u) \oplus v && \oplus\text{-3}
\end{aligned}$$

For $t \equiv xs$ we further distinguish cases; $u \equiv ys, v \equiv \sigma(v_1, \ldots, v_n)$ is analogous to the above except we use $\oplus$-2. The case $u \equiv ys$, $v \equiv zs$ follows from associativity of list append

$$xs \oplus (ys \oplus zs) = xs +\!\!+ ys +\!\!+ zs = (xs \oplus ys) \oplus zs$$

Finally for $u \equiv \sigma(u_1, \ldots, u_n)$,

$$\begin{aligned}
xs \oplus (\sigma(u_1, \ldots, u_n) \oplus v) &= xs \oplus \sigma(u_1 \oplus v, \ldots, u_n \oplus v) && \oplus\text{-3} \\
&= \sigma(xs \oplus (u_1 \oplus v), \ldots, xs \oplus (u_n \oplus v)) && \oplus\text{-2} \\
&= \sigma((xs \oplus u_1) \oplus v, \ldots, (xs \oplus u_n) \oplus v) && \text{IH} \\
&= \sigma(xs \oplus u_1, \ldots, xs \oplus u_n) \oplus v && \oplus\text{-3} \\
&= (xs \oplus \sigma(u_1, \ldots, u_n)) \oplus v && \oplus\text{-2}
\end{aligned}$$

It remains to show distributivity of $\oplus$ over the operations of Σ. The rightwards direction

$$u \oplus \sigma(t_1, \ldots, t_n) = \sigma(u \oplus t_1, \ldots, u \oplus t_n)$$

is by induction on u. The base case $u \equiv ys$ follows from $\oplus$-2

$$ys \oplus \sigma(t_1, \ldots, t_n) = \sigma(ys \oplus t_1, \ldots, ys \oplus t_n)$$

whereas the case $u \equiv \sigma'(u_1, \ldots, u_m)$ notably makes use of the fact that operations in Σ commute

$$\begin{aligned}
&\sigma'(u_1, \ldots, u_m) \oplus \sigma(t_1, \ldots, t_n) \\
&= \sigma' \begin{pmatrix} u_1 \oplus \sigma(t_1, \ldots, t_n) \\ \vdots \\ u_m \oplus \sigma(t_1, \ldots, t_n) \end{pmatrix} && \oplus\text{-3} \\
&= \sigma' \begin{pmatrix} \sigma(u_1 \oplus t_1, \ldots, u_1 \oplus t_n) \\ \vdots \\ \sigma(u_m \oplus t_1, \ldots, u_m \oplus t_n) \end{pmatrix} && \text{IH} \\
&= \sigma \begin{pmatrix} \sigma'(u_1 \oplus t_1, \ldots, u_m \oplus t_1) \\ \vdots \\ \sigma'(u_1 \oplus t_n, \ldots, u_m \oplus t_n) \end{pmatrix} && \text{comm } \sigma, \sigma' \\
&= \sigma \begin{pmatrix} \sigma'(u_1, \ldots, u_m) \oplus t_1 \\ \vdots \\ \sigma'(u_1, \ldots, u_m) \oplus t_n \end{pmatrix} && \oplus\text{-3}
\end{aligned}$$

Distributivity leftwards is immediate from $\oplus$-3. □

The following establishes that the construction in Lemma 5.4.6 is functorial.

Lemma 5.4.7. *The map $X \mapsto (TX^*, \oplus, \epsilon, \sigma)$ extends to a functor F.*

Proof. The action of F on functions $f : X \to Y$ is given by the map

$$\begin{aligned}
T(f^*) : T(X^*) &\longrightarrow T(Y^*) \\
xs &\longmapsto \mathit{map}\ f\ xs \\
\sigma(t_1, \ldots, t_n) &\longmapsto \sigma(T(f^*)(t_1), \ldots, T(f^*)(t_n))
\end{aligned}$$

Let us check that F is well-defined, in particular that $T(f^*)$ is a homomorphism of models. That operations $\sigma \in \Sigma$ are preserved by $T(f^*)$ is

immediate from the definition. Preservation of ϵ follows from the definition of *map*

$$Tf^*(\epsilon) = map\ f\ \epsilon = \epsilon$$

We show

$$Tf^*(t \oplus u) = Tf^*(t) \oplus Tf^*(u)$$

by induction on t. For $t \equiv \sigma(t_1, \ldots, t_n)$ we use the fact that $\oplus$ distributes over σ leftwards

$$\begin{aligned}
Tf^*(\sigma(t_1, \ldots, t_n) \oplus u) &= Tf^*(\sigma(t_1 \oplus u, \ldots, t_n \oplus u)) && \oplus\text{-3} \\
&= \sigma(Tf^*(t_1 \oplus u), \ldots, Tf^*(t_n \oplus u)) \\
&= \sigma(Tf^*(t_1) \oplus Tf^*(u), \ldots, Tf^*(t_n) \oplus Tf^*(u)) && \text{IH} \\
&= \sigma(Tf^*(t_1), \ldots, Tf^*(t_n)) \oplus Tf^*(u) && \text{distr} \leftarrow \\
&= Tf^*(\sigma(t_1, \ldots, t_n)) \oplus Tf^*(u)
\end{aligned}$$

The case $t \equiv xs, u \equiv \sigma(u_1, \ldots, u_n)$ is similar, except we use distributivity of $\oplus$ rightwards. Finally for $t \equiv xs, u \equiv ys$,

$$\begin{aligned}
Tf^*(xs \oplus ys) &= Tf^*(xs +\!\!+ ys) \\
&= map\ f\ (xs +\!\!+ ys) \\
&= map\ f\ xs +\!\!+ map\ f\ ys \\
&= Tf^*(xs) \oplus Tf^*(ys)
\end{aligned}$$

It remains to check the usual functor laws. We show that F preserves identities, by induction. For the base case,

$$Tid^*_X = map\ id_X = id_{TX^*}$$

In the inductive case,

$$\begin{aligned} Tid_X{}^*(\sigma(t_1,\ldots,t_n)) &= \sigma(Tid_X{}^*(t_1),\ldots,Tid_X{}^*(t_n)) \\ &= \sigma(id_{TX^*}(t_1),\ldots,id_{TX^*}(t_n)) \qquad \text{IH} \\ &= \sigma(t_1,\ldots,t_n) \\ &= id_{TX^*}(\sigma(t_1,\ldots,t_n)) \end{aligned}$$

Finally we show that F preserves composition, again by induction. In the base case,

$$T(g \circ f)^* = map\ (g \circ f) = map\ g \circ map\ f = Tg^* \circ Tf^*$$

In the inductive case,

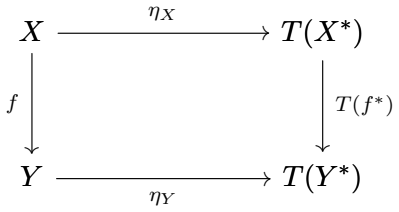
$$\begin{aligned} T(g \circ f)^*(\sigma(t_1,\ldots,t_n)) &= \sigma(T(g \circ f)^*(t_1),\ldots,T(g \circ f)^*(t_n)) \\ &= \sigma((Tg^* \circ Tf^*)(t_1),\ldots,(Tg^* \circ Tf^*)(t_n)) \quad \text{IH} \\ &= Tg^*(\sigma(Tf^*(t_1),\ldots,Tf^*(t_n))) \\ &= Tg^*(Tf^*(\sigma(t_1,\ldots,t_n))) \\ &= (Tg^* \circ Tf^*)(\sigma(t_1,\ldots,t_n)) \end{aligned}$$

□

Indeed, F constructs the *free* models of MON$\triangleright\mathbb{T}$, as shown in the following.

Proposition 5.4.8. *F is left adjoint to the forgetful functor $U : \mathsf{Mod}(\text{MON}\triangleright\mathbb{T}) \to \mathbf{Set}$.*

Proof. First we define the unit of the adjunction. Consider a function $\eta_X : X \to T(X^*)$ given by injection to a singleton list $x \mapsto [x]$. To see that this map is natural in X requires, for every $f : X \to Y$, commutativity of the diagram

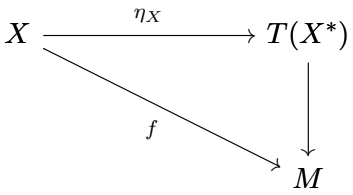


And for $x \in X$ we indeed have

$$\begin{aligned} T(f^*)(\eta_X(x)) &= map\ f\ [x] \\ &= [f(x)] \\ &= \eta_Y(f(x)) \end{aligned}$$

Hence $\eta : \mathsf{Id} \to UF$ is a natural transformation.

Next we exhibit a universal arrow—specifically, we show that for any model M with structure $(\oplus_M, \epsilon_M, \sigma_M)$ there is a bijection between functions $f : X \to M$ and maps of models $T(X^*) \to M$ making the following triangle commute



Suppose that such a map $\overline{f} : T(X^*) \to M$ exists. Since it is a homomorphism of models, its action on the empty list and Σ-terms is completely determined

$$\begin{aligned} \overline{f}(\epsilon) &= \epsilon_M \\ \overline{f}(\sigma(t_1, \ldots, t_n)) &= \sigma_M(\overline{f}(t_1), \ldots, \overline{f}(t_n)) \end{aligned}$$

Indeed (since $\oplus$ is also preserved by $\overline{f}$), so too is its action on non-empty lists $[x_1, \ldots, x_n]$

$$\begin{aligned}
\overline{f}([x_1, \ldots, x_n]) &= \overline{f}([x_1] +\!\!+ \ldots +\!\!+ [x_n]) \\
&= \overline{f}([x_n]) \oplus_M \ldots \oplus_M \overline{f}([x_n]) && \text{preserve } \oplus \\
&= f(x_1) \oplus_M \ldots \oplus_M f(x_n) && \text{comm } \triangle
\end{aligned}$$

Thus $\overline{f}$ is uniquely determined.

To check existence, we define a map

$$\begin{aligned}
\overline{f} : T(X^*) &\longrightarrow M \\
xs &\longmapsto \mathit{fold}_M(\mathit{map}\ f\ xs) \\
\sigma(t_1, \ldots, t_n) &\longmapsto \sigma_M(\overline{f}(t_1), \ldots, \overline{f}(t_n))
\end{aligned}$$

where fold_M sends

$$\begin{aligned}
\epsilon &\longmapsto \epsilon_M \\
[x_1, \ldots, x_n] &\longmapsto x_1 \oplus_M \ldots \oplus_M x_n
\end{aligned}$$

and check this is indeed a homomorphism of models. That Σ-operations are preserved is immediate from the definition. For empty lists,

$$\overline{f}(\epsilon) = \mathit{fold}_M(\mathit{map}\ f\ \epsilon) = \mathit{fold}_M(\epsilon) = \epsilon_M$$

Finally, we show

$$\overline{f}(t \oplus u) = \overline{f}(t) \oplus_M \overline{f}(u)$$

by induction on t. The case $t \equiv \sigma(t_1, \ldots, t_n)$ requires distributivity of $\oplus_M$ over σ_M leftwards

$$\begin{aligned}
\overline{f}(\sigma(t_1,\ldots,t_n) \oplus u) &= \overline{f}(\sigma(t_1 \oplus u, \ldots, t_n \oplus u)) \\
&= \sigma_M(\overline{f}(t_1 \oplus u), \ldots, \overline{f}(t_n \oplus u)) \\
&= \sigma_M(\overline{f}(t_1) \oplus_M \overline{f}(u), \ldots, \overline{f}(t_n) \oplus_M \overline{f}(u)) && \text{IH} \\
&= \sigma_M(\overline{f}(t_1), \ldots, \overline{f}(t_n)) \oplus_M \overline{f}(u) && \text{distr} \leftarrow \\
&= \overline{f}(\sigma(t_1,\ldots,t_n)) \oplus_M \overline{f}(u)
\end{aligned}$$

The case $t \equiv xs, u \equiv \sigma(u_1, \ldots, u_n)$ is similar, except distributivity rightwards is required instead. Finally for $t \equiv xs, u \equiv ys$ we use the fact that fold_M is a monoid homomorphism (marked $(*)$ below)

$$\begin{aligned}
\overline{f}(xs \oplus ys) &= \overline{f}(xs +\!\!+ ys) \\
&= (\mathit{fold}_M \circ \mathit{map}\ f)(xs +\!\!+ ys) \\
&= \mathit{fold}_M(\mathit{map}\ f\ xs +\!\!+ \mathit{map}\ f\ ys) \\
&= \mathit{fold}_M(\mathit{map}\ f\ xs) \oplus_M \mathit{fold}_M(\mathit{map}\ f\ ys) && (*) \\
&= \overline{f}(xs) \oplus_M \overline{f}(ys)
\end{aligned}$$

Hence $\overline{f}$ is indeed a homomorphism of models. Finally, $\overline{f}$ makes the triangle commute

$$\overline{f}(\eta_X(x)) = \mathit{fold}_M(\mathit{map}\ f\ [x]) = \mathit{fold}_M([f(x)]) = f(x)$$

We have thus shown that the operation $f \mapsto \overline{f}$ is bijective. With that we conclude $F \dashv U$ with unit η. □

Let us now return to the justification of ListT_0 as a distributive tensor.

Proof (theorem 5.4.4). By Proposition 5.4.8 we have a left adjoint F to the forgetful functor $U : \mathsf{Mod}(\text{MON} \triangleright \mathbb{T}) \to \mathbf{Set}$. By monadicity of U, algebras of the induced monad $UF = T(-)^*$ are equivalent to models of $\text{MON} \triangleright \mathbb{T}$. □

Note that Theorem 5.4.4 does not apply for non-commutative theories (equivalently, monads). While a distributive tensor always induces a monad, for a non-commutative m this will not be the (composite) monad given by $\mathsf{ListT}_0\ m$. For example, consider again the (single-location) state monad, which we present as a theory STATE in the usual manner.

$$\text{STATE} \triangleq (\{get, put\}, \text{mnemoid laws})$$

As discussed above, this fails to be a monad when combined with ListT_0. Indeed, as state is non-commutative (for example, the computation **do** $put\ 1; put\ 0; get$ is clearly not the same as **do** $put\ 0; put\ 1; get$), in this case the distributive tensor MON $\triangleright$ STATE does not give a meaningful equational theory that matches intuition. To see this, consider that among the equations we have distributivity of $\oplus$ over put in both the leftwards (5.45) and rightwards (5.46) direction

$$put_s(x) \oplus y = put_s(x \oplus y) \tag{5.45}$$

$$x \oplus put_s(y) = put_s(x \oplus y) \tag{5.46}$$

In the usual reading of backtracking search, the former makes sense—writing and following with x, with y the next choice, should indeed be the same as writing and continuing with $x \oplus y$. The latter however is less agreeable, since we think of $put_s(y)$ as a future choice relative to x, yet it suggests that the put_s can always be brought forward in time (crucially, even in the presence of writes in x itself).

Perhaps unsurprisingly, including both directions of distributivity leads to the possibility of contradictory equations, e.g.

$$\begin{aligned} put_0(put_1(x)) &= put_0(put_1(x \oplus \varepsilon)) \\ &\overset{!}{=} put_0(x \oplus put_1(\varepsilon)) \\ &= put_0(x) \oplus put_1(\varepsilon) \\ &\overset{!}{=} put_1(put_0(x) \oplus \varepsilon) \\ &= put_1(put_0(x \oplus \varepsilon)) = put_1(put_0(x)) \end{aligned}$$

where applications of distributivity rightwards (5.46) is marked (!). As a corollary, all *put* operations collapse in MON ▷ STATE. In the following, this somewhat problematic direction of distributivity will be excluded.

5.4.2. Alternative list transformer

Let us recall an alternative (often referred to as "done right") list monad transformer, as in Jaskelioff's thesis [**29**] and recently [**54**], where it has been characterised in terms of *Eilenberg-Moore monoids*—algebras of monads (on a monoidal category) equipped with monoidal structure. As with most monad transformers (and unlike ListT_0 thus far), it has the desirable property of transforming *any* monad. Specifically for a monad M, it yields

$$\mu X.M((- \times X) + 1)$$

Or more verbosely in Haskell syntax[11],

```
newtype ListT m a = ListT { next :: m (Step m a) }
data    Step  m a = Nil | Cons a (ListT m a)
```

Presented as such, it is clear that a $\mathsf{ListT}\ m$ computation is not so different to an ordinary Haskell list computation. By laziness, executing an effect in m at each step produces either the end of the list, or a value

[11] Adopting a presentation of ListT similar to that of the library `list-transformer` by Gabriel Gonzalez

together with *next* representing the rest of the computation. Thus we think of $\mathsf{ListT}\ m$ as a list with each element generated lazily with an associated effect in m, in contrast to ListT_0 representing a (lazy, but nevertheless) *complete* list in its entirety, with only a single "overall" effect.

This shift in the way backtracking is combined compares favourably in practice to the shortcomings of ListT_0, especially with regards to supporting programming with effectful streams, as discussed earlier. For example, the standard monadic map function

$$\begin{aligned}&\mathit{mapM} :: \mathsf{Monad}\ m \Rightarrow (a \longrightarrow m\ b) \longrightarrow [\,a\,] \longrightarrow m\ [\,b\,]\\ &\mathit{mapM}\ f = \mathit{sequence} \circ \mathit{map}\ f\end{aligned}$$

delivers (essentially) a $\mathsf{ListT}_0\ m\ b$ computation, but to compute the overall effect (via *sequence*) necessarily requires traversing the entirety of the intermediate list $[\,m\ b\,]$, buffering the output list $[\,b\,]$ as it does so. Clearly for very large or even infinite such lists, this is futile. By contrast, it is entirely natural to define a monadic map operation for ListT that streams in constant space[12].

5.4.3. Less can be more

In this section we will observe that *less* distributivity can yield a *more* practically capable list monad transformer. In [**32**], Moggi and Jaskelioff give an equational axiomatisation of ListT, amounting to the following. Let m be a monad presented by a theory $\mathbb{T}$. Then $\mathsf{ListT}\ m$ is a monad presented by the union of theories $\textsc{mon} + \mathbb{T}$ together with distributivity of $\oplus$ *leftwards* over all operations of $\mathbb{T}$. That is,

$$\sigma(x_1, \ldots, x_n) \oplus y = \sigma(x_1 \oplus y, \ldots, x_n \oplus y)$$

for σ an n-ary operation of $\mathbb{T}$. Notice in particular the exclusion of distributivity in the *rightwards* direction.

[12] `list-transformer` suggests at least three such possible variations of *mapM*

Example 5.4.9. Let us return to the Idem example from Section 5.4.1. By the description above, $\mathsf{ListT}\ \mathsf{Idem}$ is characterised by the equation (5.43) (but not (5.44)), as a rewrite rule

$$|x| \oplus y \longrightarrow |x \oplus y|$$

Now consider the term

$$1 \oplus 2 \oplus |3| \oplus 4 \oplus |5| \oplus |6| \oplus 7$$

The rewrite rule extends Abs subterms rightwards, so that this becomes the normal form

$$1 \oplus 2 \oplus \Big|3 \oplus 4 \oplus \big|5 \oplus |6 \oplus 7|\big|\Big|$$

Comparing the equational theories of the two list transformers, it is clear that while ListT is *close* to $\textsc{mon} \triangleright -$ (and hence ListT_0), it is precisely the absence of rightwards distributivity that is the marked difference. This suggests a novel notion of distributive tensor, in which not all distributivities are asserted.

Remark 5.4.10. At first glance it may appear that a variant of distributive tensor insisting only on leftwards distributivity should suffice, and indeed for ListT at least, it should. But as a note of warning, the intuitive idea of *direction* is specific to the particular case of a *binary* operation distributing. In general, an operation with arity $n + 1$ distributes in $n + 1$ positions, in which for cases $n > 1$ "direction" becomes less clear.

Thus an equational theory of ListT may be seen as similar to $\textsc{mon} \triangleright -$ but with distributivity in only the first position of $- \oplus -$ (or equivalently, similar to $\textsc{mon} \triangleright -$ but without distributivity in the last position). It remains as future work to investigate whether the aforementioned variants of distributive tensor admit natural characterisations in terms of models.

CHAPTER 6

Probability and nondeterminism

As a further application area of distributive tensor, we consider the combination of the computational effects probabilistic and nondeterministic choice. Both in isolation are standard examples of effects, but in some problem domains it is important to see them used in tandem—for example, to model probabilistic systems that depend on nondeterministic inputs. However, the algebraic properties that characterise their interaction are not entirely straightforward. In this chapter we show that a relatively well-accepted interaction of the two effects turns out to be distributive tensor, and furthermore we derive an equivalent formulation as a monad of convex-closed sets of probability distributions, as well as a technique for diagrammatic reasoning of program equations in this monad.

Section 6.1 reviews background material on the theory of convex spaces, defining the equational theory and fixing the notation. We prove several intermediate results (namely commutativity and distributivity of the theory, and a lemma about the quasi-associativity axiom) that will be useful in later sections. Note that we call this theory PROB since it gives rise to a monad of (finitely supported) probability distributions. Section 6.2 is a continuation of background for this chapter, defining the theory NDET of nondeterministic choice (or alternatively, semilattices) with some brief remarks from a process-algebraic vantage point. But while the theories PROB and NDET combine naturally by distributive tensor, their respective monad counterparts do not compose [**67**]. Thus in Section 6.3 we seek to derive a monad for this theory of *combined choice* by other means. This combination of the two effects—probability and nondeterminism—has been

studied in process algebra and domain-theoretic semantics e.g. [**57**]. Our contribution here is a derivation of this combination (as the *geometrically convex* monad) in a relatively simpler setting (i.e. the category of sets), where its characterisation as a distributive tensor is clearer. Although this particular characterisation has been mentioned before in passing [**27**], we do not believe that our main result, Theorem 6.3.7, is widely known.

Briefly, the derivation of the geometrically convex monad consists of the following. First we establish the free models of combined choice, over convex spaces (PROB-models). In doing so, we exhibit the free-forgetful adjunction between models of PROB choice and models of combined choice. Together with the knowledge of PROB-models in Section 6.1, we thus have two pairs of composable adjoint functors. Indeed, their composition yields the geometrically convex monad that we seek and by a monadicity result, we confirm the equivalence between this monad and the theory of combined choice.

The above results are applied in Section 6.4, where the focus shifts to equational reasoning in Haskell (or a similar functional programming setting). Under this lens, a contribution of this work is the identification of an incorrect assumption in equational axiomatisations of various algebraic effects such as probabilistic choice, that has previously been overlooked [**19, 18**]. Specifically, it was thought that monadic bind $\gg=$ distributes over probabilistic choice, both in the leftward and rightward directions. It turns out the latter is a much more contentious property than the former (which just states algebraicity of the operation), particularly if one assumes that such properties carry over without fuss when combined with other effects. For example, Section 6.4 considers again the algebraic effect of combined choice, where we have both probabilistic and nondeterministic choice operations. But we show that rightwards distributivity of $\gg=$ over probabilistic choice implies its commutativity with *any* operation, in particular nondeterministic choice. But this contradicts the usual equations of

combined choice—in fact we show that in such case, the probabilistic choice operations collapse. Note that we also give a direct equational proof of this result in [**1**].

As it can be difficult to get the equational properties of the interaction between probabilistic and nondeterministic choice right, in the remainder of Section 6.4 we explore a technique for reasoning about such equations visually, by taking a geometric interpretation of the free models of combined choice. That is, we interpret computations of the geometrically convex monad (over three values) as convex polygons on a plane. Guided by the definitions and results of Section 6.3, probabilistic choice of two convex polygons is given by their weighted sum, while nondeterministic choice is the convex hull of their union. We illustrate the use of this diagrammatic model by exhibiting an inequality (specifically, non-distributivity of nondeterministic over probabilistic choice). Compared to our earlier exposition of the work on diagrammatic reasoning [**1**], we give a more detailed treatment here, and in addition, describe the interpretation of $\gg\!=$.

6.1. Convex spaces

In what follows, we use the notation $\mathsf{Mod}(\mathbb{T})$ to mean the category of models of the algebraic theory $\mathbb{T}$ (in the category of sets, unless otherwise stated). Morphisms in $\mathsf{Mod}(\mathbb{T})$ are functions on the underlying carriers preserving the operations of $\mathbb{T}$ (in the usual sense). We also denote $\bar{p} \triangleq 1-p$.

To begin, let us define the notion of a formal convex space (see e.g. [**14**] for a recent overview), and recall the adjoint functors giving rise to the monad of its free algebras.

Definition 6.1.1. Let PROB denote the theory of formal convex spaces, consisting of a family of binary operations

$$+ \equiv (+_p)_{p\in[0,1]}$$

called *convex combination* or *probabilistic choice*, subject to the axioms unit, idempotence, quasi-symmetry and quasi-associativity, respectively

$$x +_0 y = y \tag{6.1}$$

$$x +_p x = x \tag{6.2}$$

$$x +_p y = y +_{\bar{p}} x \tag{6.3}$$

$$x +_p (y +_q z) = (x +_r y) +_s z \qquad (p = rs, \bar{s} = \bar{p}\bar{q}) \tag{6.4}$$

where axiom (6.4) has the side conditions $p = rs$ and $\bar{s} = \bar{p}\bar{q}$. Notice that as well as the unit law (6.1), we also have $x +_1 y = x$, derivable from (6.1) and (6.3).

We say that models of PROB are *convex spaces* (also called *barycentric algebras* by some authors), that is a set equipped with a family of binary operations satisfying (6.1)-(6.4). Homomorphisms are maps f preserving convex combination

$$f(x +_p y) = f(x) +_p f(y)$$

which we call *convex-linear* maps[1]. Thus $\mathsf{Mod}(\text{PROB})$ is the category of convex spaces and convex-linear maps.

For example, any real vector space can be understood as a convex space, with convex combination given by the usual linear combination structure

$$x +_\lambda y \triangleq \lambda x + \overline{\lambda} y$$

of scalar multiplication and vector addition.

Notation 6.1.2. Occasionally we will find it convenient to use n-ary convex combinations for $n > 2$

$$\sum_{i=1}^{n} \lambda_i x_i \qquad \left(\lambda_i \geqslant 0, \quad \sum_{i=1} \lambda_i = 1 \right)$$

[1]Convex spaces are closely related to the notion of *affine spaces*. Indeed, an affine space is a convex space (with extra properties), hence it is entirely appropriate to also call these *affine maps*.

defined in terms of the basic binary convex combination. For example, ternary convex combination may be written (by (6.4), interchangeably between two ways)

$$\begin{aligned} \lambda x + \theta y + \gamma z &\triangleq x +_{\lambda} \left(y +_{\frac{\theta}{\theta+\gamma}} z \right) && (\lambda \neq 1) \\ &\triangleq \left(x +_{\frac{\lambda}{\lambda+\theta}} y \right) +_{\lambda+\theta} z && (\gamma \neq 1) \end{aligned}$$

Notice that since $\lambda + \theta + \gamma = 1$, ternary convex combination may be interpreted as in the first equation provided $\lambda \neq 1$ so that we do not have both $\theta = \gamma = 0$, and similarly $\gamma \neq 1$ in the second equation.

It is worth pausing here to provide a little justification of the side conditions in the quasi-associativity axiom (6.4). The first of these $p = rs$ simply equates the probability of x, as does $\bar{s} = \bar{p}\bar{q}$ for y. Note then that $\bar{p}q = \bar{r}s$ (for z) is also a perfectly legitimate side condition—of course, any two suffices to determine the third. By making appropriate substitutions of the variables, the following justifies the use of quasi-associativity as a rewrite rule in both directions.

Lemma 6.1.3. *For $p, q \in [0,1]$ there exist $r, s \in [0,1]$ satisfying (6.4), and vice versa.*

Proof. Let $p, q \in [0,1]$. Then $\bar{s} = \bar{p}\bar{q} \in [0,1]$ since the unit interval is closed under multiplication and complement. Hence $s = \bar{\bar{s}} \in [0,1]$. From the other side condition, we derive the following expression for r

$$r = \begin{cases} \text{arbitrary}, & \text{if } p = q = 0 \\ \frac{p}{p+\bar{p}q}, & \text{otherwise.} \end{cases}$$

While r is arbitrary (within $[0,1]$) in the degenerate case, there we will as a convention fix $r = 0$, say. In the non-degenerate case, it is easy to verify

$$0 \leqslant \frac{p}{p + \bar{p}q} \leqslant 1$$

In the other direction, it is clear that $p = rs \in [0, 1]$, and from the other side condition,

$$q = \begin{cases} \text{arbitrary}, & \text{if } r = s = 1 \\ \frac{\bar{r}s}{\overline{rs}}, & \text{otherwise.} \end{cases}$$

Again we choose to fix $q = 0$ in the degenerate case. In the non-degenerate case, we have

$$0 \leqslant \frac{\bar{r}s}{\overline{rs}} \leqslant 1$$

□

Proposition 6.1.4. PROB *is a commutative (or* entropic*) theory—for* $p, q \in [0, 1]$*,*

$$(x_1 +_q y_1) +_p (x_2 +_q y_2) = (x_1 +_p x_2) +_q (y_1 +_p y_2)$$

PROOF.

$$\begin{aligned}
(x_1 +_q y_1) +_p (x_2 +_q y_2) &\overset{(6.4)}{=} x_1 +_r (y_1 +_s (x_2 +_q y_2)) && r = pq, \bar{p} = \bar{r}\bar{s} \\
&\overset{(6.3)}{=} x_1 +_r ((x_2 +_q y_2) +_{\bar{s}} y_1) \\
&\overset{(6.4)}{=} x_1 +_r (x_2 +_t (y_2 +_u y_1)) && t = \bar{s}q, s = \bar{t}\bar{u} \\
&\overset{(6.4)}{=} (x_1 +_v x_2) +_w (y_2 +_u y_1) && r = vw, \bar{w} = \bar{r}\bar{t} \\
&\overset{(6.3)}{=} (x_1 +_v x_2) +_w (y_1 +_{\bar{u}} y_2)
\end{aligned}$$

A straightforward calculation by substituting side conditions shows that $v = \bar{u} = p$ and $w = q$. □

Corollary 6.1.5. *Choice distributes over itself—for* $p, q \in [0, 1]$*,*

$$x +_p (y +_q z) = (x +_p y) +_q (x +_p z)$$

and similarly leftwards.

PROOF. By idempotence $x = x +_q x$ and Proposition 6.1.4. □

6.1.1. Free convex spaces

There is a forgetful functor $U_0 : \mathsf{Mod}(\textsc{prob}) \to \mathbf{Set}$ that given a convex space, discards the operations, keeping only the carrier set. For a set X, the left adjoint F_0 constructs the free convex space $F_0(X)$, consisting of a carrier the "simplex" $\Delta(X)$, which is a convex subset of the real vector space $\mathbb{R}^X$, with convex combination structure given by the linear combination structure of $\mathbb{R}^X$. The adjunction $F_0 \dashv U_0$ induces the so-called (finitary) *Giry* monad [**20**] $\mathscr{G}$.

Definition 6.1.6. The monad $\mathscr{G} = (\Delta, \eta^0, \mu^0)$ on **Set** consists of the functor Δ sending a set X to its *simplex*

$$\left\{ d : X \to [0,1] \;\middle|\; d \text{ finitely supported and } \sum_{x \in X} d(x) = 1 \right\}$$

Alternatively, we may think of the simplex $\Delta(X)$ as the set of finite *formal* convex combinations $\sum_i \lambda_i \underline{x}_i$ where $\underline{x}_i$ denotes $x_i \in X$ treated as a variable[2]. Two such terms are identified iff the weights λ_i of each $\underline{x}_i$ are the same. For a map f, the action $\Delta(f)$ is a map given by

$$\sum_i \lambda_i \underline{x}_i \longmapsto \sum_i \lambda_i \underline{f(x_i)}$$

The unit of the monad has components η^0_X given by $x \mapsto \underline{x}$. The multiplication has components $\mu^0_X : \Delta\Delta X \to \Delta X$ given by

$$\sum_i \lambda_i \underline{\sum_j \theta_{ij} \underline{x}_{ij}} \longmapsto \sum_i \left(\lambda_i \sum_j \theta_{ij} \right) \underline{x}_{ij}$$

6.2. Nondeterminism

Let us now define a theory of nondeterministic choice, which is closely related to probabilistic choice but with proper symmetry and associativity laws.

[2] In terms of distributions, $\underline{x}$ represents the Dirac "point" distribution on x

Definition 6.2.1. Let $\sqcap$ be a binary operation subject to idempotence, symmetry and associativity i.e. the theory of a semilattice. Denote this theory NDET.

Free NDET-models induce a variant of the powerset (or *Manes*) monad $\mathscr{M}$.

Definition 6.2.2. The monad $\mathscr{M} = (\mathcal{P}^{\aleph_0}, e, m)$ on **Set** consists of the finite nonempty powerset functor, denoted $\mathcal{P}^{\aleph_0}$

$$X \longmapsto \{Y \subseteq X \mid Y \neq \emptyset \text{ and finite}\}$$

with action on morphisms given by direct image. The unit e_X is given by $x \mapsto \{x\}$, the multiplication $m_X : (\mathcal{P}^{\aleph_0})^2(X) \to \mathcal{P}^{\aleph_0}(X)$ by set union

$$Z \longmapsto \bigcup_{Y \in Z} Y$$

Remark 6.2.3. As an aside, nondeterministic choice (as we have defined it here) is sometimes also called *internal* choice, especially in the process algebra literature. This is distinct from *external* choice which (algebraically) has a unit, conventionally thought of as a "stop" process S. Briefly, we think of external choice as offering the "environment"—some external entity—a choice as to how to proceed next. Since S by convention offers nothing to the environment however, the unit law simply axiomatises the idea that the environment cannot distinguish between a process P, and the "choice" between P and S. On the other hand, an internal choice is invisible to the environment. As an example, one might think of such choices being made by an operating system scheduler, which we as external users cannot influence. It turns out that internal choice *without a unit* is preferred in the context of combining with probability, as the presence of a unit leads to undesired properties. For instance, if we had such a "unit" process S' such that $P \sqcap S' = P$, since the environment has no influence over the choice, the process on the left could feasibly evolve to either P or indeed S', implying

(rather unreasonably) that every process P could be equivalent to S'. [**7**] has some discussion.

6.3. Combined choice

The combined theory of probabilistic and nondeterministic choice—which we shall henceforth simply call "combined choice", should at a minimum consist of all the equational laws of both sub-theories (in terms of Lawvere theories, the sum PROB + NDET). What is left must be a specification of how the choice operations $+_p$ and $\sqcap$ interact. For very good reasons (e.g. discussed in [**48, 18, 57**]), this interaction is distributivity of $+_p$ over $\sqcap$. In particular, if instead we insist on distributivity of $\sqcap$ over $+_p$, Keimel and Plotkin [**57**] show that this is equivalent to the theory of so-called join-distributive bisemilattices—the point being a collapse in the convex combination structure. Thus in keeping with the former, we require the equations

$$x +_p (y \sqcap z) = (x +_p y) \sqcap (x +_p z)$$

$$(y \sqcap z) +_p x = (y +_p x) \sqcap (z +_p x)$$

so that combined choice is the distributive tensor PROB $\triangleright$ NDET, in the terminology of Lawvere theories.

Remark 6.3.1. Relating back to the discussion in Remark 6.2.3, notice that in adorning nondeterministic choice with a unit S' (call this theory NDET' say), if one assumes that combined choice continues to be a distributive tensor PROB $\triangleright$ NDET', then by consequence we have additional interaction

$$S' +_p x = S'$$

namely that a probabilistic choice involving the unit is always the unit (even when $p = 0$), which is clearly problematic.

Having seen the monads $\mathscr{G}$ and $\mathscr{M}$ in correspondence with the respective theories PROB and NDET, one might well hope that a monad for combined choice could be sought by a composite of the above, presumably from the

composition of functors $\mathcal{P}^{\aleph_0}\Delta$. Unfortunately this line of thought proves not to be fruitful, for there is in general no distributive law of monads $\mathscr{G}$ over $\mathscr{M}$, as discussed by e.g. Varracca [**67**]. However, the following shows that one may freely construct a model of combined choice over an arbitrary *convex space*, (that is, a PROB-model, as opposed to just a set) which turns out to be a more promising approach.

6.3.1. Free models over convex spaces

Given an arbitrary convex space $(X, +)$, construct the set

$$\mathcal{P}_+^{\aleph_0}(X) \triangleq \{Y \in \mathcal{P}^{\aleph_0}(X) \mid Y \text{ convex wrt } +\}$$

that is, the finite nonempty subsets of X that are convex. From here on we will drop the superscript and sometimes the parentheses to simply $\mathcal{P}_+X$, both to lessen notational baggage and to avoid confusion with the powerset functor $\mathcal{P}^{\aleph_0}$ (of Definition 6.2.2). Only occasionally, we may even drop the subscript (to $\mathcal{P}$) when the convex operation is clear from context.

Given two such subsets $Y, Y' \in \mathcal{P}_+X$, their probabilistic p-choice is defined in a pointwise fashion

$$Y \oplus_p Y' \triangleq \{y +_p y' \mid y \in Y, y' \in Y'\}$$

for $p \in [0, 1]$. Occasionally we use

$$\bigoplus_{i=1}^{n} \lambda_i Y_i \qquad \left(\lambda_i \geqslant 0, \quad \sum_{i=1} \lambda_i = 1\right)$$

to denote n-ary probabilistic choice, defined in terms of binary choice $\oplus$ as discussed in Notation 6.1.2. Nondeterministic choice of Y, Y' is given by (convex hull of) their set union

$$Y \boxplus Y' \triangleq \mathrm{Conv}_+ (Y \cup Y')$$

where for any $Z \in \mathcal{P}^{\aleph_0} X$, convex hull is defined as

$$\begin{aligned}\operatorname{Conv}_+ Z &\triangleq \{z +_p z' \mid z, z' \in Z, p \in [0,1]\} \\ &\in \mathcal{P}_+ X\end{aligned}$$

We will usually drop the subscript to simply $\operatorname{Conv} Z$ when the convex operation is clear from context. More generally, n-ary nondeterministic choice is defined in the obvious way

$$\boxed{+}_{i=1}^{n} Y_i \triangleq \operatorname{Conv}\left(\bigcup_{i=1}^{n} Y_i\right)$$

Proposition 6.3.2. *The data $(\mathcal{P}_+ X, \oplus, ⨅)$ constitutes a model of combined choice.*

Proof. For idempotence of $\oplus$, first note that for any $p \in [0,1]$,

$$\begin{aligned}Y &= \{y \mid y \in Y\} \\ &= \{y +_p y \mid y \in Y\} \\ &\subseteq \{y +_p y' \mid y, y' \in Y\} \\ &= Y \oplus_p Y\end{aligned}$$

In the other direction,

$$\begin{aligned}Y \oplus_p Y &= \{y +_p y' \mid y, y' \in Y\} \\ &\subseteq Y\end{aligned}$$

since $(Y, +)$ is a convex space, so that $y +_p y' \in Y$. Hence $Y \oplus_p Y = Y$. Quasi-symmetry and quasi-associativity of $\oplus$ are similar, and follow from the respective laws of $+$.

The semilattice structure of $⨅$ is easy to verify. Idempotence for example,

$$Y ⨅ Y = \operatorname{Conv}(Y \cup Y) = \operatorname{Conv} Y = Y$$

where the last equality is due to Y already being convex (wrt $+$). Commutativity follows again from the semilattice structure of set union,

$$Y \boxplus Y' = \mathrm{Conv}\,(Y \cup Y') = \mathrm{Conv}\,(Y' \cup Y) = Y' \boxplus Y$$

Only associativity requires a little more care, requiring the use of quasi-associativity of $+$ and Lemma 6.1.3 as follows

$$\begin{aligned}
Y_1 \boxplus (Y_2 \boxplus Y_3) &= \mathrm{Conv}\,(Y_1 \cup (\mathrm{Conv}\,(Y_2 \cup Y_3))) \\
&= \{y_1 +_p y \mid y_1 \in Y_1, y \in \mathrm{Conv}\,(Y_2 \cup Y_3), p \in [0,1]\} \\
&= \{y_1 +_p (y_2 +_q y_3) \mid (y_i \in Y_i)_{i=1\ldots3}; p, q \in [0,1]\} \\
&= \{(y_1 +_r y_2) +_s y_3 \mid (y_i \in Y_i)_{i=1\ldots3}; p, q \in [0,1]; p = rs, \bar{s} = \bar{p}\bar{q}\} \\
&\subseteq \{(y_1 +_r y_2) +_s y_3 \mid (y_i \in Y_i)_{i=1\ldots3}; r, s \in [0,1]\} \\
&= \{y' +_s y_3 \mid y_3 \in Y_3, y' \in \mathrm{Conv}\,(Y_1 \cup Y_2), s \in [0,1]\} \\
&= \mathrm{Conv}\,(\mathrm{Conv}\,(Y_1 \cup Y_2) \cup Y_3) \\
&= (Y_1 \boxplus Y_2) \boxplus Y_3
\end{aligned}$$

The inclusion in the other direction follows a similar argument. Hence $\boxplus$ is associative, as required.

For distributivity, first recall from Corollary 6.1.5 that $+$ distributes over itself, in the sense

$$x +_p (y +_q z) = (x +_p y) +_q (x +_p z)$$

for all $p, q \in [0,1]$—notice on both sides we have x w.p. p, y w.p. $\bar{p}q$ and z w.p. $\bar{p}\bar{q}$. Then we reason as follows

$$\begin{aligned}
Y_1 \oplus_p (Y_2 \boxplus Y_3) &= \{y_1 +_p z \mid y_1 \in Y_1, z \in \mathrm{Conv}\,(Y_2 \cup Y_3)\} \\
&= \{y_1 +_p (y_2 +_q y_3) \mid y_1 \in Y_1, y_2 \in Y_2, y_3 \in Y_3, q \in [0,1]\} \\
&= \{(y_1 +_p y_2) +_q (y_1 +_p y_3) \mid y_1 \in Y_1, y_2 \in Y_2, y_3 \in Y_3, q \in [0,1]\} \\
&= \mathrm{Conv}\,((Y_1 \oplus_p Y_2) \cup (Y_1 \oplus_p Y_3)) \\
&= (Y_1 \oplus_p Y_2) \boxplus (Y_1 \oplus_p Y_3)
\end{aligned}$$

That $\oplus_p$ distributes over $\boxplus$ leftwards as well is a similar equational proof. □

Proposition 6.3.3. *The construction above defines a functor*

$$F : \mathsf{Mod}(\textsc{prob}) \longrightarrow \mathsf{Mod}(\textsc{prob} \triangleright \textsc{ndet})$$
$$(X, +) \longmapsto (\mathcal{P}_+ X, \oplus, \boxplus)$$

Proof. For a convex-linear map $f : (X, +) \to (Y, +')$, the action on morphisms is given by direct image, i.e.

$$Ff : (\mathcal{P}_+ X, \oplus, \boxplus) \longrightarrow (\mathcal{P}_{+'} Y, \oplus', \boxplus')$$
$$Z \longmapsto f[Z]$$

Let us check that F is well defined, in particular that Ff is a homomorphism of $\textsc{prob} \triangleright \textsc{ndet}$-models. To see that probabilistic choice is preserved for any $p \in [0, 1]$,

$$\begin{aligned}
(Ff)(Z \oplus_p Z') &= f[\{z +_p z' \mid z \in Z, z' \in Z'\}] \\
&= \{f(z +_p z') \mid z \in Z, z' \in Z'\} \\
&= \{f(z) +'_p f(z') \mid z \in Z, z' \in Z'\} \\
&= \{f(z) \mid z \in Z\} \oplus'_p \{f(z') \mid z \in Z'\} \\
&= (Ff)(Z) \oplus'_p (Ff)(Z')
\end{aligned}$$

For preservation of nondeterministic choice, first note that convex hull is preserved by image

$$\begin{aligned}
f[\operatorname{Conv} Z] &= \{f(z +_p z') \mid z, z' \in Z, p \in [0, 1]\} \\
&= \{f(z) +'_p f(z') \mid z, z' \in Z, p \in [0, 1]\} \\
&= \operatorname{Conv} f[Z]
\end{aligned}$$

so that

$$\begin{aligned}(Ff)(Z \boxplus Z') &= f[\mathrm{Conv}\,(Z \cup Z')] \\ &= \mathrm{Conv}\, f[(Z \cup Z')] \\ &= \mathrm{Conv}\,(f[Z] \cup f[Z']) \\ &= (Ff)(Z) \boxplus' (Ff)(Z')\end{aligned}$$

Finally, the functor laws go through as follows.

$$\begin{aligned}(F\ \mathsf{id}_{(X,+)})(Z) &= \mathsf{id}_{(X,+)}[Z] \\ &= Z \\ &= \mathsf{id}_{F(X,+)}(Z)\end{aligned}$$

$$\begin{aligned}(F(f \circ g))(Z) &= (f \circ g)[Z] \\ &= f[g[Z]] \\ &= (Ff)((Fg)(Z)) \\ &= (Ff \circ Fg)(Z)\end{aligned}$$

□

In fact, as the following shows, F is a free construction—it creates the *free* models of combined choice over convex spaces.

Proposition 6.3.4. *The forgetful functor* $U : \mathsf{Mod}(\textsc{prob}\triangleright\textsc{ndet}) \to \mathsf{Mod}(\textsc{prob})$ *given by discarding the semilattice operation, is right adjoint to* F*.*

Proof. For a convex space $(X, +)$, consider the function $\eta_{(X,+)} : X \to \mathcal{P}_+X$ given by the singleton map $x \mapsto \{x\}$. Clearly singletons are convex subsets, and notice that this map preserves probabilistic choice i.e. for $x, y \in X$ and

p in the unit interval,

$$\begin{aligned}\eta_{(X,+)}(x +_p y) &= \{x +_p y\} \\ &= \{x\} \oplus_p \{y\} \\ &= \eta_{(X,+)}(x) \oplus_p \eta_{(X,+)}(y)\end{aligned}$$

so that we have a convex-linear map

$$\eta_{(X,+)} : (X, +) \longrightarrow (\mathcal{P}_+X, \oplus)$$

To show naturality, we need for any convex-linear map f the following diagram to commute

$$\begin{array}{ccc} (X,+) & \xrightarrow{\eta_{(X,+)}} & (\mathcal{P}_+X, \oplus) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle UFf} \\ (Y,+') & \xrightarrow[\eta_{(Y,+')}]{} & (\mathcal{P}_{+'}Y, \oplus') \end{array}$$

And indeed for any $x \in X$, we have

$$\begin{aligned}(UFf)(\eta_{(X,+)}(x)) &= f[\{x\}] \\ &= \{f(x)\} \\ &= \eta_{(Y,+')}(f(x))\end{aligned}$$

Hence $\eta : \mathsf{Id} \to UF$ is a natural transformation.

Now for a model $(Y, +, \sqcap)$, consider the function $\varepsilon_{(Y,+,\sqcap)} : \mathcal{P}_+Y \to Y$ given by

$$\begin{aligned}\{y_1, \ldots, y_n\} &\longmapsto \prod_{i=1}^{n} y_i \\ &\equiv y_1 \sqcap \ldots \sqcap y_n\end{aligned}$$

This map preserves probabilistic choice, for

$$\begin{aligned}\varepsilon_{(Y,+,\sqcap)}(Z \oplus_p Z') &= \bigsqcap_{z\in Z, z'\in Z'} z +_p z' \\ &= \bigsqcap_{z\in Z} z +_p \bigsqcap_{z'\in Z'} z' \\ &= \varepsilon_{(Y,+,\sqcap)}(Z) +_p \varepsilon_{(Y,+,\sqcap)}(Z')\end{aligned}$$

where the second equality follows from distributivity of $+_p$ over $\sqcap$. To see that nondeterministic choice is also preserved,

$$\begin{aligned}\varepsilon_{(Y,+,\sqcap)}(Z \boxplus Z') &= \varepsilon_{(Y,+,\sqcap)}(\mathrm{Conv}\,(Z \cup Z')) \\ &= \bigsqcap_{\substack{z\in Z, z'\in Z',\\ p\in[0,1]}} z +_p z' \\ &= \bigsqcap_{p\in[0,1]} \left(\bigsqcap_{z\in Z, z'\in Z'} z +_p z' \right) \\ &= \bigsqcap_{p\in[0,1]} \left(\bigsqcap_{z\in Z} z +_p \bigsqcap_{z'\in Z'} z' \right) \\ &= \bigsqcap_{z\in Z} z \sqcap \bigsqcap_{z'\in Z'} z' \\ &= \varepsilon_{(Y,+,\sqcap)}(Z) \sqcap \varepsilon_{(Y,+,\sqcap)}(Z')\end{aligned}$$

where the fourth equality is due to distributivity again, and the fifth by virtue of a "convexity" property: if x and y are possible outcomes of a nondeterministic choice, by distributivity and idempotence we have that

$$\begin{aligned}x \sqcap y &= (x \sqcap y) +_p (x \sqcap y) \\ &= (x +_p x) \sqcap (x +_p y) \sqcap (y +_p x) \sqcap (y +_p y)\end{aligned}$$

for arbitrary probability p.

Hence our map is indeed a morphism of models

$$\varepsilon_{(Y,+,\sqcap)} : (\mathcal{P}_+ Y, \oplus, \boxplus) \longrightarrow (Y, +, \sqcap)$$

To check naturality, for any model homomorphism h we seek commutativity of the following diagram

$$\begin{array}{ccc} (\mathcal{P}_+Y, \oplus, \boxplus) & \xrightarrow{\varepsilon_{(Y,+,\sqcap)}} & (Y, +, \sqcap) \\ \Big\downarrow{\scriptstyle FUh} & & \Big\downarrow{\scriptstyle h} \\ (\mathcal{P}_{+'}Z, \oplus', \boxplus') & \xrightarrow[\varepsilon_{(Z,+',\sqcap')}]{} & (Z, +', \sqcap') \end{array}$$

And indeed for any convex subset $S \in \mathcal{P}_+Y$,

$$\begin{aligned} h(\varepsilon_{(Y,+,\sqcap)}(S)) &= h\left(\prod_{y \in S} y\right) \\ &= \prod_{y \in S}{}' h(y) \\ &= \varepsilon_{(Z,+',\sqcap')}(h[S]) \\ &= \varepsilon_{(Z,+',\sqcap')}((FUh)(S)) \end{aligned}$$

where the second equality holds by preservation of nondeterministic choice. Hence, $\varepsilon : FU \to \mathsf{Id}$ is indeed a natural transformation.

Finally, to show that F and U are adjoint, we check the "triangle" laws. That is, we require commutativity of the following diagrams

$$\begin{array}{ccc} & (\mathcal{P}_+Y, \oplus) & \\ {\scriptstyle \eta_{(Y,+)}}\nearrow & & \searrow{\scriptstyle U\varepsilon_{(Y,+,\sqcap)}} \\ (Y, +) & \xrightarrow[\mathsf{id}]{} & (Y, +) \end{array} \qquad \begin{array}{ccc} & F(\mathcal{P}_+X, \oplus) & \\ {\scriptstyle F\eta_{(X,+)}}\nearrow & & \searrow{\scriptstyle \varepsilon_{(\mathcal{P}_+X,\oplus,\boxplus)}} \\ (\mathcal{P}_+X, \oplus, \boxplus) & \xrightarrow[\mathsf{id}]{} & (\mathcal{P}_+X, \oplus, \boxplus) \end{array}$$

Diagram-chasing on the left, we have for any $y \in Y$

$$\begin{aligned} (U\varepsilon_{(Y,+,\sqcap)} \circ \eta_{(Y,+)})(y) &= (U\varepsilon_{(Y,+,\sqcap)})(\{y\}) \\ &= \bigsqcap y \\ &= y \end{aligned}$$

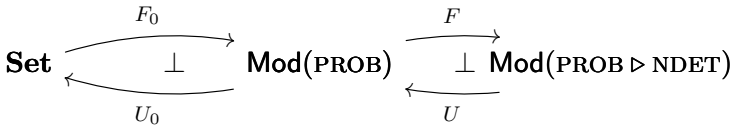


FIGURE 6.1. Adjoint functors considered so far

And taking a convex subset $S \in \mathcal{P}_+X$ on the right,

$$\begin{aligned}
(\varepsilon_{(\mathcal{P}_+X,\oplus,\boxplus)} \circ F\eta_{(X,+)})(S) &= \varepsilon_{(\mathcal{P}_+X,\oplus,\boxplus)}(\eta_{(X,+)}[S]) \\
&= \varepsilon_{(\mathcal{P}_+X,\oplus,\boxplus)}(\{\{x\} \mid x \in S\}) \\
&= \bigboxplus_{x\in S} \{x\} \\
&= \mathrm{Conv}\left(\bigcup_{x\in S} \{x\}\right) \\
&= \mathrm{Conv}\, S \\
&= S
\end{aligned}$$

Notice that the carrier of $F(\mathcal{P}_+X, \oplus)$ is $\mathcal{P}_\oplus\mathcal{P}_+X$, so that the set of singletons $\{\{x\} \mid x \in S\} \in \mathcal{P}_\oplus\mathcal{P}_+X$ is well-defined in the second equality.

With this we conclude $F \dashv U$, with unit η and counit ε. □

6.3.2. Geometrically convex monad

Recall the adjunction $F_0 \dashv U_0$ for convex spaces as described in Section 6.1.1. Consider this composed with the adjunction $F \dashv U$ in Proposition 6.3.4. In particular, the composition of left adjoints FF_0 (along the top of the picture in Figure 6.1) constructs a free model of combined choice over an arbitrary set, as described briefly in the following.

Let X be a set. The free model of combined choice over X is the following data,

$$FF_0X = (\mathcal{P}_+\Delta X, \oplus, \boxplus)$$

where $(\Delta X, +)$ denotes the free PROB-model over X, consisting of the formal convex combinations $\underline{x}_1 +_\lambda \underline{x}_2$ (and more generally $\sum_i \lambda_i \underline{x}_i$). Hence $\mathcal{P}_+\Delta X$ above comprises the finite nonempty subsets of such terms, where each subset is convex with respect to $+$. As before, probabilistic choice $\oplus$ is defined pointwise with respect to $+$, and nondeterministic choice $\boxplus$ by convex hull of union.

Definition 6.3.5. The composite adjunction $FF_0 \dashv U_0U$ induces a monad, the *geometrically convex* monad $\mathscr{P}_\Delta$ on **Set**, consisting of the (composite) functor $\mathcal{P}\Delta$. The action of the functor on a set X is defined as above. For a map $f : X \to X'$, its action on f is given by image under Δf

$$\begin{array}{ccc} \mathcal{P}_+\Delta X & & \left\{ \ldots, \sum_i \lambda_i \underline{x}_i \, , \ldots \right\} \\ \Big\downarrow{\scriptstyle \mathcal{P}\Delta f} & & \Big\downarrow \\ \mathcal{P}_{+'}\Delta X' & & \left\{ \ldots, \sum_i{}' \lambda_i \underline{f(x_i)} \, , \ldots \right\} \end{array}$$

To calculate the rest of the monad structure $\mathscr{P}_\Delta$, let us first recall a relevant formula about composite adjunctions (Mac Lane [**41**], Chapter on *Adjoints*).

Theorem 6.3.6. *Given two pairs of composable adjoint functors, such as* $(F_0 \dashv U_0, \eta^0, \varepsilon^0)$ *and* $(F \dashv U, \eta, \varepsilon)$ *in Figure 6.1, we have a composite adjunction* $FF_0 \dashv U_0U$ *with respective unit and counit*

$$H \triangleq U_0\eta F_0 \circ \eta^0$$

$$E \triangleq \varepsilon \circ F\varepsilon^0 U$$

By applying Theorem 6.3.6, let us give the structure of the monad $\mathscr{P}_\Delta$ explicitly. The unit H_X is given by

$$\begin{array}{ccccc} X & \xrightarrow{\eta^0_X} & \Delta X & \xrightarrow{U_0\eta_{(\Delta X,+)}} & \mathcal{P}_+\Delta X \\ x & \longmapsto & \underline{x} & \longmapsto & \{\underline{x}\} \end{array}$$

Before giving the multiplication, first note that for a PROB-model $(X, +')$, the counit of the adjunction $F_0 \dashv U_0$ is as follows

$$\begin{aligned} \varepsilon^0_{(X,+')} : (\Delta X, +) &\longrightarrow (X, +') \\ \sum_i \lambda_i \underline{x}_i &\longmapsto \sum_i{}' \lambda_i x_i \end{aligned}$$

Again by Theorem 6.3.6, the counit $E_{(Y,+',\sqcap)}$ of the composite adjunction then amounts to a composition of functions

$$\mathcal{P}_+\Delta Y \xrightarrow{F\varepsilon^0_{(Y,+')}} \mathcal{P}_{+'}Y \xrightarrow{\varepsilon_{(Y,+',\sqcap)}} Y$$

—in fact, a composition of PROB $\triangleright$ NDET-model homomorphisms, preserving probabilistic and nondeterministic choice (though only the carriers shown here for economy of notation). This map is given by

$$\left\{ \ldots, \sum_i \lambda_i \underline{y}_i, \ldots \right\} \longmapsto \left\{ \ldots, \sum_i{}' \lambda_i y_i, \ldots \right\} \longmapsto \prod_j \left(\sum_i{}' \lambda_i y_i \right)_j$$

Equipped with E, we may now derive the multiplication $M = U_0UEFF_0$ of the monad. The component M_X is a function

$$\mathcal{P}_{+^2}\Delta\mathcal{P}_+\Delta X \xrightarrow{U_0UE_{(\mathcal{P}_+\Delta X,\oplus,\boxplus)}} \mathcal{P}_+\Delta X$$

The domain of this map is worth some explanation. For any (finite nonempty convex) subset $S \in \mathcal{P}_+\Delta X$ of formal terms $\sum_i \lambda_i \underline{x}_i$, the (free) PROB-model $(\Delta\mathcal{P}_+\Delta X, +^2)$ should be understood as containing (formal) convex combinations $\sum_j^2 \theta_j \underline{S}_j$. Thus the domain of M_X comprises finite nonempty sets of such terms, convex with respect to $+^2$.

With sufficient notation fixed, we can give the definition of the multiplication M_X as

$$\left\{ \ldots, \sum_j{}^2 \theta_j \underline{S}_j \, , \ldots \right\} \longmapsto \boxed{+}_k \left(\bigoplus_j \theta_j S_j \right)_k$$

This completes the description of the geometrically convex monad $\mathscr{P}_\Delta = (\mathcal{P}\Delta, H, M)$.

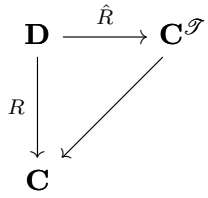
6.3.3. Relating the theory of combined choice with $\mathscr{P}_\Delta$

Returning to the original question, the monad we informally referred to as "convex sets of probability distributions" is precisely $\mathscr{P}_\Delta$, and it indeed characterises the theory PROB $\triangleright$ NDET of combined choice, as recorded in the following.

Theorem 6.3.7. *The models of combined choice are equivalent to the algebras of the geometrically convex monad*

$$\mathsf{Mod}(\text{PROB} \triangleright \text{NDET}) \simeq \mathbf{Set}^{\mathscr{P}_\Delta}$$

Before justifying this result, let us recall briefly the notion of monadicity (see e.g. Mac Lane [**41**] for much more detail). Let $L \dashv R : \mathbf{D} \to \mathbf{C}$ be an adjunction, inducing a monad $\mathscr{T}$ on $\mathbf{C}$ formed by the composite RL. This monad in turn gives rise to another adjunction, this time between $\mathbf{C}$ and the Eilenberg-Moore category $\mathbf{C}^{\mathscr{T}}$, with right adjoint $\mathbf{C}^{\mathscr{T}} \to \mathbf{C}$ (the evident forgetful functor sending a $\mathscr{T}$-algebra (C, α) to its carrier object C). Then there is a "comparison" functor $\hat{R} : \mathbf{D} \to \mathbf{C}^{\mathscr{T}}$ such that the diagram commutes.



We say that a functor R is *monadic* when it has a left adjoint that exhibits the comparison functor $\hat{R}$ as an equivalence of categories $\mathbf{D} \simeq \mathbf{C}^{\mathscr{T}}$. In general, one might check monadicity of a functor by the conditions characterised in Beck's monadicity theorem (Mac Lane, chapter VI). Indeed, a consequence of Beck's theorem is that forgetful functors $\mathsf{Mod}(\mathbb{T}) \to \mathbf{Set}$ from models of algebraic theories $\mathbb{T}$ are always monadic in this precise sense. With this in mind, the theorem is simply an appeal to this fact.

PROOF (THEOREM 6.3.7). The composite forgetful functor $U_0U : \mathsf{Mod}(\text{PROB}\triangleright \text{NDET}) \to \mathbf{Set}$ is monadic, in the sense discussed above. The equivalence follows immediately. □

6.4. Diagrammatic reasoning

Let us take the formal development of combined choice from above, and formulate this in the setting of Haskell. Adopting a similar presentation to [**19**], we declare a class for monadic types supporting a binary "nondeterministic choice" operation

```
class Monad m ⇒ MonadNondet m where
  (⊓) :: m a ⟶ m a ⟶ m a
```

where we require $\sqcap$ to be symmetric, idempotent and associative, i.e. equations of a semilatice. As discussed in Section 6.2, the canonical instance is a monad of (finite, nonempty) sets[3]—notice for example, the requirement for symmetry and idempotence rules out both lists and Maybe as instances. As usual, we would like the operation $\sqcap$ to be *algebraic*, that is, respecting $\gg\!=$ in the sense

$$xm \sqcap ym \gg\!= f = (xm \gg\!= f) \sqcap (ym \gg\!= f) \tag{6.5}$$

for all $xm, ym :: m\ a$, maps $f :: a \longrightarrow m\ b$ and MonadNondet m. In words, monadic composition $(\gg\!= f)$ preserves nondeterministic choice $\sqcap$. Indeed,

[3]For example, the library `Data.Set.Monad` contains a set-based data type

any nondeterministic computation composed with a continuation may be obtained simply by "pushing" the continuation through the subcomputations.

Similarly we will declare a class for monadic types supporting a family of binary "probabilistic" choice operations, indexed over probabilities—values of a type Prob (e.g. rationals in the closed unit interval).

$$\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadProb}\ m\ \textbf{where}$$
$$\quad \mathit{choice} :: \mathsf{Prob} \longrightarrow m\ a \longrightarrow m\ a \longrightarrow m\ a$$

Using an infix operator $+_{(-)}$, we write $\mathit{choice}\ p\ \mathit{xm}\ \mathit{ym}$ as $\mathit{xm} +_p \mathit{ym}$. We require instances of MonadProb to satisfy idempotence, quasi-symmetry and quasi-associativity, i.e. equations (6.2)-(6.4), as well as algebraicity

$$\mathit{xm} +_p \mathit{ym} \gg\!= f = (\mathit{xm} \gg\!= f) +_p (\mathit{ym} \gg\!= f) \tag{6.6}$$

for all $p :: \mathsf{Prob}$, $\mathit{xm}, \mathit{ym} :: m\ a$, maps $f :: a \longrightarrow m\ b$ and $\mathsf{MonadProb}\ m$.

Finally, mirroring Section 6.3, we declare a class of combined choice

$$\textbf{class}\ (\mathsf{MonadNondet}\ m, \mathsf{MonadProb}\ m) \Rightarrow \mathsf{MonadProbNondet}\ m$$

that introduces no new methods, but is required to satisfy the additional law that *choice* distributes over $\sqcap$.

6.4.1. Distributivity of $\gg\!=$

The equations above for algebraicity, that is $\gg\!=$ distributing *leftwards*, seem sensible enough as axioms—indeed, they are justified in universal algebra [**58**]. How about distributivity *rightwards*? Previously [**19, 18**], it was thought that the latter should also be axiomatic for MonadProb, but this was later [**1**] shown that when combined with nondeterminism (i.e. into MonadProbNondet), leads to a collapse of the probabilistic choice operations.

The problematic equation

$$\begin{array}{c} xm \gg\!= \lambda x \longrightarrow f\ x +_p g\ x \\ \| \\ (xm \gg\!= f) +_p (xm \gg\!= g) \end{array} \tag{6.7}$$

for $p :: \mathsf{Prob}$, $xm :: m\ a$ and maps $f, g :: a \longrightarrow m\ b$, is in fact not so problematic (indeed, perfectly agreeable) when the monad m is merely MonadProb. To see why, one should understand the equation (6.7) as implying that $+_p$ *commutes with everything.*

Example 6.4.1. Take xm to be *return* $1 +_q$ *return* 2, for $q :: \mathsf{Prob}$. Then equation (6.7) becomes

$$(f_1 +_p g_1) +_q (f_2 +_p g_2) = (f_1 +_q f_2) +_p (g_1 +_q g_2)$$

which holds, by Proposition 6.1.4.

Thus in the present setting, with MonadProb m, equation (6.7) implies that probabilistic choice commutes with itself, which is indeed the case. However, when the ambient monad m is "larger"—in particular, MonadProbNondet—one needs to be careful, for the equation implies additional commutativity between $\sqcap$ and *choice* operations, a special case of the following fact.

Proposition 6.4.2. *Idempotent, symmetric binary operations coincide when they commute with each other.*

PROOF. Let $\star$, $\bullet$ be idempotent, symmetric and commutative. Then for any x, y

$$\begin{aligned}
x \star y &= (x \bullet x) \star (y \bullet y) && \text{idem } \bullet \\
&= (x \star y) \bullet (x \star y) && \text{comm} \\
&= (y \star x) \bullet (x \star y) && \text{symm } \star \\
&= (y \bullet x) \star (x \bullet y) && \text{comm} \\
&= (x \bullet y) \star (x \bullet y) && \text{symm } \bullet \\
&= x \bullet y && \text{idem } \star
\end{aligned}$$

□

Corollary 6.4.3. *Equation* (6.7) *implies that* $\sqcap$ *commutes with all* $+_p$*, in particular* $+_{\frac{1}{2}}$*, which is idempotent and symmetric (proper). Hence by Proposition 6.4.2,* $\sqcap = +_{\frac{1}{2}}$*.*

From this observation, it becomes evident that other *choice* operations collapse. For example, given x, y

$$\begin{aligned}
x +_{\frac{1}{2}} y &= (x +_{\frac{1}{2}} x) +_{\frac{1}{2}} y \\
&= (x \sqcap x) \sqcap y \\
&= x \sqcap (x \sqcap y) \\
&= x +_{\frac{1}{2}} (x +_{\frac{1}{2}} y) \\
&= (x +_{\frac{2}{3}} x) +_{\frac{3}{4}} y \\
&= x +_{\frac{3}{4}} y
\end{aligned}$$

But we also have

$$\begin{aligned} x +_{\frac{1}{2}} y &= x +_{\frac{1}{2}} (y +_{\frac{1}{2}} y) \\ &= x \sqcap (y \sqcap y) \\ &= (x \sqcap y) \sqcap y \\ &= (x +_{\frac{1}{2}} y) +_{\frac{1}{2}} y \\ &= x +_{\frac{1}{4}} (y +_{\frac{1}{3}} y) \\ &= x +_{\frac{1}{4}} y \end{aligned}$$

So that $+_{\frac{1}{4}} = +_{\frac{1}{2}} = +_{\frac{3}{4}}$—and so on.

The subtlety is that while the equation holds for MonadProb, it is not a reasonable property to have in MonadProbNondet. This may seem somewhat counter-intuitive. After all, in the setting of algebraic theories we think of all operations and laws of a smaller theory transporting into a larger theory where they continue to hold—the theory $\mathsf{P} \triangleright \mathsf{N}$ is *at least* the sum $\mathsf{P} + \mathsf{N}$ of its parts, including (notably) their equations. But it is worth pointing out that while this is true of the more "usual" algebraic equations (such as idempotence, symmetry, associativity etc.), distributivity of $\gg\!=$ is not such an algebraic equation. Indeed, equations for algebraicity (i.e. $\gg\!=$ distributing leftwards) are usually left implicit in the setting of algebraic theories. Due to their different nature, equations involving $\gg\!=$ should thus be treated with some care.

Remark 6.4.4. A similar situation arises when one considers whether $\gg\!=$ distributes rightwards over $\sqcap$:

$$xm \gg\!= \lambda x \longrightarrow f\ x\ \sqcap\ g\ x = (xm \gg\!= f)\ \sqcap\ (xm \gg\!= g)$$

for $xm :: m\ \ a$ and $f, g :: a \longrightarrow m\ \ b$. Again, this holds in the setting MonadNondet m—in particular, $\sqcap$ plainly commutes with itself, by associativity and symmetry. But for the same reason as Corollary 6.4.3, this equation presents problems when $m ::$ MonadProbNondet.

More generally, such an equation is undesired when one does not preclude an "angelic" interpretation of nondeterminism—that is, one that will always explore all choices—in conjunction with effects of a non-idempotent nature in m [**18**]. For instance, if m includes stateful effects such as writing, an angelic choice may invoke a single write on the left (hand side), but two on the right. But note that in the presence of MonadPlus (rather than MonadNondet) choice $\oplus$, a *backtracking* interpretation is usually preferred, matching the behaviour of the composite monad StateT s []. In such case we do have distributivity of $\gg=$ over $\oplus$ rightwards (as well as leftwards), for commutativity (or in the language of Lawvere theories, tensor product) between choice and stateful operations is indeed desirable.

6.4.2. Diagrammatic model

In light of the foregoing discussion, and motivated by the formal development of Section 6.3, as in [**1**] we develop a geometric interpretation of combined choice (based on its free models) as a means to better visualise the equations of the theory.

We will consider computations over a three-valued type of "outcomes" x, y, z. This keeps the exposition simple, and is also the most natural setting for reasoning with points, lines and polygons. From Section 6.1, the simplex $\Delta(3)$ is a convex subset of 3-dimensional space $\mathbb{R}^3$. Specifically, it amounts to vectors (x, y, z) in the intersection of the non-negative octant $0 \leqslant x, y, z \leqslant 1$ with the plane $x + y + z = 1$, where we identify the outcomes with the standard basis vectors of $\mathbb{R}^3$—see Figure 6.2. Thus we interpret free probabilistic computations (over three outcomes) as points in this bounded plane. In general, free probabilistic computations over $n + 1$ outcomes occupy points in a n-dimensional hyper-plane.

From Section 6.1.1, the free convex combination is given by (the restriction of) the usual linear combination structure of scalar multiplication and vector addition in $\mathbb{R}^3$. This motivates the following.

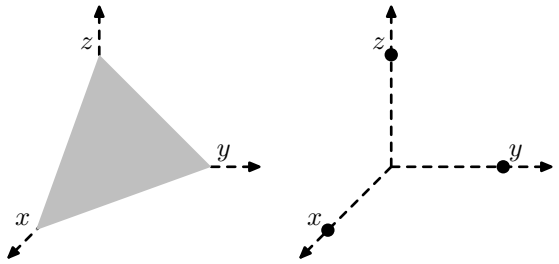


FIGURE 6.2. Probabilistic computations occupy points on the triangular plane; the three pure computations are as indicated

Definition 6.4.5. Let d, e be free probabilistic computations, represented as points in the geometric model[4]. Then for $\lambda \in [0,1]$, probabilistic choice $d +_\lambda e$ is represented by the weighted sum of vectors $\lambda d + \bar{\lambda} e$.

Of course, every free probabilistic computation arises as a convex combination of the basis vectors. For example, $x +_{1/3} y$ is interpreted by Definition 6.4.5 as $\frac{1}{3}x + \frac{2}{3}y = (1/3, 2/3, 0)$ i.e. in Figure 6.3, the point $1/3$ of the way (along the dashed line) from y to x as shown.

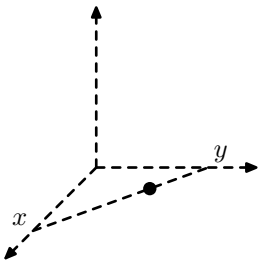


FIGURE 6.3. Probabilistic choice $x +_{1/3} y$ as weighted sum of basis vectors

Now consider free computations of combined choice, that is, of the geometrically convex monad $\mathscr{P}_\Delta$. From Definition 6.3.5, such computations lie in $(\mathcal{P}_+\Delta)(3)$. Since free probabilistic computations in $\Delta(3)$ are represented in the geometric model as *points* in the triangular plane xyz, free combined choice computations are thus *convex sets* of such points. In other

[4]In the terminology of Definition 6.1.6, computations of the Giry monad $\mathscr{G}$ over x, y, z.

words, convex polygons. We recall earlier definitions of probabilistic and nondeterministic choice in this setting from Section 6.3.

Definition 6.4.6. For computations p, q of free combined choice represented as convex polygons, nondeterministic choice $p \sqcap q$ is the convex hull of their union. Probabilistic choice $p +_\lambda q$ is defined point-wise, i.e. $\{d +_\lambda e \mid d \in p, e \in q\}$.

As a simple example, nondeterministic choice of the pure computations x, y (we elide brackets when denoting singleton sets) is represented by the straight line connecting x and y, as shown in Figure 6.4. More generally when taking nondeterministic choice of convex polygons, the notion of convex hull and union are well-understood. Example 6.4.7 illustrates the definition of probabilistic choice for convex polygons.

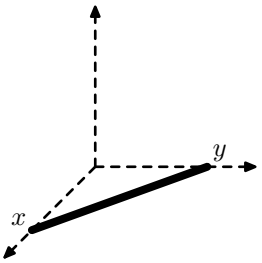


FIGURE 6.4. Nondeterministic choice $x \sqcap y$ is a straight line connecting the points

Example 6.4.7. Let m, n be computations of free combined choice, represented respectively as the triangle and diamond in Figure 6.5. Probabilistic choice $m +_{3/5} n$ is given by the (convex) set of weighted sums $d +_{3/5} e$, for each $d \in m, e \in n$. Indeed by taking convex hull, it is sufficient to consider only the vertices of m, n. For instance, for a vertex of m as marked, we take its weighted sum with each of the four vertices of n, thereby projecting an image of n a fraction $3/5$ of the way. Similarly with the other two vertices of m, we have thus three projections of n, whose convex hull is the heptagon as shown.

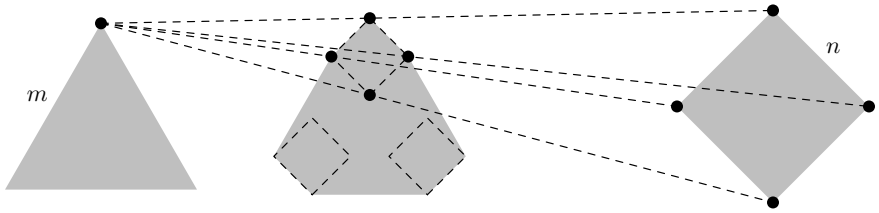


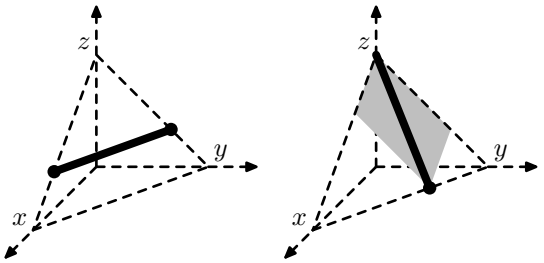
FIGURE 6.5. Probabilistic choice $m +_{3/5} n$ of convex polygons

With this simple geometric model, we begin to explore the potential for diagrammatic reasoning. Recall (e.g. Section 6.3) that the theory of combined choice is axiomatised by distributivity of probabilistic over nondeterministic choice, but not the other direction. We justify this informally with simple examples.

Example 6.4.8. Consider the computation $(x \sqcap y) +_{2/3} z$. Since $x \sqcap y$ is the line shown in Figure 6.4, taking the weighted sum of every point in this line with z yields a line $2/3$ of the way from z, as in Figure 6.6(a). By distributivity, we expect this to be the same as $(x +_{2/3} z) \sqcap (y +_{2/3} z)$. By definition of $\sqcap$ this is the line connecting the points $x +_{2/3} z$ and $y +_{2/3} z$, and indeed they are precisely the endpoints of the line in Figure 6.6(a).

Example 6.4.9. Let us seek a diagrammatic counter-example to distributivity of nondeterministic over probabilistic choice. Consider Figure 6.6(b). The point $x +_{1/3} y$ is as indicated at the base of the triangle, and its nondeterministic choice with z is the line connecting it with z as shown. We should not expect this to be the same as the right hand side, and indeed it is not—the lines $x \sqcap z$ and $y \sqcap z$ are the upper edges of the triangle, and their weighted sum is the entire shaded parallelogram as shown.

The geometric interpretation of the operation $\gg=$ is somewhat more complicated—let us start by recalling from Section 6.3.2 the multiplication M for the geometrically convex monad $\mathscr{P}_\Delta$, from which (together with Definition 6.3.5, the action of its functor $\mathcal{P}\Delta$) one derives $\gg=$ using the



(a) (b)

FIGURE 6.6. (a) Distributivity $(x \sqcap y) +_{2/3} z = (x +_{2/3} z) \sqcap (y +_{2/3} z)$ (b) Non-distributivity $(x +_{1/3} y) \sqcap z \neq (x \sqcap z) +_{1/3} (y \sqcap z)$

well-known Haskell equality

$$(-) \gg\!= f = join \circ fmap\ f$$

where *join* is taken to be M. Specifically, for a Kleisli map $f : X \to (\mathcal{P}\Delta)(Y)$ this is the composite map $M_Y \circ (\mathcal{P}\Delta)(f)$ given by

$$\begin{array}{ccc} (\mathcal{P}\Delta)(X) & & \left(\sum_i \lambda_{i,k}\ \underline{x}_i\right)_k \\ \Big\downarrow & & \Big\downarrow \\ (\mathcal{P}\Delta)(Y) & & \left\lceil + \right\rceil_k \left(\bigoplus_i \lambda_{i,k} f(x_i)\right) \end{array}$$

In our particular geometric setting $X = Y = 3$, we specialise this even further, as in the following.

Definition 6.4.10. Monadic bind $(-) \gg\!= f$ for the geometrically convex monad $\mathscr{P}_\Delta$ over x, y, z is given by the map

$$(\lambda_k x + \theta_k y + \gamma_k z)_k \longmapsto \left\lceil + \right\rceil_k \lambda_k f(x) \oplus \theta_k f(y) \oplus \gamma_k f(z)$$

with $\lambda + \theta + \gamma = 1$. In more words, let m be a convex polygon. $m \gg\!= f$ considers each vertex k of m (for it is sufficient to consider only the vertices) as a weighted sum of the bases x, y, z. Their images under f (i.e. the triple $f(x), f(y), f(z)$ of convex polygons) are in turn convex-combined with the same weights. Finally, convex hull is taken over all such convex combinations.

CHAPTER 7

Conclusion

Let us begin by summarising the main results of this dissertation. In Chapter 4 we began to explore what we believe to be an open question, about the relationship between distributive tensors (of Lawvere theories) and distributive laws (of monads). There, we gave an account of a simple example of a distributive law (the list monad over the bag monad) which turned out to coincide with the distributive tensor of the respective theories. We also showed that a distributive tensor of the semigroup over unit theory is equivalent to the free *semigroup with zero* monad. This leads to the question of *when* the two notions—distributive tensor and distributive law—exhibit such coincidence generally. We have more thoughts on this in Section 7.2.

In Chapter 5, we gave a number of correspondence results based on the observation that the essential structure of various combinatorial search strategies (such as depth-first and breadth-first) can be captured by two equivalent formulations—as *bunch* monadic types, and as more structured theories of monoids. Specifically in the latter, we add appropriate equations to the theory of monoids MON with a unary operation WRAP, thereby specifying their interaction. As such, we showed that a bunch monad of *forests* arises from taking the sum of MON and WRAP i.e. without adding new equations. We gave two proofs of this—first by considering the free models of the sum-theory, the other by application of a theorem about the generalised resumptions monad transformer. While the list monad (characterising depth-first traversal) arises by discarding WRAP, other bunch types satisfy equations given by *adding* interaction between the two theories. In the case of the DBound bunch monadic type, such values were shown to be

models of a distributive tensor of MON over WRAP. We considered also the free models of this theory, giving rise to a monad of *fences*. Furthermore, by imposing symmetry on the monoid operation, we obtained a distributive tensor theory that matches breadth-first traversal closely, and while we showed that the Matrix bunch type is not quite a model of this theory, we proposed a refinement InfMatrix that is. Again we considered the free models, giving rise to a monad of *bundles*.

The remainder of the chapter shifted focus to list computations, where the (`mtl`) list monad transformer was critiqued with respect to its application to practical stream programming. The main contribution here was a novel characterisation of this transformer as a distributive tensor from MON. This is in contrast to the equational presentation of the alternative, "done right" list transformer, whose difference we clarified: while it too exhibits distributivity of the monoid operation, it does so crucially only in the leftwards direction and not the right.

Contained in Chapter 6 is a detailed derivation of the *geometrically convex* monad—the combination of probabilistic and nondeterministic choice effects, which we call *combined choice*—in a relatively simpler setting than the usual presentation of it in the (mostly domain theoretic semantics) literature. As such, its characterisation as a distributive tensor is clearer. Although this particular characterisation has been mentioned before in passing [**27**], we do not believe that our main correspondence result (Theorem 6.3.7) is widely known.

Following on from this, the focus shifted to equational reasoning in Haskell (or a similar functional programming setting). Under this lens, we identified an incorrect assumption in some of the literature on the equational axiomatisations of various algebraic effects such as probabilistic choice. Specifically, it was thought that monadic bind $\gg=$ distributes over probabilistic choice, both in the leftward and rightward directions. It turns out the latter is a far more contentious property than the former (which just states

algebraicity of the operation), particularly if one assumes that such properties continue to hold without fuss when combined with other effects. For example, Section 6.4 considered once more the algebraic effect of combined choice, where we have both probabilistic and nondeterministic choice operations. But then we showed that rightwards distributivity of $\gg=$ over probabilistic choice implies its commutativity with *any* operation, in particular nondeterministic choice. But this contradicts the usual equations of combined choice—in fact we showed that in such case, the probabilistic choice operations collapse. Note that we also give a direct equational proof of this result in [**1**].

Since it can be difficult to get the equational properties of the interaction between probabilistic and nondeterministic choice right, in the remainder of the chapter we developed a technique for reasoning about such equations visually, by taking a geometric interpretation of the free models of combined choice. That is, we interpret computations of the geometrically convex monad (over three values) as convex polygons on a plane. We illustrated the use of this diagrammatic model by exhibiting an inequality (specifically, non-distributivity of nondeterministic over probabilistic choice). Compared to our earlier exposition of the work on diagrammatic reasoning [**1**], we gave a more detailed treatment here, and in addition, described the interpretation of $\gg=$.

7.1. Concluding remarks

In this dissertation we have paid particular attention to a relatively lesser-known construction on Lawvere theories, the distributive tensor. In this section we provide some commentary on the relevance of the results discussed above, as well as the wider role that the distributive tensor plays in the theory of combining computational effects. In a seminal paper on combining effects [**26**], it was remarked that sum and (commutative) tensor account for most of the examples in which effects are combined in practice.

For example, they respectively explain the exception and state monad transformers in a mathematically precise way, in terms of the *interaction* between operations of two theories. That said, while sum and tensor do characterise many monad transformers of practical interest, the overall picture is more nuanced than the above remark might suggest. As a case in point, in this dissertation we studied the list (or *backtracking search*) transformer. In Section 5.4 we showed that ListT_0, one form of this transformer, is in fact characterised as a distributive tensor. Furthermore, the variant ListT (as in [**29**]) is not any of the three constructions. Instead it could be described crudely as "half" a distributive tensor, since it insists on distributivity of the monoid operation only in the *leftwards* direction. Thus it is interesting to note that it is the *absence* of some positions of distributivity (a point that we return to later) that largely results in, for most practical applications, a more capable transformer for backtracking effects.

Backtracking in the list monad, it turns out, follows a depth-first traversal strategy. Earlier in Chapter 5 we investigated a class of bunch monadic types representing the traversals of other kinds of search strategy. The various correspondence results that we established, between a bunch type and an equational theory, are mostly of theoretical interest. From them, we have an understanding of the monads that capture the most essential structure of the search strategy in question. From a more practical viewpoint, since these monads represent the idealised form of a search space traversed with a particular strategy in mind, we can write functions on such abstract data types accordingly, without needing to know the concrete representation. For example, if we know our input data has been traversed and organised in a breadth-first order that we wish to preserve, but not its concrete data type, it makes sense for our function to operate on Bundle values. In this particular scenario, the provider of the data can decouple the specific type from the client. Provided it is a bunch monadic type, reconciling with bundle values is a simple matter of giving a mediating bunch morphism.

The other application of distributive tensor that we studied in some depth in this dissertation is the combination of probabilistic and nondeterministic effects. Both in isolation are standard examples of effects, but in some problem domains it is important for them to be used in concert (i.e. what we have called *combined choice*)—for example, to model probabilistic systems that depend on nondeterministic inputs. Our main result, establishing the equivalence between the geometrically convex monad and the theory of combined choice, is of theoretical interest. Although mentioned in [**27**], we do not believe this characterisation as a distributive tensor is very widely known. That said, soon after proving this result, we discovered that a similar one has been shown in Beaulieu's thesis [**7**]. Nevertheless, our presentation is somewhat different. While Beaulieu preferred to establish this equivalence "directly", we instead found it insightful to give due emphasis on the actions of the various adjoint functors at play—the equivalence itself is then merely an immediate consequence by appeal to monadicity.

But there is also practical relevance in the work on combined choice, as we began to explore in our technique for reasoning about such equations with diagrams—specifically, convex polygons. We believe a visual tool is of benefit to reasoning about the equational properties of programs with combined choice effects, if only because—due to their non-trivial interaction—it can be notoriously difficult to get such equations right. Indeed, had such a tool been known at the time, it might have helped prevent earlier erroneous axiomatisations of probabilistic choice in the literature—specifically, the assumption that monadic bind $\gg=$ distributed rightwards over it. Having said that, our discussion of this equation (or rather, non-equation) shows that properties involving $\gg=$ can be subtle and requires some care, especially when moving from one monad to a "larger" one (e.g. via a monad transformer) as the scope of $\gg=$ is enlarged. This also strongly suggests that

such distributive properties about $\gg\!=$ (whether this direction, or algebraicity) should be considered differently to usual "algebraic" equations when giving an equational theory of a type class, say.

7.2. Further work

We end with some final remarks, and directions for future work. In the applications chapters, we note that in our investigations of monad-theory correspondences of search strategies, we accounted for depth-first, breadth-first and depth-bounded traversals. Specifically for breadth-first, we proposed a refinement bunch type InfMatrix, but note that it is only one of several such refinements that could reasonably be made models of WRAP $\triangleright$ CMON, and it would be worth considering the others. More broadly speaking, it would also be interesting to explore heuristic (e.g. A*) search in terms of algebraic effects, and beyond this, search algorithms from planning and reinforcement learning communities. And while we established a main correspondence result in probabilistic and nondeterministic effects, it remains further work to investigate the limits of the diagrammatic framework for more serious reasoning about properties of combined choice programs, especially those of reasonable size and complexity—the definition of $\gg\!=$ for example, is currently not very straightforward to interpret visually. One would also hope in later work to establish connections with results in probabilistic completions of nondeterminism [**6**], that is, the "dual" situation in terms of the composite adjunction involved. Again in broader terms, we believe the combination of two instances of nondeterminism (e.g. [**45, 44**]), characterising two-player games and protocols, has many parallels that are worth studying. For example, its model as an "upwards-closed" set of sets strongly resembles (and is perhaps a specialisation of) convex-closure—as in the model of combined choice. Equationally, it has also been claimed [**27**] (and is very likely) to be a *bi-distributive* tensor, although it seems not to have been studied much in this way thus far.

We established a result about the list monad transformer: ListT_0 corresponds to a distributive tensor from MON. As we saw however, the other variant ListT is not—in fact, it is none of the known constructions on Lawvere theories that we have considered in this dissertation, for it insists on distributivity of the monoid operation in the first argument but not the second. It remains to be seen whether a novel variant of the notion of distributive tensor to account for this proves to be fruitful, but it would appear that such a notion would probably be difficult to generalise, since it implies flexibility in the choice of positions that distribute and it is difficult to see how that could be easily achieved. The general theory would presumably need some means of "singling out" positions, for which currently there is no obvious facility for. A related question is whether pairs of operations themselves may be singled out for distributivity, rather than insisting that all do. In a similar vein for the commutative tensor, one may conceivably want commutativity in some selective pairs of operations, or none in others. For example, one might not want one constant commuting with another, since doing so would collapse them together. Again, this flexibility appears difficult (if not impossible) to accommodate, since it suggests singling out particular operations—recall that in a Lawvere theory, all operations are treated uniformly, in some sense. This is perhaps indicative of a wider point about the limitations of sum and tensor—they could be described crudely as "all or nothing" constructions. But as we have seen, in practice there are computational effects involving operations that interact in more ad hoc ways (with both ListT and "cut" from logic programming as examples in Section 5.4) that do not quite fit the classification of commutativity / distributivity in every position.

Of course, the very fact that sum and tensors are defined in such natural ways makes them broadly applicable—indeed, they always exist. By contrast, distributive laws (hence composite monads) do not always exist (e.g. between the distribution and set monads [**67**], in either direction), but are a

more flexible notion—after all, there may well be more than one distributive law between two monads. That is of course, assuming one exists at all—but when it does, the evidence would appear to suggest that a canonical one matches the description given by distributive tensor, at least when expressed in terms of reduction rules from rewriting theory [**4**]. We conjecture that recursively defined distributive laws arising from commutative monads and Kleisli strength [**42, 43**] coincide with distributive tensors of their respective theories. As discussed in Chapter 4, the precise relationship between the two notions does not appear to have been studied much, and remains as future work. It is also worth noting that work on *composite Lawvere theories* [**12**] reconciles the two notions of Lawvere theory and distributive law, but while of much theoretical interest, its relevance to the more concrete problems of combining effects requires further investigation. This work does, however, reinforce the idea that in comparison to distributive tensors, distributive laws are a more flexible notion. Likewise, while the characterisation of distributive tensor in terms of models [**27**] appeared promising for simplifying correspondence proofs in this dissertation, at least in the author's experience, its highly technical formulation proved difficult to apply in calculating specific examples. In future work, one would hope (either by refining this formulation, or otherwise) to employ a more modular style to the proofs of monad-theory equivalences.

Bibliography

[1] Faris Abou-Saleh, Kwok-Ho Cheung, and Jeremy Gibbons. Reasoning about probability and nondeterminism. In *POPL workshop on Probabilistic Programming Semantics*, January 2016.

[2] Jiri Adámek, Stefan Milius, Nathan Bowler, and Paul B. Levy. Coproducts of monads on Set. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 45–54, Washington, DC, USA, 2012. IEEE Computer Society.

[3] Steven Awodey and Andrej Bauer. Introduction to categorical logic. 2009.

[4] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[5] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[6] Guy Beaulieu. Probabilistic completion of nondeterministic models. *Electronic Notes in Theoretical Computer Science*, 173:67–84, 2007.

[7] Guy Beaulieu. *Probabilistic completion of nondeterministic models*. PhD thesis, University of Ottawa (Canada)., 2008.

[8] Jon Beck. Distributive laws. In B. Eckmann, editor, *Seminar on Triples and Categorical Homology Theory*, volume 80 of *Lecture Notes in Mathematics*, pages 119–140. Springer Berlin Heidelberg, 1969.

[9] Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(04):395–410, 2001.

[10] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 133–144, New York, NY, USA, 2013. ACM.

[11] Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. Technical report, In Proceedinges of the Conference on Category Theory and Computer Science, 1993.

[12] Eugenia Cheng. Distributive laws for Lawvere theories. *arXiv preprint arXiv:1112.3076*, 2011.

[13] Eugenia Cheng. Iterated distributive laws. *Mathematical Proceedings of the Cambridge Philosophical Society*, 150:459–487, 5 2011.

[14] Tobias Fritz. Convex spaces I: Definition and examples. *arXiv preprint arXiv:0903.5522*, 2009.

[15] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 18–27, New York, NY, USA, 1999. ACM.

[16] Neil Ghani and Christoph Lüth. Composing monads using coproducts. *Intl. Conference on Functional Programming 2002*, 37(9):133–144, 2002.

[17] Neil Ghani and Tarmo Uustalu. Coproducts of ideal monads. *Journal of Theoretical Informatics and Applications*, 38(4):321–342, 2004.

[18] Jeremy Gibbons. Unifying theories of programming with monads. In Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi, editors, *Unifying Theories of Programming*, volume 7681 of *Lecture Notes in Computer Science*, pages 23–67. Springer Berlin Heidelberg, 2013.

[19] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 2–14, New York, NY, USA, 2011. ACM.

[20] Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.

[21] Zhenjiang Hu, John Hughes, and Meng Wang. How functional programming mattered. *National Science Review*, 2(3):349–370, 2015.

[22] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[23] John Hughes. Programming with arrows. In *International School on Advanced Functional Programming*, pages 73–129. Springer, 2004.

[24] Graham Hutton and Diana Fulger. Reasoning about effects: Seeing the wood through the trees. In *Proceedings of the Ninth Symposium on Trends in Functional Programming*, 2008.

[25] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437444, 1998.

[26] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.

[27] Martin Hyland and John Power. Discrete Lawverefd theories and computational effects. *Theoretical Computer Science*, 366(1-2):144–162, 2006. Algebra and Coalgebra in Computer Science.

[28] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electronic Notes in Theoretical Computer Science*, 172:437–458, 2007.

[29] Mauro Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.

[30] Mauro Jaskelioff. Modular monad transformers. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 64–79. Springer Berlin Heidelberg, 2009.

[31] Mauro Jaskelioff. Monatron: An extensible monad transformer library. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 233–248. Springer Berlin Heidelberg, 2011.

[32] Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(5152):4441 – 4466, 2010. European Symposium on Programming 2009 ESOP 2009.

[33] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM.

[34] Steven Keuchel and Tom Schrijvers. Modular monadic reasoning, a (co-) routine. In *Draft Proceedings of the 24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, page 6, 2012.

[35] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, pages 59–70, New York, NY, USA, 2013. ACM.

[36] Anders Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23(1):113–120, 1972.

[37] William F. Lawvere. *Functorial Semantics of Algebraic Theories*. PhD thesis, 2004.

[38] Daan Leijen. Koka: Programming with row-polymorphic effect types. Submitted for publication, 2013.

[39] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[40] Fred E. J. Linton. Some aspects of equational categories. In *Proceedings of the Conference on Categorical Algebra*, pages 84–94. Springer, 1966.

[41] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1978.

[42] Ernie Manes and Philip Mulry. Monad compositions I: general constructions and recursive distributive laws. *Theory and Applications of Categories*, 18(7):172–208, 2007.

[43] Ernie Manes and Philip Mulry. Monad compositions II: Kleisli strength. *Mathematical Structures in Computer Science*, 18(3):613–643, 2008.

[44] C.E. Martin and S.A. Curtis. The algebra of multirelations. *Mathematical Structures in Computer Science*, 23(3):635674, 2013.

[45] C.E. Martin, S.A. Curtis, and I. Rewitzky. Modelling angelic and demonic nondeterminism with multirelations. *Science of Computer Programming*, 65(2):140 – 158, 2007. Special Issue dedicated to selected papers from the conference of program construction 2004 (MPC 2004).

[46] Conor McBride. The Frank manual. Unpublished manual, 2012.

[47] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(01):1–13, 2008.

[48] Michael Mislove, Joël Ouaknine, and James Worrell. Axioms for probability and nondeterminism. *Electronic Notes in Theoretical Computer Science*, 96:7–28, 2004.

[49] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[50] Eugenio Moggi. *An abstract view of programming languages*. University of Edinburgh. Laboratory for Foundations of Computer Science, 1990.

[51] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[52] Philip Mulry. Notions of monad strength. In Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff, editors, *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday,* Manhattan, Kansas, USA, 19-20th September 2013, volume 129 of *Electronic Proceedings in Theoretical Computer Science*, pages 67–83. Open Publishing Association, 2013.

[53] Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.

[54] Maciej Piróg. Eilenberg-Moore monoids and backtracking monad transformers. In *Proceedings 6th Workshop on Mathematically Structured Functional Programming*, volume 207, pages 23–56. Open Publishing Association, 2016.

[55] Andrew M. Pitts. Categorical logic. Technical report, University of Cambridge, Computer Laboratory, 1995.

[56] Gordon Plotkin and John Power. Notions of computation determine monads. In *Proc. FOSSACS 2002, Lecture Notes in Computer Science 2303*, pages 342–356. Springer, 2002.

[57] Gordon D Plotkin and Klaus Keimel. Mixed powerdomains for probability and nondeterminism. *Logical Methods in Computer Science*, 13, 2017.

[58] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[59] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[60] Matija Pretnar and Gordon D Plotkin. Handling algebraic effects. *Logical Methods in Computer Science*, 9, 2013.

[61] Edmund Robinson. Variations on algebra: Monadicity and generalisations of equational theories. *Formal Aspects of Computing*, 13(3):308–326, 2002.

[62] Michael J. Spivey. Algebras for combinatorial search. *Journal of Functional Programming*, 19(3-4):469–487, 2009.

[63] Sam Staton. An algebraic presentation of predicate logic. In *Proceedings of the 16th international conference on Foundations of Software Science and Computation Structures*, FOSSACS'13, pages 401–417, Berlin, Heidelberg, 2013. Springer-Verlag.

[64] Sam Staton. Instances of computational effects: an algebraic perspective. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 519–519. IEEE, 2013.

[65] Bruce A. Tate. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Pragmatic Bookshelf, 1st edition, 2010.

[66] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[67] Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical. Structures in Comp. Sci.*, 16(1):87–113, February 2006.

[68] Philip Wadler. How to replace failure by a list of successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[69] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM, 1990.

[70] Gavin C. Wraith. *Algebraic theories*. Number 22. Aarhus universitet, Matematisk institut, 1970.

APPENDIX A

Algebraic effects for programming

Popularised by the Haskell programming language, monads have become an abstraction tool useful not only to language designers and semanticists, but also programmers. In this chapter, we give some evidence of the claim that algebraic theories too, have much to offer to programmers. The central concept here is that of a *handler* of algebraic effects.

A.1. Handling effects

The recent activity in the research area of effects and handlers have manifested into several compelling languages worthy of study. First however, let us connect the notion of handler to the background material on algebraic theories. Since exception handling is pervasive in programming, it will serve as our starting point in motivating a more formal explanation of handler.

This section reviews the introductory sections of Pretnar & Plotkin [**60**], in more explicit detail. Recall the computational effect of *exceptions*. As an algebraic theory, it is a signature comprising a family $(\texttt{raise}_e)_{e \in E}$ of nullary operators called `raise`, indexed by a set E of exceptions, with no axioms. Alternatively, it is conventional to consider a *single* operator $\texttt{raise} : 0 \to E$, with 0 and E denoting respectively its arity and coarity. Intuitively, `raise` is nullary because we expect for there to be zero choices of continuation following the raising of an exception. Now let Σ_E denote the theory of exceptions, and Σ_E-**Alg** its category of (set-theoretic) algebras. Evidently, there is a forgetful functor $U : \Sigma_E\text{-}\mathbf{Alg} \to \mathbf{Sets}$ given by $(Y, -) \mapsto Y$. Its left adjoint (the free algebra functor) F sends a set X to the algebra

$$(X + E, [\![\texttt{raise}]\!] = \mathsf{inr} : E \to X + E)$$

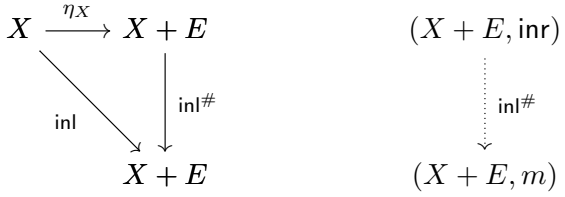
with carrier $X+E$ and interpreting function $[\![\texttt{raise}]\!]$ given by the coproduct injection inr.

The monad induced by this adjunction, i.e. with functor part $UF = - + E$ (and unit η with components given by inl) is the usual Moggi monad of exceptions. Indeed, since there are no axioms associated with the theory of exceptions, we may view Σ_E as the constant signature functor $- \mapsto E$, and then this monad is precisely the free monad Σ_E^* over it.

As well as an operation to raise exceptions, many programming languages allow one to "catch" and handle raised exceptions, e.g. this could mean logging the exception, returning a value, or aborting the program. We can imagine such an operation $\texttt{handle}_e(K, L)$ that continues with the computation K, unless it raises an exception e—in which case it proceeds with L. Unfortunately, it can be shown that such an operation is not *algebraic*—it turns out that `handle` lacks a certain naturality property, and so it must be treated differently.

A.1.1. Exception Handling

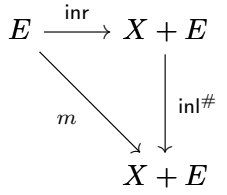
To explain the true nature of exception handlers, let us continue our analysis of the theory of exceptions. Consider a Σ_E-algebra $(X + E, m)$. Noting that this has the same carrier as the free Σ_E-algebra FX over the set X, with universal arrow η_X, there is a unique extension of the coproduct injection inl such that the triangle below on the left commutes:



In programming terms, we can think of the interpreting function $m : E \to X + E$ as an exception handler that a programmer has defined, where a handling computation $M_e \in X + E$ is given for each exception e. Under this reading, $\mathsf{inl}^\#$ gives the semantics of the handling construct: it takes a

computation, and interprets it such that a $\texttt{raise}_e()$ is instead replaced by the computation M_e determined by m.

More precisely, the commutative triangle reflects one of the equational properties of the handling semantics—that interpreting a computation that is a pure value against our handler is the same as interpreting just the pure computation alone. For the other equational property, first observe that as $\mathsf{inl}^\#$ is a homomorphism of Σ_E-algebras, we also have the following commuting triangle:



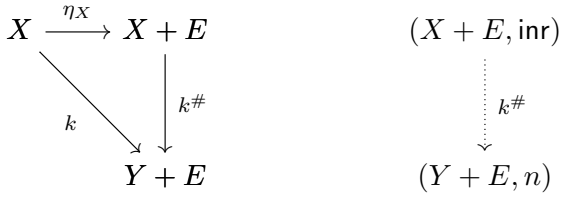
which confirms that interpreting a $\texttt{raise}_e()$ operation with our handler semantics is the same as as the computation M_e as determined by our exception handler m. In sum, an exception handler is modelled by a Σ_E-algebra with the same carrier as that of the free algebra, where the semantics of handling is given by the homomorphism $\mathsf{inl}^\#$ between them.

A.1.2. Generalised Handlers

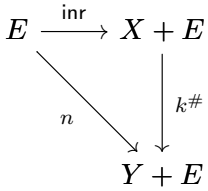
Let us now see how the generalised exception handlers of Benton and Kennedy [**9**] fit into our analysis of exceptions. Notice that for simple exception handlers of section A.1.1, algebras m have the same carrier as the free algebra, and the induced homomorphism is required to be the extension of inl, ensuring that computations are handled in a value-preserving way. Abstractly of course, the universal property of free objects does not strictly require this to be so; hence let us lift this restriction somewhat.

Take any Σ_E-algebra $(Y+E, n)$ and a function $k : X \to Y+E$. Again, we may think of $n : E \to Y + E$ as a user-defined exception handler, assigning to each exception e a computation N_e that either returns a value in the (possibly new) set Y, or again raises an exception. The map k can be read

as a user-defined continuation in $Y + E$ that takes a value in X as parameter. By the universal property, we have a unique homomorphism $k^{\#}$ of algebras such that the triangle below on the left commutes:



Again, $k^{\#}$ defines the semantics of this generalised exception handling construct: the commutative triangle reflects the equational property that a handled pure computation of a value x is interpreted in the same way as the continuation k on x, and since $k^{\#}$ is a homomorphism of algebras:



we have the other equational property that a $\texttt{raise}_e()$ operation handled with respect to n is interpreted in the same way as N_e.

What we have seen here is an explanation of the notion of a generalised exception handler, but in fact this construction extends to arbitrary algebraic effects—not only exceptions. Whereas raised exceptions were replaced with handling computations, in any algebraic theory we may give similar handling semantics to each of its operations. This generalised, extended notion of handler is the one we refer to.

A.2. Algebraic effect libraries: a short survey

The view of computational effects as algebraic theories offers a compelling alternative to the more familiar monadic view. For example, studying effects in this setting has shown how one may combine effects together

[**26**] in ways distinct to the usual "monad transformer stack" method, avoiding issues of compositionality often cited when combining monadic effects. With the introduction of handlers, the situation becomes yet more interesting, as they open the door to the design of languages with new programming abstractions centred around this view of effects.

In this section we will survey some of these languages, and focus on a particular few in slightly more depth[1]. I will assume the reader has seen functional programs before. Rather than give a formal account of language specifications however, for the purposes of comparing the salient ideas of each language, I opt for a more "Bruce Tate" [**65**] inspired style of exposition, motivated by small examples involving exceptions, state and nondeterminism: three very familiar effects that have been studied previously in my term paper. The point here is not to study any one language in depth, rather to see how *several* relate to each other, and perhaps most pertinently, how they relate to the theory.

A.2.1. eff

The language EFF [**5**] was the first to explore the ideas of algebraic effects and handlers for practical programming. Although the syntax resembles ML (intentionally—a design choice of the authors), it is an entirely standalone language designed by Bauer and Pretnar (the latter of course one of the pioneers of handlers [**60**]). As a first example of EFF programming, let us once again look at how the exceptions effect is treated:

```
  type 'a exception = effect  operation raise : '
a ⟶ empty
  end
```

[1]Due to the novel and experimental nature of these languages, the choice of which to focus was determined mostly by the amount of literature / documentation that happened to be available at the time of writing.

The `effect` ... `end` construct forms an *effect type*, which we think of abstractly as a signature. In this case, `exception`, parameterised in a type `a`, comprises a single operation `raise`. Notice that `raise` is defined in the form of a generic effect [**58**], in keeping with Plotkin-style presentation. Other than that, this corresponds exactly to the familiar algebraic theory of exceptions analysed in section A.1.

We can define a handler for exceptions using the `handler` ... syntax—e.g. let us say we are not interested in the particular exception that was raised, only the value returned; then we may craft a handler `optionalise` that converts a computation that may raise exceptions into one that optionally returns a value:

let *optionalise e* = *handler*
  | *e* # *raise* _ _ ⟶ None
  | *val x* ⟶ Some *x*

A `raise` operation is evaluated to `None`, regardless of the particular exception in `a`, and makes no use of the continuation (indeed, as discussed in section A.1.1, it cannot)—hence they are both unused parameters. Pure values `x` on the other hand, are returned as-is inside a `Some` wrapper.

The `e` parameterises over *effect instances* (in this case, of `exception`), so that it is possible to handle (possibly in a non-uniform way) *multiple* instances of a single effect within a computation. One creates an effect instance with the operator `new`, and `with` ... `handle` attaches a handler to a computation:

let *ex* = *new exception*
with optionalise ex handle
  (∗*computation*∗)

Let us now look at another example where the continuation *is* used. Consider the theory of (binary) nondeterminism—its signature consists solely of

the algebraic operation $\vee : 2 \to 1$, and represented as the following effect type `choice` in EFF:

```
type choice = effect
  operation decide : unit ⟶ bool
end
```

We will define a handler `all` that interprets the `decide` operation by collecting all possible results into a list. Under this interpretation, a single value is of course transformed into a singleton list:

```
let all c = handler
  | c # decide () k ⟶ k true @@ k false
  | val x ⟶ [x]
```

Notice the continuation `k` is called not just once but *twice*, to get both outcomes that is then concatenated together with @. As an example of its usage, consider the following handled computation:

```
let ch = new choice
with all ch handle
  let x = (if ch # decide () then 10 else 20) in
  let y = (if ch # decide () then 33 else 66) in
    x + y
```

By giving the abstract computation (i.e. the last three lines) the handling semantics of `all`, the result is the list `[43;76;53;86]` of *all possible results* from all choices of the `decide` operation! Equally, we could instead assign a handler that interpreted `decide` to always choose `true`, and this would yield an arguably more natural-looking result of just `43`.

As a final case study for EFF, we look at the computational effect of state, which has an operation for looking up a stateful value, and one for updating it:

```
type 'a state = effect
operation lookup : unit -> 'a
operation update : 'a -> unit
end
```

A handler for `state` that reflects the behaviour of references in ML, or a "reader" monad in Haskell, threads the state around a computation by interpretations as functions taking the state as parameter. The return clause is simply the constant function returning the value. The interpreting functions for `lookup` and `update` return, respectively, the continuation parameterised by the state (without changing the state itself), and the continuation with the state updated:

$$
\begin{aligned}
&\textbf{let } \mathit{reader}\ r\ \mathit{init} = \mathit{handler} \\
&\quad \mid \mathit{val}\ y \longrightarrow (\mathit{fun}\ s \longrightarrow y) \\
&\quad \mid r \,\#\, \mathit{lookup}\ ()\ k \longrightarrow (\mathit{fun}\ s \longrightarrow k\ s\ s) \\
&\quad \mid r \,\#\, \mathit{update}\ s'\ k \longrightarrow (\mathit{fun}\ s \longrightarrow k\ ()\ s') \\
&\quad \mid \mathit{finally}\ f \longrightarrow f\ \mathit{init}
\end{aligned}
$$

Note that `reader` takes an additional parameter `init` for the initial value of the state, and there is a `finally` clause. Such a clause, when present, performs one final transformation on the whole handled computation. In this case, the computation—interpreted as one big state function `f`—is applied to the supplied initial state.

Remark A.2.1. Although the handler implementation `reader` serves its purpose, there is an unfortunate consequence. Namely, if an instance of `state` falls outside the scope of its `reader` handler, its behaviour will be undefined. Hence Bauer and Pretnar prefer in this case an alternative idiom that, in some sense, gives a "default" behaviour to effect instances in the absence of an appropriate handler:

```
let ref init =
  new state @@ init with
    operation lookup () @@ s ⟶ (s, s)
    operation update s' @@ _ ⟶ ((), s')
  end
```

This uses an alternative form of the `new` operator that, as well as creating an effect instance, associates with it a *resource* (an idea based on the theory of "comodels" which we will not explore here). In this case, the resource is the current state, initially set to `init`, and its evolution in response to `lookup` and `update` are similar to that of the handler above.

A.2.2. hia

Whereas EFF is a completely stand-alone language, HIA is a library[2] designed to extend the capability of Haskell with algebraic effectful programming [**33**]. The ideas in HIA bear resemblance to those in EFF, for one usually starts by defining the abstract operations, e.g. for state:

```
[ operation | Get s :: s |]
[ operation | Put s :: s ⟶ () |]
```

which automatically derive wrappers `get` and `put` (types omitted for brevity) for use within an abstract computation. A simple example of such a computation is the following:

```
comp :: SComp Int Int
comp = do { x ← get; put (x + 1);
  y ← get; put (y + y); get }
```

where `SComp s a` is roughly the type of abstract computations of state type `s` returning values of type `a`. Note the use of conventional Haskell `do` notation,

[2] As the authors do not give an explicit name for their library, for convenience I refer to it here HIA, after the title of their paper.

though the reader is warned that `comp` is a *pure* computation! We may think of it simply as a term in the free algebra over the signature of state. To give meaning to the operations `get` and `put` (and hence to the computation), as for EFF we define a handler for state:

```
[handler | ReaderState s a :: s ⟶ a handles {Get s, Put s} where
  Return x s ⟶ return x
  Get k s ⟶ k s s
  Put s k _ ⟶ k () s |]
```

The anatomy of this handler is completely analogous to that of the `reader` handler in EFF earlier. Again, this generates a convenient wrapper function:

```
readerState :: s ⟶ SComp s a ⟶ a
```

whose first argument is the initial value of the state. To actually run a computation against the handler, we may apply this function to the arguments `1` and `comp`, say, to get a result of `4`.

While the handler above worked fine for `comp`, what if it had other (non-stateful) operations within it? Due to the *closed* nature of the handler, unfortunately such a computation would not be supported by it. A novel feature of HIA that addresses this shortcoming is the notion of an *open* handler, that is, one that interprets the operations that it supports, and forwards those it does not to a *parent* handler, if one has been composed with it. "Opening up" a closed handler, so to speak, is a simple matter of adding some syntactic sugar to its declaration:

```
[handler | forward h ∘
  ReaderState s a :: s ⟶ a   -- (etc. as above)
 |]
```

To illustrate a simple example of composing open handlers, we consider two operations similar to those of nondeterminism and exceptions:

$$[\,\mathit{operation} \mid \mathit{forall}\ a \circ \mathsf{Choose} :: a \longrightarrow a \longrightarrow a \mid]$$
$$[\,\mathit{operation} \mid \mathit{forall}\ a \circ \mathsf{Failure} :: a \mid]$$

The intuition, unsurprisingly, is that `Choose` decides on one of its two arguments, and `Failure` aborts. Assuming we have a computation that performs both operations, we may wish to handle it with a composition of two handlers—one for each operation. Keeping with similar handlers considered for EFF for simplicity:

$$\begin{aligned}
&[\,\mathit{handler} \mid \mathit{forward}\ h \circ \\
&\quad \mathsf{AllResults}\ a :: [a]\ \mathit{handles}\ \{\mathsf{Choose}\}\ \textbf{where} \\
&\qquad \mathsf{Return}\ x \longrightarrow [x] \\
&\qquad \mathsf{Choose}\ x\ y\ k \longrightarrow k\ x +\!\!+ ky \mid] \\
&[\,\mathit{handler} \mid \mathit{forward}\ h \circ \\
&\quad \mathsf{MaybeResult}\ a :: \mathsf{Maybe}\ a\ \mathit{handles}\ \{\mathsf{Failure}\}\ \textbf{where} \\
&\qquad \mathsf{Return}\ x \longrightarrow \mathit{return}\ (\mathsf{Just}\ x) \\
&\qquad \mathsf{Failure}\ k \longrightarrow \mathit{return}\ \mathsf{Nothing} \mid]
\end{aligned}$$

Now we can compose these two handlers together. Taking `CF a` to be the type of abstract computations involving `Choose` and `Failure`, we can handle such a computation with the following:

$$\begin{aligned}
&\mathit{allOrNothing} :: \mathsf{CF}\ a \longrightarrow \mathsf{Maybe}\ [a] \\
&\mathit{allOrNothing} = \mathit{maybeResult} \circ \mathit{allResults}
\end{aligned}$$

The handler `allResults` interprets `choose` to yield a list of all results, forwarding `failure` to `maybeResult`, which in turn translates them into optional (lists of) results.

Remark A.2.2. The example here is of course rather contrived: e.g. there is no reason why `AllResults` cannot also handle `Failure` (a translation into an empty list would be the canonical implementation). The point here

really is to demonstrate, in the simplest way possible, how composition of open handlers work in HIA.

A.2.3. Effects

EFFECTS is another library developed for the purpose of extending an existing language with effects and handlers—this time, the dependently-typed language IDRIS [**10**]. More precisely, EFFECTS is a domain specific language embedded in IDRIS, its implementation of which makes essential use of the host language's dependent types. Our first example is the familiar state effect, whose signature is formulated in EFFECTS as follows:

```
data State : Effect where
  Get : State a a a
  Put : b ⟶ State a b ()
```

Unlike HIA, declaring the algebraic operations of an effect require no special syntax—simply a data type of kind `Effect`, which is a ternary type constructor. Like EFF, EFFECTS makes use of resources, but rather than being an auxiliary notion, they are much more central to the design of the library. For the type `State a b c`, `a` denotes the type of the input resource (in this case, the state itself), `b` the output resource and `c` the actual type of the computation.

Remark A.2.3. While it may seem unusual for the type of the state to change, this flexibility accommodates "resource usage protocols" whereby a resource of *dependent* type $A(s)$ may well transition to another $A(t)$ differing only in the value indexed.

Looking at the operations, `Get` returns the state, without altering it. `Put` on the other hand, returns unit but updates the state to the one given. Observe that if one ignores the resource state parameters, the types are no different to what we have seen already for state operations.

Giving a handler is a similar affair to before, except that in Effects, one does this by defining an instance of the type class `Handler` (consisting of the single method `handle`) parameterised over not only the `Effect` in question but also a computational context—which may be a monad (but not necessarily so). The latter allows context-dependent handlers to be written, e.g. we may want exceptions to be handled differently in an `IO` context compared to a `Maybe` context. For state, we can implement a handler that works uniformly over all contexts `m`:

```
instance Handler State m where
  handle st Get k = k st st
  handle st (Put n) k = k n ()
```

and this is just the implementation we have seen previously. To start writing effectful computations, we require a so-called *concrete* effect (`STATE`), and correspondingly, concrete operations (`get` and `put`) that are "promotable" from the internal algebraic description above via helper functions in the library.

An example of an abstract stateful computation is the following tree-labeling program made famous by Hutton and Fulger [**24**]:

```
data Tree a = Leaf
    | Node (Tree a) a (Tree a)
tag : Tree a ⟶ Eff m [STATE Int] (Tree (Int, a))
tag Leaf = return Leaf
tag (Node l x r) = do l' ← tag l
  lb ← get; put (lb + 1)
  r' ← tag r
  return (Node l' (lb, x) r')
```

In Effects, effectful computations are of type `Eff m es a`, where `m` denotes the computational context that the program will run in, `es` a list of concrete

effects that the program is permitted to use, and `a` is the program's return type. Hence the type signature of `tag` indicates that, given an input tree, it returns a computation that may invoke stateful operations (i.e. `get` and `put`), and returns an appropriately labeled tree.

Remark A.2.4. It is worth mentioning that an effectful computation can be declared to use any mixture of different effects. For example, a program that permits console I/O, threads integer state, and might throw an exception of type `Error` may have a type like the following:

$$\mathit{example} : \mathsf{Eff}\ \mathsf{IO}\ [\mathsf{EXCEPTION}\ \mathsf{Error}, \mathsf{STDIO}, \mathsf{STATE}\ \mathsf{Int}]\ ()$$

What if a program involves multiple instances of a single effect? EFFECTS has a concept of *labeled effects* that serves a similar purpose to effect instances in EFF. For instance, if in addition to labeling a tree, we also hope to maintain a count of nodes as the tree is traversed, we could define our program with a type signature:

$$\begin{aligned}&\textbf{data}\ \mathsf{StateInstances} = \mathsf{Tag} \mid \mathsf{Count}\\ &\mathit{tag} : \mathsf{Tree}\ a \longrightarrow \mathsf{Eff}\ m\ [\mathsf{Tag} ::: \mathsf{STATE}\ \mathsf{Int},\\ &\quad \mathsf{Count} ::: \mathsf{STATE}\ \mathsf{Int}]\ (\mathsf{Tree}\ (\mathsf{Int}, a))\end{aligned}$$

and in the implementation itself, `get` and `put` operations would be prefixed with `Tag` $:-$ or `Count` $:-$ to indicate the particular instance of the effect to invoke the operation on.

Now to actually run the computation, it must be passed as parameter to a "run" function, together with a list of initial values for each resource associated with the effects declared in the computation. As such, it is very similar in spirit to the wrapper functions generated from handlers in HIA, except that the run functions here work generically. The simplest example of such a function is `runPure`, which assumes that the computation is run in the context `id`, i.e. a pure function:

$$tagFrom : \mathsf{Int} \longrightarrow \mathsf{Tree}\ a \longrightarrow \mathsf{Tree}\ (\mathsf{Int}, a)$$
$$tagFrom\ init\ t = runPure\ [init]\ (tag\ t)$$

Since our handler above is defined for all contexts, it certainly supports `id` as a context, and hence this is the one used to run the computation against. Now, the reader may have noticed that this handler definition, unlike EFF and HIA, lacked a "return clause". In fact, in EFFECTS this function gets passed as parameter to the run function, or in some overloads this will not even be necessary if the context in which to run is an applicative functor—since this comes equipped with such an operation (usually called `pure`).

Remark A.2.5. The treatment of exceptions and nondeterminism is similar and not surprising, hence they are omitted here. However, it is worth noting a limitation raised by Brady regarding the latter. Namely, when nondeterminism is combined with other effects, the values of resources for those other effects are reset at the start of each branch. In particular, this implies there can be no sharing of state between branches. The reason this is so can be explained by a certain distributivity law [**19**], which given the absence of equations of any kind, gives rise to this limitation.

A.2.4. Frank & Friends

The previous subsections have shown that while EFF, HIA and EFFECTS each have their own nuanced approach to effectful, algebraic programming, fundamentally they are coherent with the core ideas of the theory. That is, they each have some form of representing the theory of an effect (or at least, its operations), and a notion of handler that interprets operations (thus computations).

Indeed, they are not the only languages to embrace the algebraic approach to effects. Briefly, McBride's FRANK [**46**], for example, is a language

that resembles EFF, but has a more developed effect typing system influenced heavily by Levy's call-by-push-value calculus. It only supports so-called "shallow" handlers, unlike those that we have seen for the languages above (although HIA is able to support shallow handlers too). Another language that brings effects into its type system is KOKA [**38**], which makes use of *row-polymorphic* effect types.

Elsewhere, Staton has shown a characterisation of syntax-invariant *parameterised* algebraic theories [**63**]—a more general form of algebraic theories considered here—in terms of enriched clones. He presents a fragment of predicate logic in this parameterised form, giving rise to a logic programming language. Finally, Kiselyov et al have developed a library [**35**] for Haskell whereby effects arise from interactions between a client and effect handler. While it is used to build a monad to model the appropriate effects, its use of handlers is inspired by the algebraic approach.