JFP **27**, e17, 15 pages, 2017.
doi:10.1017/S0956796817000077

Theoretical Pearl

*Backtracking with cut via a distributive law and left-zero monoids**

MACIEJ PIRÓG
University of Wrocław, Poland
(*e-mail:* maciej.adam.pirog@gmail.com)

SAM STATON
University of Oxford, UK

Abstract

We employ the framework of algebraic effects to augment the list monad with the pruning *cut* operator known from Prolog. We give two descriptions of the resulting monad: as the monad of free left-zero monoids, and as a composition via a distributive law of the list monad and the 'unary idempotent operation' monad. The scope delimiter of *cut* arises as a handler.

1 Introduction

This paper is about the analysis of monads, as used in functional programming, via ideas from algebraic theories and universal algebra. From this perspective, combining monads amounts to combining algebraic theories, and scoping or handling monads amounts to algebraic structures. This is a case study of this algebraic approach as applied to backtracking with cut.

Lists, backtracking, and cuts. In functional programming, especially in languages with lazy evaluation, backtracking computations are often encoded using the list monad, which is extensively illustrated in the literature with solvers for combinatorial puzzles such as Sudoku (Bird, 2006). Unfortunately, the list monad is too simple to allow any control over the course of the computation, such as pruning the search space based on partial results. In this pearl, we show how to enhance the list monad with Prolog's *cut* operator, which makes it possible to discard some yet uninspected choices. The desired behaviour of *cut* in connection with the list monad can be demonstrated with the following Haskell fragment:

* Research supported by a Royal Society University Research Fellowship and EPSRC Grant EP/N007387/1

```
do x <- [100, 200, 300]
   y <- [10, 20, 30]
   if x >= 200 && y >= 20 then cut else return ()
   z <- [1, 2]
   return (x+y+z)
```

The intended result of the program above is `[111, 112, 121, 122, 131, 132, 211, 212, 221, 222]`. Invoking `cut` discards the choice (`x` = 200, `y` = 30) together with the choice (`x` = 300).

It is impossible to define a value `cut :: [()]` with the behaviour discussed above within the list monad. We need a different, enhanced monad to accommodate this functionality. Moreover, this is somewhat useless unless the scope of cut can be delimited, so that a cut in one library call does not cause cuts to unrelated backtracking in another library.

Synopsis. In this pearl, we systematically obtain an enhanced monad that supports both backtracking and cut, by means of algebraic presentations of monads in terms of operations and equations (Plotkin & Power, 2004). The list monad has such an algebraic presentation: the theory of monoids (Section 2). We enhance it with a new operation and some equations capturing the intended semantics of *cut* to build a monad for backtracking with cut.

We show that we can present the resulting theory in two alternative ways: using a left zero (Section 4) and using an idempotent unary operation (Section 6). These two ways lead to two different Haskell implementations of the desired monad by considering normal forms of the theory (Sections 3 and 7). The second way allows us to see the resulting monad as a composition via a distributive law of two simpler monads, one of which is the list monad. Moreover, the second way is better suited to understanding the idea of delimiting the scope of cut, in the context of handlers (Plotkin & Pretnar, 2013) (Sections 5 and 8).

2 Algebraic theories

In this section, we give an overview of equational theories. Some references are Baader & Nipkow (1998, Chapter 3), or Mac Lane (1998) for the categorical aspects, or Plotkin & Power (2004) for the programming languages side.

For our purposes, a *theory* $\mathbb{T}$ consists of a signature $\Sigma^{\mathbb{T}}$ (that is, a non-empty finite set of function symbols with finite arities, which we write in superscript) and a finite set $E^{\mathbb{T}}$ of formal equations between $\Sigma^{\mathbb{T}}$-terms with variables from a countable set $\mathscr{X} = \{x, y, z, \ldots\}$. For example, the theory of monoids Mon can be given as follows:

$$\Sigma^{\text{Mon}} = \{\cdot^{(2)},\ \epsilon^{(0)}\}$$
$$E^{\text{Mon}} = \{(x \cdot y) \cdot z = x \cdot (y \cdot z), \quad \epsilon \cdot x = x, \quad x \cdot \epsilon = x\}$$

This reads that the signature of Mon consists of a binary symbol $\cdot$ and a nullary symbol ϵ such that $\cdot$ is associative, and ϵ is both a left and a right unit of $\cdot$.

The idea of algebraic effects is that the operations are ways of building new computation structures. In the monoids example, if x and y are computation

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

structures of some kind, then $x \cdot y$ is a non-deterministic combination of x and y, and ϵ is failure.

We fix a category $\mathbf{C}$ with products denoted as $\times$ and a terminal object 1. For instance, the category $\mathbf{C}$ can be **Set**, that is, the category of sets and functions, with $\times$ given by Cartesian product and 1 given by a one-element set. We define the *iterated product* of an object A as $A^0 = 1$ and $A^{n+1} = A \times A^n$. Given a family of morphisms $f_i : X \to A$ for $1 \leqslant i \leqslant n$ for some n, by $\langle f_1, \ldots, f_n \rangle : X \to A^n$ we denote the product mediator. In a Haskell implementation, we can use tuples for iterated products, and the unit type `()` for the terminal object. The product mediator can be implemented as `\x -> (f1 x, ..., fn x)`.

A *model* $\mathfrak{A}$ of a theory $\mathbb{T}$ in $\mathbf{C}$ consists of a *carrier*, that is, an object A of $\mathbf{C}$, and an *interpretation*, that is, a morphism $[\![f]\!]^{\mathfrak{A}} : A^n \to A$ for every operation $f^{(n)} \in \Sigma^{\mathbb{T}}$. Notice that given a *valuation* of variables, that is, a family of morphisms $\sigma_X : 1 \to A$ for $x \in \mathscr{X}$, we can extend the interpretation to work on $\Sigma^{\mathbb{T}}$-terms:

$$[\![x]\!]^{\mathfrak{A}}_{\sigma} = \sigma_X$$
$$[\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_{\sigma} = [\![f]\!]^{\mathfrak{A}} \circ \langle [\![t_1]\!]^{\mathfrak{A}}_{\sigma}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\sigma} \rangle$$

Additionally, we need $[\![\text{-}]\!]$ to respect the equations, that is, for all valuations of variables σ and all equations $(t = s) \in E^{\mathbb{T}}$, we require that $[\![t]\!]^{\mathfrak{A}}_{\sigma} = [\![s]\!]^{\mathfrak{A}}_{\sigma}$. The theory Mon has many familiar models. For example, if $\mathbf{C} = \mathbf{Set}$, we define the monoid $\mathscr{N}$ of natural numbers with addition. We assign A to be the set of natural numbers together with $[\![\cdot]\!]^{\mathscr{N}}(n, m) = n + m$ and $[\![\epsilon]\!]^{\mathscr{N}}() = 0$.

Given a theory $\mathbb{T}$ and its two models $\mathfrak{A}$ with a carrier A and $\mathfrak{B}$ with a carrier B, a *homomorphism* between $\mathfrak{A}$ and $\mathfrak{B}$ is a morphism $h : A \to B$ such that $h \circ [\![f]\!]^{\mathfrak{A}} = [\![f]\!]^{\mathfrak{B}} \circ h^n$ for all $f^{(n)} \in \Sigma^{\mathbb{T}}$.

Given a theory $\mathbb{T}$, its *free* model in the category $\mathbf{C}$ generated by an object A consists of

- a model of $\mathbb{T}$, denoted $\mathscr{F}^{\mathbb{T}}A$, with a carrier $F^{\mathbb{T}}A$,
- a morphism $\eta^{\mathbb{T}}_A : A \to F^{\mathbb{T}}A$

such that, given any other model $\mathfrak{B}$ with a carrier B and a morphism $g : A \to B$, there exists a unique homomorphism from $\mathscr{F}^{\mathbb{T}}A$ to $\mathfrak{B}$ (given by a morphism $\widehat{g} : F^{\mathbb{T}}A \to B$ in $\mathbf{C}$) such that $g = \widehat{g} \circ \eta^{\mathbb{T}}_A$. Free models are important, because they give rise to monads:

Proposition 1 (*e.g., Mac Lane, 1998, Sections IV.1 and VI.1*)
If a theory $\mathbb{T}$ has a free model $\mathscr{F}^{\mathbb{T}}A$ for every object A in the category $\mathbf{C}$, then the mapping $A \mapsto F^{\mathbb{T}}A$ extends to a monad. In detail, it is an endofunctor with the action on morphisms given as:

$$F^{\mathbb{T}}(g : A \to B) = \widehat{\eta^{\mathbb{T}}_B \circ g}$$

The unit (return) of the monad is given by the family of morphisms $\eta^{\mathbb{T}}$, while the multiplication (join) is given as $\mu^{\mathbb{T}}_A = \widehat{id}$ for the identity morphism $id : F^{\mathbb{T}}A \to F^{\mathbb{T}}A$.

Every theory $\mathbb{T}$ has a free model in the category of sets. If $E^{\mathbb{T}}$ is empty, $F^{\mathbb{T}}A$ is the set of all $\Sigma^{\mathbb{T}}$-terms with variables from the set A. (See, e.g., Baader & Nipkow,

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

1998, Corollary 3.5.8, or Mac Lane, 1998, Section VI.8.) In other words, the obtained monad is the free monad generated by the signature understood as an endofunctor. If there are some equations, $F^{\mathbb{T}}A$ is the quotient set of $\Sigma^{\mathbb{T}}$-terms induced by the equivalence relation $\approx_{\mathbb{T}}$ defined as the smallest congruence generated by $E^{\mathbb{T}}$.

(One can also consider other basic categories; for example, **C** as a category of cpo's allows us consider recursion, e.g., Plotkin & Power, 2004.)

In general, the situation is less straightforward, since **C** does not necessarily have quotients. In Haskell, free models of theories with no equations can be modelled by inductive types, but there is no way to directly encode quotients in Haskell. Thus, given a theory with some equations, we have to put some effort into implementing a free model, if one exists.

3 Term rewriting and normal forms

Another possibility to find free models is to represent each equivalence class of terms by a selected representative, a *normal form*, such that the set of normal forms have a representation in **C**. One technique to obtain normal forms is to interpret the equations ($E^{\mathbb{T}}$) as a confluent and normalising term-rewriting system.

A rewriting system is *confluent* if for all terms t, t', and t'' such that $t \rightsquigarrow t'$ and $t \rightsquigarrow t''$, there exists a term s such that $t' \rightsquigarrow \cdots \rightsquigarrow s$ and $t'' \rightsquigarrow \cdots \rightsquigarrow s$. We call a rewriting system *normalising* if for every term t, there exists a term s such that $t \rightsquigarrow \cdots \rightsquigarrow s$ and there is no term s' such that $s \rightsquigarrow s'$. We call such s a *normal form* of t. In a system that is both confluent and normalising, every term has a unique normal form. In such a case, we write $\mathsf{nf}(t)$ for the normal form of a term t.

Proposition 2 (*See, e.g., Baader & Nipkow, 1998, Theorem 2.1.9*)
Let $\mathbb{T}$ be a theory and R be a set of rewrite rules obtained from $\mathbb{T}$ by associating a rewrite rule (either $t \rightsquigarrow s$ or $s \rightsquigarrow t$) with each equation $(t = s) \in E^{\mathbb{T}}$. If R is confluent and normalising, then for all $\Sigma^{\mathbb{T}}$-terms t and s, it is the case that $t \approx_{\mathbb{T}} s$ if and only if $\mathsf{nf}(t) = \mathsf{nf}(s)$.

If we fix such a confluent and normalising system, the carrier of the induced free model $\mathscr{F}^{\mathbb{T}}A$ in **Set** is given by the set of normal forms of terms with variables from A together with the interpretations $[\![f]\!]^{\mathscr{F}^{\mathbb{T}}A}(t_1, \ldots, t_n) = \mathsf{nf}(f(t_1, \ldots, t_n))$ for $f^{(n)} \in \Sigma^{\mathbb{T}}$. The associated morphism is given as $\eta^{\mathbb{T}}_A(x) = \mathsf{nf}(x)$, while $\widehat{g}(t) = [\![t]\!]^{\mathfrak{B}}_g$ for any model $\mathfrak{B}$ and $g : A \to B$ understood as an A-indexed family of maps $1 \to B$. In particular, this means that the monadic multiplication is given simply as $\mu_A(t) = \mathsf{nf}(\mu^{\Sigma}_A(t))$, where μ^{Σ} is the multiplication of the monad of $\Sigma^{\mathbb{T}}$-terms. In general categories, we can recreate this construction, by using the free monad $F^{\Sigma}A$ generated by $\Sigma^{\mathbb{T}}$ for the set of terms, a subobject of $F^{\Sigma}A$ for the set of normal forms, and an appropriate retraction of the inclusion of the subobject for $\mathsf{nf}(\text{-})$.

Going back to the example of monoids, we can interpret the equations in E^{MON} as rewrite rules going from the left-hand sides to the right-hand sides of the equations, that is:

$$(x \cdot y) \cdot z \rightsquigarrow x \cdot (y \cdot z), \quad \epsilon \cdot x \rightsquigarrow x, \quad x \cdot \epsilon \rightsquigarrow x.$$

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

It is not too difficult to see that this system is confluent and normalising and that normal forms are either of the shape ϵ or $x_0 \cdot (x_1 \cdot (\ldots \cdot (x_{n-1} \cdot x_n) \ldots))$ for $n \geqslant 1$. Terms of these shapes are in 1–1 correspondence with terms of the shape $x_0 \cdot (x_1 \cdot (\ldots \cdot (x_n \cdot \epsilon) \ldots))$ for $n \geqslant 0$. In Haskell, the set of such terms can be expressed as the familiar cons-list datatype. Thus, in Haskell, the type of lists `[a]` is an implementation of the free monoid on a type `a`, and the list monad is a reasonable monad for non-determinism.

4 Cut as a left zero

In Section 3, we go through a bit of a hassle to reinvent the usual list monad. The gain is that we can apply the same machinery to different theories to obtain a list monad with *cut*.

To represent the *cut* operation, we extend the theory Mon with a nullary symbol **!**. Its intuitive meaning is to discard all future choices, that is, those on the right-hand side of $\cdot$. We can formalise this as the following algebraic theory:

$$\Sigma^{\mathrm{LZ}} = \Sigma^{\mathrm{Mon}} \cup \{\mathbf{!}^{(0)}\}$$
$$E^{\mathrm{LZ}} = E^{\mathrm{Mon}} \cup \{\mathbf{!} \cdot x = \mathbf{!}\}$$

Models of this theory are often called *left-zero monoids*. Note that **!** is nullary, hence, on the programming languages side, it does not have a result that would play the role of the rest of the computation. This means that **!** in fact represents the *cut-fail* operation, while *cut* described in Section 1, which has the value of the unit type as its result, can be given as $() \cdot \mathbf{!}$.

Interpreting the equations in E^{LZ} as rewrite rules going from the left-hand sides to the right-hand sides, we obtain a confluent and normalising term-rewriting system with the normal forms of the shapes ϵ, $x_0 \cdot (x_1 \cdot (\ldots \cdot (x_{n-1} \cdot \mathbf{!}) \ldots))$, and $x_0 \cdot (x_1 \cdot (\ldots \cdot (x_{n-1} \cdot x_n) \ldots))$ for $n \geqslant 1$. They are in 1–1 correspondence with terms of the shape

$$x_0 \cdot (x_1 \cdot (\ldots \cdot (x_n \cdot N) \ldots))$$

for $n \geqslant 0$, where $N \in \{\epsilon, \mathbf{!}\}$. We can represent terms of this shape in Haskell as a datatype, which give us the implementation shown in Figure 1. As discussed in Section 3, the monadic multiplication arises as an implementation of the rewrite rules. The function `concatC` does not implement the rules directly, but one can easily see that it computes the normal form of its argument.

The Haskell implementation comes with an equational theory, derived from the algebraic theory of left-zero monoids. This mirrors the algebra-of-programming style reasoning used by Hinze (2000) in his analysis of backtracking. But we argue that this approach, starting with algebraic theories, is more principled, because we distinguish between the fundamental equations of the notion of computation (E^{LZ}) and the other standard equations such as the monad laws and algebraicity, which we discuss in the next section.

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

```
data CutList a = a :. CutList a | Nil | Zero

cut :: CutList ()
cut = () :. Zero

fromList :: [a] -> CutList a
fromList (x : xs) = x :. fromList xs
fromList []       = Nil

toList :: CutList a -> [a]
toList (x :. xs) = x : toList xs
toList _         = []

instance Functor CutList where
  fmap f (x :. xs) = f x :. fmap f xs
  fmap f Nil       = Nil
  fmap f Zero      = Zero

concatC :: CutList (CutList a) -> CutList a
concatC ((a :. xs) :. xss) = a :. concatC (xs :. xss)
concatC (Nil       :. xss) = concatC xss
concatC (Zero      :. _)   = Zero
concatC Nil                = Nil
concatC Zero               = Zero

instance Monad CutList where
  return a = a :. Nil
  m >>= f  = concatC (fmap f m)
```

Fig. 1. Haskell implementation via left-zero monoids.

5 Scope delimiter

As defined in Section 4, the *cut* operator is global: It discards *all* subsequent choices. Sometimes, however, we want to delimit its scope. (In Prolog, its scope is always limited to the predicate in which it is used.) For instance, consider the following function. It allows us to extract the longest prefix of a `CutList` for which a predicate `p` holds:

```
fail :: CutList ()
fail = fromList []

takeWhileC :: (a -> Bool) -> CutList a -> CutList a
takeWhileC p xs = do
  x <- xs
  when (not (p x)) (cut >> fail)
  return x
```

In isolation, this function works as intended. For example:

```
toList (takeWhileC even (fromList [2,4,5,8])) == [2,4]
```

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

But when we want to use `takeWhileC` in a bigger program, an unexpected behaviour occurs. The program below is supposed to return elements from the longest 'even' prefixes of the given lists:

```
prefixes :: CutList Int
prefixes = do
  x <- fromList [fromList [2,4,5,2], fromList [8,9,10]]
  y <- takeWhileC even x
  return y
```

However:

```
toList prefixes == [2,4]
```

Why not the intended `[2,4,8]`? The reason is that the cut in `takeWhileC` works globally: It discards also the choice `x = [8,9,10]`. A solution would be to delimit the scope of the cut with a function `scope :: CutList a -> CutList a`, which does not allow the cuts to leak outside of its argument. With such a function, the correct definition of `takeWhileC` would be:

```
takeWhileC :: (a -> Bool) -> CutList a -> CutList a
takeWhileC p xs = scope ( do
  x <- xs
  when (not (p x)) (cut >> fail)
  return x )
```

With this definition, we should get:

```
toList prefixes == [2,4,8]
```

Unfortunately, `scope` does not arise as an operation in an algebraic theory. This follows from the fact that every such operation is *algebraic*. In Haskell terms, an operation `op` is algebraic if it commutes with the bind operator:

```
op (t1, ..., tn) >>= f == op (t1 >>= f, ..., tn >>= f)
```

However, taking into account the intended semantics of `scope`, we see that it is not algebraic:

```
scope (fromList [1,2]) >> cut == cut
```

while

```
scope (fromList [1,2] >> cut) == fromList [()]
```

Such non-algebraic operations (another example is the *catch* operation related to exceptions) are often explained in terms of *handlers* introduced by Plotkin and Pretnar (2013). A handler of a theory $\mathbb{T}$ is a model $\mathfrak{B}$ with a carrier B together with a morphism $g : A \to B$. The unique homomorphism $\widehat{g} : F^{\mathbb{T}}A \to B$ induced by the freeness of the model represents the action of handling a computation with variables of the type A. For example, consider the theory Mon and the monoid $\mathcal{N}$ of natural

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

numbers with addition. Given a value of $F^{\text{Mon}}A$ and a morphism $g : A \to \mathcal{N}$, we can find the sum of values assigned to elements on the list by the morphism g.

A naive attempt to define a *scope* handler as a simple erasure of cuts, that is, the carrier $F^{\text{LZ}}A$ with the interpretation

$$[\![\cdot]\!]^E(a,b) = a \cdot b$$
$$[\![\epsilon]\!]^E() = \epsilon$$
$$[\![!]\!]^E() = \epsilon$$

and η^{LZ} for the morphism g, is not a model of LZ. In particular:

$$[\![! \cdot x]\!]^E = [\![!]\!]^E \cdot [\![x]\!]^E = \epsilon \cdot x = x \neq \epsilon = [\![!]\!]^E$$

In order to solve this problem, in the subsequent sections, we introduce a theory that is equivalent to LZ, but allows us to study the relationship between the *cut* operator and the theory Mon in finer details.

6 Cut as a unary idempotent operation

An alternative way to give a theory for backtracking with *cut* is to extend the theory Mon with a unary symbol -* written postfix. Intuitively, we think of an expression e^* as 'discard the yet uninspected choices and continue with e'. As used in the example from Section 1, `cut` corresponds to `()`*, so the Haskell expression `cut >> e` corresponds to `e`*. We introduce some equations, arriving at the following theory, which we call Cut:

$$\Sigma^{\text{Cut}} = \Sigma^{\text{Mon}} \cup \{\text{-}^{*(1)}\}$$
$$E^{\text{Cut}} = E^{\text{Mon}} \cup \{(x^*) \cdot y = x^*, \quad x \cdot (y^*) = (x \cdot y)^*, \quad (x^*)^* = x^*\}$$

The first equation simply states that -* discards the subsequent choices (that is, those on the right-hand side of $\cdot$). The second equation states that -* does not affect the previous choices. The third one states that -* is idempotent. (In future, we assume that $-^*$ binds more tightly than $\cdot$, so the second equation can be written $x \cdot y^* = (x \cdot y)^*$.)

We prove the equivalence of Cut and LZ. In one direction, we define $! = \epsilon^*$. We verify the equality:

$$! \cdot x = \epsilon^* \cdot x \approx_{\text{Cut}} \epsilon^* = !$$

In the other direction, we define $x^* = x \cdot !$. We verify the equalities:

$$(x^*)^* = (x \cdot !) \cdot ! \approx_{\text{LZ}} x \cdot (! \cdot !) \approx_{\text{LZ}} x \cdot ! = x^*$$
$$x^* \cdot y = (x \cdot !) \cdot y \approx_{\text{LZ}} x \cdot (! \cdot y) \approx_{\text{LZ}} x \cdot ! = x^*$$
$$x \cdot y^* = x \cdot (y \cdot !) \approx_{\text{LZ}} (x \cdot y) \cdot ! = (x \cdot y)^*$$

The two transformations are mutual inverses. In one direction:

$$! \mapsto \epsilon^* \mapsto \epsilon \cdot ! \approx_{\text{LZ}} !$$

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

In the other direction:

$$x^* \mapsto x \cdot ! \mapsto x \cdot \epsilon^* \approx_{\text{CUT}} (x \cdot \epsilon)^* \approx_{\text{CUT}} x^*$$

We can see the theory CUT as a sum of the theory MON with the theory of a unary idempotent operation IDEM:

$$\Sigma^{\text{IDEM}} = \{\text{-}^{*(1)}\}$$
$$E^{\text{IDEM}} = \{(x^*)^* = x^*\}$$

together with the following set of compatibility equations:

$$Q = \{x^* \cdot y = x^*, \quad x \cdot y^* = (x \cdot y)^*\}$$

That is, $\Sigma^{\text{CUT}} = \Sigma^{\text{MON}} \cup \Sigma^{\text{IDEM}}$ and $E^{\text{CUT}} = E^{\text{MON}} \cup E^{\text{IDEM}} \cup Q$.

The theory IDEM has normal forms that are easy to describe. Each term is either a variable, say x, or is provably equal to x^*. One could also recognise the free monad of IDEM as the writer monad for the monoid of truth values with disjunction.

A challenge, then, is to find normal forms for the combined theory CUT.

7 Distributive law

To find normal forms of the theory CUT, we notice that any expression can be rewritten to an expression e or e^*, where e is an expression in the theory of monoids. This is done by orienting the equations in Q:

$$x^* \cdot y \rightsquigarrow x^* \qquad x \cdot y^* \rightsquigarrow (x \cdot y)^* \tag{1}$$

to pull -^* to the outside of any expression.

However, the situation is slightly subtle. The expression $x^* \cdot y^*$ can be rewritten using (1) in two different ways:

$$x^* \cdot (y^*) \rightsquigarrow x^* \qquad (x^*) \cdot y^* \rightsquigarrow ((x^*) \cdot y)^* \rightsquigarrow (x^*)^*$$

The rewriting is only confluent modulo the theories IDEM and MON.

This kind of distributivity is a common phenomenon in algebraic theories. To illustrate the method, we describe the situation in some generality, since our treatment appears to be novel. As a shorthand, we write $n \vdash_{\mathbb{S}} s$ to mean s is a term in $\mathbb{S}$ with free variables in $\{x_1, \ldots, x_n\}$.

Definition 3
Let $\mathbb{U}$ be an algebraic theory that contains two theories, say $\mathbb{S}$ and $\mathbb{T}$.

1. A term in $\mathbb{U}$ is *separated* if it is a term built from $\mathbb{S}$ over terms built from $\mathbb{T}$. More precisely: The separated terms are of the form $s[t_i/x_i]$, where $m \vdash_{\mathbb{S}} s$ and $n \vdash_{\mathbb{T}} t_1, \ldots, n \vdash_{\mathbb{T}} t_m$.
2. The theory $\mathbb{U}$ is a *composite* of $\mathbb{S}$ and $\mathbb{T}$ if every term in $\mathbb{U}$ is equal (in $\mathbb{U}$) to a separated term, and moreover this separation is unique-modulo-$(\mathbb{S}, \mathbb{T})$ in the following sense: If there are terms $m \vdash_{\mathbb{S}} s$, $m' \vdash_{\mathbb{S}} s'$, $n \vdash_{\mathbb{T}} t_1, \ldots, n \vdash_{\mathbb{T}} t_m$, $n \vdash_{\mathbb{T}} t'_1, \ldots, n \vdash_{\mathbb{T}} t'_{m'}$ such that $s[t_i/x_i] \approx_{\mathbb{U}} s'[t'_i/x'_i]$, then there are functions $m \xrightarrow{f} p \xleftarrow{f'} m'$ and terms $n \vdash_{\mathbb{T}} \bar{t}_1, \ldots, n \vdash_{\mathbb{T}} \bar{t}_p$ such that:

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

- $s[f(i)/x_i] \approx_{\mathbb{S}} s'[f'(i)/x_i]$,
- $t_i \approx_{\mathbb{T}} \bar{t}_{f(i)}$ $(1 \leqslant i \leqslant m)$,
- $t'_i \approx_{\mathbb{T}} \bar{t}_{f'(i)}$ $(1 \leqslant i \leqslant m')$.

For instance, Cut is a composite of Idem and Mon since the rules (1) rewrite any expression to a separated one in a way that is unique-modulo-(Idem, Mon).

The classic example of a composite theory is the theory of rings, where the distributive law $x \cdot (y + z) = x \cdot y + x \cdot z$ separates terms into sums of products. We consider this precisely for a moment, to illustrate Definition 3. Consider the theory of Abelian groups:

$$\Sigma^{\text{Ab}} = \{0^{(0)}, +^{(2)}, -^{(1)}\}$$
$$E^{\text{Ab}} = \{x + 0 = x, \quad x + (y + z) = (x + y) + z, \quad x + y = y + x, \quad x + (-x) = 0\}$$

The theory of rings is:

$$\Sigma^{\text{Ring}} = \Sigma^{\text{Ab}} \cup \Sigma^{\text{Mon}}$$
$$E^{\text{Ring}} = E^{\text{Ab}} \cup E^{\text{Mon}} \cup \{x \cdot (y + z) = x \cdot y + x \cdot z, \quad (x + y) \cdot z = x \cdot z + y \cdot z\}$$

It is a composite of the theories Ab and Mon, since every ring expression can be rewritten as a sum of products, uniquely mod-(Ab, Mon). The uniqueness is not trivial: Notice that the expression $x + -(x \cdot \epsilon)$ is separated, but so is the expression 0, and they are equal in Ring. Thus, the separated normal form is not unique, but it is unique-mod-renaming, according to our definition, since $x \cdot \epsilon \approx_{\text{Mon}} x$ and $a - a \approx_{\text{Ab}} 0$.

Composite theories are a tool for building monads. Recall that if S and T are monads, some extra data are needed to turn the composite functor ST into a monad. One way to express such data is via a *distributive law*:

Definition 4 (*Beck, 1969*)
Let (S, η^S, μ^S), (T, η^T, μ^T) be monads on a category. A *distributive law* is a morphism $\lambda : TS \to ST$ such that the following hold:

- $\mu^S T \cdot S\lambda \cdot \lambda S = \lambda \cdot T\mu^S : TSS \to ST$
- $S\mu^T \cdot \lambda T \cdot T\lambda = \lambda \cdot \mu^T S : TTS \to ST$
- $\lambda \cdot \eta^T S = S\eta^T : S \to ST$
- $\lambda \cdot T\eta^S = \eta^S T : T \to ST$

Such a distributive law λ yields a monad $(ST,\ S\eta^T \cdot \eta^S,\ \mu^S T \cdot SS\mu^T \cdot S\lambda T)$.

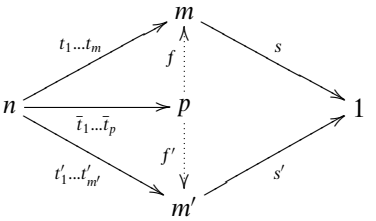
Theorem 5
Let $\mathbb{U}$ be a composite of theories $\mathbb{S}$ and $\mathbb{T}$. If, in some category with products, $\mathbb{S}$ and $\mathbb{T}$ have free models, then so does $\mathbb{U}$, and the free-model-monad of $\mathbb{U}$ arises from a distributive law $\lambda : F^{\mathbb{T}} F^{\mathbb{S}} \to F^{\mathbb{S}} F^{\mathbb{T}}$.

Conversely, consider theories $\mathbb{S}$ and $\mathbb{T}$. Every distributive law $\lambda : F^{\mathbb{T}} F^{\mathbb{S}} \to F^{\mathbb{S}} F^{\mathbb{T}}$ on **Set** induces a monad that is the free-model-monad of a composite of $\mathbb{S}$ and $\mathbb{T}$.

Thus, in particular, the free models of the theory Cut are of the form $F^{\text{Idem}}(F^{\text{Mon}}(A))$.

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

Note. We covered this section in some generality because composite theories are a key part of our method, and Theorem 5 is very important from the perspective of building monads for functional programming, and yet we cannot find it in the literature. Distributive laws have long been studied; a popular reference is Barr & Wells (1985). The idea of composite theories is often mentioned but usually left informal. The first formal treatment that we can find is Cheng (in press), which analyses the situation is terms of factorization systems in Lawvere theories. Our Definition 3 is inspired by Cheng's Definition 4.10; we built on Cheng's work by being more concrete (using algebraic theories instead of Lawvere theories) and by simplifying the notion of uniqueness condition (the zig-zag condition in Cheng's result appears to be redundant in this setting). The readers familiar with Cheng's work will find it helpful to visualize the conditions in our definition as the following diagram in the corresponding Lawvere theory.



In the case of term-rewriting rules, if the normal forms are separated, the distributive law $\lambda : F^{\mathbb{T}}F^{\mathbb{S}} \to F^{\mathbb{S}}F^{\mathbb{T}}$ can be explicitly defined simply as $\lambda_A(t) = \mathsf{nf}(t)$, where $t : F^{\mathbb{T}}F^{\mathbb{S}}A$ is understood as a $\Sigma^{\mathbb{U}}$-term. For instance, a Haskell implementation of the free monad of the theory CUT via such a distributive law is shown in Figure 2.

8 Delimiting scope: models and handlers

Recall from Section 5 the problem of defining a scope delimiter for *cut* in the theory LZ. While the intuitive understanding of what we aim at is simple—we want to erase cuts—the obvious implementation is not a handler, as what we obtain is not a model of the theory.

With the development from Section 7, we can use the fact that CUT is a compatible composition of IDEM and MON to solve the problem. The idea is to interpret only the 'outer' layer, that is, the monad F^{IDEM}. We use the model of IDEM with the carrier $F^{\text{IDEM}}F^{\text{MON}}A$, the interpretation

$$[\![\text{-}^*]\!](a) = a,$$

and $\eta^{\text{IDEM}} : F^{\text{MON}}A \to F^{\text{IDEM}}F^{\text{MON}}A$ for g. It is a proper handler, and so the induced morphism is a homomorphism of IDEM (but it is not a homomorphism of CUT). In Haskell, such erasure of cuts can be implemented without difficulty:

```
scope = fromList . toList
```

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

```
data Idem a = Ret a | Flag a

fromIdem :: Idem a -> a
fromIdem (Ret a)  = a
fromIdem (Flag a) = a

instance Functor Idem where
  fmap f (Ret a)  = Ret (f a)
  fmap f (Flag a) = Flag (f a)

instance Monad Idem where
  return a = Ret a
  Ret a  >>= f = f a
  Flag a >>= f = Flag (fromIdem (f a))

distr :: [Idem a] -> Idem [a]
distr (Ret a  : xs) = fmap (\y -> a : y) (distr xs)
distr (Flag a : xs) = Flag [a]
distr []            = Ret []

type CutList a = Idem [a]

cut :: CutList ()
cut = Flag (return ())

fromList :: [a] -> CutList a
fromList xs = Ret xs

toList :: CutList a -> [a]
toList x = fromIdem x

instance Monad CutList where
  return a = fmap return (return a)
  m >>= f  = fmap join (join (fmap distr (fmap (fmap f) m)))
```

Fig. 2. Haskell implementation via a distributive law.

9 Remarks

Equations defining cut. The equations specifying the semantics of *cut* are similar to the ones studied by Hinze (2000), who presents them as equations between Haskell expressions involving the monadic operators return and bind:

$$(! \gg m) \cdot n = ! \gg m$$
$$! \gg (m \cdot n) = m \cdot (! \gg n)$$
$$! \gg return\ () = !$$

With equational theories as used in this pearl—as well as the more general algebraic theories à la Plotkin and Power (2004)—we do not think about the monadic structure, as it is determined by the algebraic specification.

The normal forms of $\mathscr{F}^{\mathrm{CUT}}$ and $\mathscr{F}^{\mathrm{LZ}}$ are somewhat dual: If *cut* is present, it is always the outermost operation in the former, and the innermost operation in the latter. Hinze (2000) also essentially obtain the left-zero implementation of a list with two terminators. A similar construction was considered also by Billaud (1990).

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

To the authors' best knowledge, the dual construction that uses a distributive law is new.

Distributive laws. It is known that the list monad distributes over any commutative monad in a canonical way. The unary-idempotent monad is commutative, but the canonical distributive law is a different one than `distr` shown in Figure 2. The canonical one checks whether at least one element is 'flagged', but it does not discard any elements. It arises from the theory $\Sigma^{\text{MON}} \cup \Sigma^{\text{IDEM}}$ together with $E^{\text{MON}} \cup E^{\text{IDEM}} \cup C$, where the coherence set C is given as follows:

$$C = \{x \cdot (y^*) = (x \cdot y)^*, \quad (x^*) \cdot y = (x \cdot y)^*\} \tag{2}$$

For example:

$$x \cdot y^* \cdot z \cdot t^* \cdot u \rightsquigarrow (x \cdot y \cdot z \cdot t \cdot u)^*$$

There is also a third non-trivial distributive law, which arises from the sum of theories MON and IDEM together with the following set of coherence equations D:

$$D = \{x \cdot y^* = y^*, \quad x^* \cdot y = (x \cdot y)^*\}$$

It is equivalent to the theory of right-zero monoids. One could interpret this as backtracking with a 'start over' operator, which discards previous results.

These three distinct possibilities define three different ways to make the endofunctor $F^{\text{IDEM}}(F^{\text{MON}}(-))$ into a monad while still respecting the monads F^{IDEM} and F^{MON} (according to Section 7). As an aside we recall that Hyland *et al.* (2006) have observed that many theories can be built by combining other theories using tensor and sum constructions; a 'distributive tensor' has also been proposed, which corresponds to the 'canonical' distributive law of (1). But we contend that constructions like these will never be enough to capture all the useful ways of combining theories. The main example of this paper (Section 6) shows that it is ultimately profitable to consider arbitrary distributive laws.

Monad transformers. As discussed by Jaskelioff & Moggi (2010), and by Piróg (2016), to obtain a proper combination of backtracking with other effects, one can compose a theory $\mathbb{T}$ with MON by adding the following set of equations:

$$Q = \{f(x_1, \ldots, x_n) \cdot y = f(x_1 \cdot y, \ldots, x_n \cdot y)\}_{f^{(n)} \in \Sigma^{\mathbb{T}}}$$

The free models of such combinations give rise to the list monad transformer (done right) known from Haskell libraries. Interestingly, if we want to combine $\mathbb{T}$ with CUT or LZ, it seems that we do not need any additional coherence equations except for Q. While the left-zero approach easily scales to the transformer case, yielding a list monad transformer with two possible terminators, the distributive-law approach fails, as there is no general way to pull -* outside of $\mathbb{T}$-operations.

Implementation via continuations. Hinze (2012) shows how to derive a continuation-based implementation of the list monad using the codensity monad construction complemented with Cayley representation of monoids. It is an interesting challenge

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press

to find similar representations of the constructions presented in this article to obtain more efficient implementations.

10 Conclusion

The goal of this pearl is to investigate how algebraic understanding of effects relates to monads as used in functional programming. Backtracking with cut is an appealing example: It can be captured by simple algebraic theories, but it is still peculiar enough not to be reduced to an instance of the most standard constructions, such as sums and different kinds of tensors (Hyland & Power, 2006; Hyland *et al.*, 2006).

We argue that the choice of a presentation of a theory can lead to different implementations of the notion of computation and that it can reveal interesting properties of the induced monad—in our example it is the fact that backtracking with cut can be obtained as a principled composition of the list monad with the 'unary idempotent operation' monad.

Acknowledgments

We would like to thank Ralf Hinze, who encouraged us to write this pearl in the first place, Kwok-Ho Cheun for enlightening discussions, and the anonymous reviewers, whose comments and suggestions helped us improve the presentation.

References

Baader, F. & Nipkow, T. (1998) *Term Rewriting and All That*. Cambridge University Press.

Barr, M. & Wells, C. (1985) *Toposes, Triples and Theories*. Springer-Verlag.

Beck, J. M. (1969) Distributive laws. In *Seminar on Triples and Categorical Homology Theory*, Lecture Notes in Mathematics, vol. 80. Berlin/Heidelberg: Springer, pp. 119–140.

Billaud, M. (1990) Simple operational and denotational semantics for Prolog with cut. *Theor. Comput. Sci.* **71**(2), 193–208.

Bird, R. S. (2006) Functional pearl: A program to solve Sudoku. *J. Funct. Progr.* **16**(6), 671–679.

Cheng, E. (2011) Distributive laws for Lawvere theories. *Algebra Universalis*. `arXiv:1112.3076`.

Hinze, R. (2000) Deriving backtracking monad transformers. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00), pp. 186–197.

Hinze, R. (2012) Kan extensions for program optimisation or: Art and Dan explain an old trick. In Proceedings of Mathematics of Program Construction—11th International Conference, MPC 2012, Lecture Notes in Computer Science, vol. 7342. Berlin/Heidelberg: Springer, pp. 324–362.

Hyland, M. & Power, J. (2006) Discrete Lawvere theories and computational effects. *Theor. Comput. Sci.* **366**(1), 144–162.

Hyland, M., Plotkin, G. D. & Power, J. (2006) Combining effects: Sum and tensor. *Theor. Comput. Sci.* **357**(1–3), 70–99.

Jaskelioff, M. & Moggi, E. (2010) Monad transformers as monoid transformers. *Theor. Comput. Sci.* **411**(51–52), 4441–4466.

Mac Lane, S. (1998) *Categories for the Working Mathematician*, 2nd ed. Springer.

Piróg, M. (2016) Eilenberg–Moore monoids and backtracking monad transformers. In Proceedings 6th Workshop on Mathematically Structured Functional Programming, Electronic Proceedings in Theoretical Computer Science, vol. 207, pp. 23–56.

Plotkin, G. D. & Power, A. J. (2004) Computational effects and operations: An overview. *Electron. Notes Theor. Comput. Sci.* **73**, 149–163.

Plotkin, G. D. & Pretnar, M. (2013) Handling algebraic effects. *Log. Methods Comput. Sci.* **9**(4).

https://doi.org/10.1017/S0956796817000077 Published online by Cambridge University Press