Under consideration for publication in J. Functional Programming

FUNCTIONAL PEARLS

Reasoning and Derivation of Monadic Programs

A Case Study of Non-determinism and State

Shin-Cheng Mu
Academia Sinica, Taiwan
(*e-mail:* `scm@iis.sinica.edu.tw`)

Abstract

Equational reasoning is among the most important tools that functional programming provides us. Curiously, relatively little attention has been paid to reasoning about monadic programs. In this pearl we aim to develop theorems and patterns useful for the derivation of monadic programs, focusing on non-determinism and state. We model the aggregation function in Spark as a monadic program and derive conditions under which it is deterministic. We also investigate the intricate interaction between state and non-determinism, and derive two backtracking algorithms, respectively using local and global states, to solve the *n*-queens and Sudoku puzzles.

1 Introduction

Equational reasoning is among the many gifts that functional programming offers us. Functional programs preserve a rich set of mathematical properties, which not only helps to prove property about programs in a relatively simple and elegant manner, but also aids the development of programs. One may refine a clear but inefficient specification, stepwise through equational reasoning, to an efficient program whose correctness may not be obvious without such a derivation.

It is misleading if one says that functional programming does not allow side effects. In fact, even a purely functional language may allow a variety of side effects. It is just that they insist side effects should be introduced in a rigorous, mathematically manageable manner. Since the introduction of *monads* into the functional programming community (Moggi, 1989; Wadler, 1992), it has become the main framework in which effects are modelled. Various monads were developed for different effects, from general ones such as IO, state, non-determinism, exception, continuation, environment passing, to specific purposes such as parsing. Numerous research were also devoted to producing practical monadic programs.

Curiously, as noted by Hutton and Fulger (2007), relatively little attention has been paid on reasoning about monadic programs. This is perhaps due to the impression that impure programs are bound to be difficult to reason about. In fact, the laws of monads and their operators are sufficient to prove quite a number of useful properties about monadic programs. Validity of these properties, being proved using only these laws, is independent from particular implementation of monads.

$$\begin{aligned}
&(\Leftarrow\!\!\ll) :: (b \to \mathsf{M}\,c) \to (a \to \mathsf{M}\,b) \to a \to \mathsf{M}\,c\\
&(f \Leftarrow\!\!\ll g)\,x = f =\!\!\ll g\,x\\
&(\langle\$\rangle)\ \ :: (a \to b) \to \mathsf{M}\,a \to \mathsf{M}\,b\\
&f \langle\$\rangle\, m = (\mathit{return} \cdot f) =\!\!\ll m\\
&(\langle\bullet\rangle)\ \ :: (b \to c) \to (a \to \mathsf{M}\,b) \to (a \to \mathsf{M}\,c)\\
&f \langle\bullet\rangle\, g = (\mathit{return} \cdot f) \Leftarrow\!\!\ll g
\end{aligned}$$

Fig. 1: Some monadic operators we find handy for this paper.

This pearl follows the trail of Hutton and Fulger (2007) and Gibbons and Hinze (2011), aiming to develop theorems and patterns that are useful for reasoning about monadic programs. We focus on two effects — non-determinism and state, and their interaction. Regarding non-determinism, we propose a collection of operators and laws, and discuss a case study: we model Spark aggregation, a distributive computation model, as a monadic program, and derive conditions under which the Spark aggregation is actually deterministic. The interaction between non-determinism and state is known to be intricate. We derive two backtracking algorithms that uses non-determinism and state, in which the states are respectively local and global. In both cases we develop theorems that convert a variation of *scanl* to a *foldr* that uses the state monad, as well as theorems constructing hylomorphism. The algorithms are used to solve the n-queens and the Sudoku puzzles.

2 Monad and Effect Operators

A monad consists of a type constructor $\mathsf{M} :: * \to *$ and two operators $\mathit{return} :: a \to \mathsf{M}\,a$ and "bind" $(=\!\!\ll) :: (a \to \mathsf{M}\,b) \to \mathsf{M}\,a \to \mathsf{M}\,b$ that satisfy the following *monad laws*:

$$f =\!\!\ll \mathit{return}\,x = f\,x \quad , \tag{1}$$

$$\mathit{return} =\!\!\ll m = m \quad , \tag{2}$$

$$f =\!\!\ll (g =\!\!\ll m) = (\lambda x \to f =\!\!\ll g\,x) =\!\!\ll m \quad . \tag{3}$$

Rather than the usual $(\gg\!\!=) :: \mathsf{M}\,a \to (a \to \mathsf{M}\,b) \to \mathsf{M}\,b$, in the laws above we use the reversed bind $(=\!\!\ll)$, which is consistent with the direction of function composition and more readable when we program in a style that uses composition a lot. In a program using bind with λ-abstractions, it is more natural to write $m \gg\!\!= \lambda x \to f\,x$. In this pearl we use both binds depending on the context. We also define $m_1 \ll m_2 = \mathit{const}\,m_1 =\!\!\ll m_2$, which has type $(\ll) :: m\,a \to m\,b \to m\,a$.

More operators we find useful are given in Figure 1. Right-to-left Kleisli composition, denoted by $(\Leftarrow\!\!\ll)$, composes two monadic operations $a \to \mathsf{M}\,b$ and $b \to \mathsf{M}\,c$ into an operation $a \to \mathsf{M}\,c$. Operators $(\langle\$\rangle)$ and $(\langle\bullet\rangle)$ are monadic counterparts of function application and composition: $(\langle\$\rangle)$ applies a pure function to a monad, while $(\langle\bullet\rangle)$ composes a pure function after a monadic function. The following properties can be proved from their definitions:

$$f =\!\!\ll (g \langle\$\rangle\, m) = (f \cdot g) =\!\!\ll m \quad , \tag{4}$$

$$(f \langle\bullet\rangle\, g)\,x = f \langle\$\rangle\, g\,x \quad , \tag{5}$$

$$(f \cdot g) \langle\$\rangle\, m = f \langle\$\rangle\, (g \langle\$\rangle\, m) \quad , \tag{6}$$

$$(f \cdot g) \langle\bullet\rangle\, m = f \langle\bullet\rangle\, (g \langle\bullet\rangle\, m) \quad . \tag{7}$$

It is interesting to notice, in (5) through (7), that $(\langle\$\rangle)$ and $(\langle\bullet\rangle)$ share properties similar to pure function application and composition. None of these operators are strictly necessary: they can all be reduced to *return*, $(=\!\ll)$, and λ-abstractions. As is often the case when designing notations, having more operators allows ideas to be expressed concisely in a higher level of abstraction, at the expense of having more properties to memorise. It is personal preference where the balance should be. Properties (4) through (7) may look like a lot of properties to remember. In practice, we find it usually sufficient to let us be guided by types. For example, when we have $f \langle\$\rangle\ g\ x$ and want to bring f and g together, by their types we can figure out the resulting expression should be $(f \langle\bullet\rangle\ g)\ x$.

Free monads and effect handling Monads are used to model effects, and each effect comes with its collection of operators. For example, to model non-determinism we might assume two operators $\emptyset$ and $(\talloblong)$ (*mplus*), respectively modeling failure and choice. A state effect comes with operators *get* and *put*, which respectively reads from and writes to an unnamed state variable.

A program might involve more than one effect. *Effect handling* (Kiselyov *et al.*, 2013; Kiselyov & Ishii, 2015) is a recent approach for combining multiple effects in a program. In a typical implementation, a data structure $\mathsf{Eff}\ r\ a$ is used to represent the abstract syntax tree of an effectful program yielding a result of type a, where r is a type-level list recording the effects the program may have. The syntax tree is interpreted by *handlers*, for example:

$$\begin{array}{ll} \mathit{runNondet} & :: \mathsf{Eff}\ (\mathsf{Nondet} : r)\ a \to \mathsf{Eff}\ r\ [a]\ \ , \\ \mathit{runState} & :: \mathsf{Eff}\ (\mathsf{State}\ s : r)\ a \to s \to \mathsf{Eff}\ r\ (a,s)\ \ . \end{array}$$

The type $\mathsf{Eff}\ (\mathsf{Nondet} : r)\ a$ denotes a syntax tree that may contain operators $\emptyset$ and $(\talloblong)$. The handler *runNondet* returns a program, in which $\emptyset$ and $(\talloblong)$ are interpreted and thus removed, yielding a list of a. Similarly, *runState* takes a program that may contain *get* and *put*, and returns a function that takes an initial state and returns a program that yields the result a and a final state. In the returned program, those *get* and *put* are removed, thus the type is $\mathsf{Eff}\ r\ (a,s)$. These handlers can be chained together until all effects are removed and the program contains only a pure value.

Effect handling was proposed to address some of the shortcomings of monad transformers (Liang *et al.*, 1995). We mention effect handling, however, for two morals we may learn. Firstly, it is possible to encode, in the type of a program, the effects it may incur. A program of type $\mathsf{Eff}\ [\mathsf{Nondet}]\ \mathsf{Int}$, for example, would not have *get* and *put* to be interpreted. Secondly, monadic programs can be seen as data structures that admit induction, if we restrict ourselves to finite, terminating programs. For example, to prove a property for all programs having only effect Nondet, it is sufficient to prove by induction on syntax trees consisting of only *return*, $\emptyset$, and $(\talloblong)$. [1]

A language of effectful programs To avoid the impression that the results of this pearl depend on a particular implementation of effect handling, we adopt a more abstract presentation of monads and effects. In fact, the theorems in this pearl apply to programs built by

[1] The bind operator can be seen as grafting of syntax trees (with possible optimisations).

$$\begin{array}{lll}
\text{pure programs} & \mathscr{E} & = \text{pure, non-monadic programs} \mid \mathscr{P} \\
\text{monadic progs} & \mathscr{P} & = \mathit{return}\ \mathscr{E} \mid \mathscr{E} =\!\!\ll \mathscr{P} \mid \emptyset \mid \mathscr{P} \mathbin{[\!]} \mathscr{P} \mid \mathit{get} \mid \mathit{put}\ \mathscr{E} \\
\text{types} & \mathscr{T} & = a \mid c \mid \mathscr{T} \to \mathscr{T} \mid \mathsf{M}_{\{\mathscr{F}\}}\ \mathscr{T} \\
\text{effects} & \mathscr{F} & = \mathsf{S}\ a \mid \mathsf{S}\ c \mid \mathsf{N}
\end{array}$$

a ranges over type variables,
while *c* ranges over for built-in type constants.

$$\frac{\Gamma \vdash e :: a}{\Gamma \vdash \mathit{return}\ e :: \mathsf{M}_\varepsilon\ a} \qquad \frac{\Gamma \vdash f :: a \to \mathsf{M}_\varepsilon\ b \qquad \Gamma \vdash m :: \mathsf{M}_\varepsilon\ a}{\Gamma \vdash f =\!\!\ll m :: \mathsf{M}_\varepsilon\ b}$$

$$\frac{\mathsf{N} \in \varepsilon}{\Gamma \vdash \emptyset :: \mathsf{M}_\varepsilon\ a} \qquad \frac{\Gamma \vdash m_1 :: \mathsf{M}_\varepsilon\ a \quad \Gamma \vdash m_2 :: \mathsf{M}_\varepsilon\ a \quad \mathsf{N} \in \varepsilon}{\Gamma \vdash m_1 \mathbin{[\!]} m_2 :: \mathsf{M}_\varepsilon\ a}$$

$$\frac{\mathsf{S}\ s \in \varepsilon}{\Gamma \vdash \mathit{get} :: \mathsf{M}_\varepsilon\ s} \qquad \frac{\Gamma \vdash e :: s \qquad \mathsf{S}\ s \in \varepsilon}{\Gamma \vdash \mathit{put}\ e :: \mathsf{M}_\varepsilon\ ()}$$

Fig. 2: A small language and type system for effectful programs.

monad transformers or any monad built on one's own, provided that the operators satisfy demanded properties.

We need a mechanism that clarifies what effects may occur when a program is run. We introduce a small language of effectful programs and a simple type system in Figure 2. The type $\mathsf{M}_\varepsilon\ a$ denotes a monad whose return type is a and, during whose execution, effects in ε might occur. This pearl considers only two effects: non-determinism and state. Non-determinism is denoted by N, for which we assume two operators $\emptyset :: \mathsf{M}_\varepsilon\ a$ and $(\mathbin{[\!]}) :: \mathsf{M}_\varepsilon\ a \to \mathsf{M}_\varepsilon\ a \to \mathsf{M}_\varepsilon\ a$, where $\mathsf{N} \in \varepsilon$. A state effect is denoted by $\mathsf{S}\ s$, where s is the type of the state, with two operators $\mathit{get} :: \mathsf{M}_\varepsilon\ s$ and $\mathit{put} :: s \to \mathsf{M}_\varepsilon\ ()$ where $\mathsf{S}\ s \in \varepsilon$. Note that inference of effects is not unique: a program having $\emptyset$ can be typed as both $\mathsf{M}_{\mathsf{N}}\ a$ and $\mathsf{M}_{\{\mathsf{N},\mathsf{S}\ \mathsf{Int}\}}\ a$. The effects we consider are first-order: in $\mathsf{S}\ s$, the type s cannot be monadic.

We sometimes denote the constraint on ε in a type-class-like syntax, e.g $\emptyset :: \mathsf{N} \in \varepsilon \Rightarrow \mathsf{M}_\varepsilon\ a$. Note that our type system allows syntax trees to be characterised more strictly than Haskell type classes. Given $e :: \mathsf{MonadPlus}\ m \Rightarrow m\ \mathsf{Int}$, we know that e may have $\emptyset$, $(\mathbin{[\!]})$, and possibly other effect operators. Given $e :: \mathsf{M}_{\mathsf{N}}\ \mathsf{Int}$, we know that non-determinism is the *only effect* allowed.

Like in other literature on program derivation, we assume a set-theoretic semantics in which functions are total. Lists in this pearl are inductive types, and unfolds generate finite lists too. Non-deterministic choices are finitely branching. Given a concrete input, a function always expands to a finitely-sized expression consisting of syntax allowed by its type. We may therefore prove properties of a monad of type $\mathsf{M}_\varepsilon\ a$ by structural induction over its syntax.

3 Non-deterministic Monad

As pointed out by Gibbons and Hinze (2011), for proofs and derivations, what matters is not how a monad is implemented but what properties its operators satisfy. Therefore, in the next few sections when we discuss each effect, we start with introducing its operators, discussing the properties we expect them to have, before finding actual implementations of the monad and operators that meet these properties.

We start with non-determinism. As mentioned before, we assume two operators $\emptyset$ and $(\talloblong)$: the former denotes failure, while $m \talloblong n$ denotes that the computation may yield either m or n. As discussed by Kiselyov (2015), however, what laws they should satisfy can be a tricky issue, which eventually comes down to what we use the monad for. It is usually expected that $(a, (\talloblong), \emptyset)$ forms a monoid. That is, $(\talloblong)$ is associative, with $\emptyset$ as its zero:

$$\begin{aligned}(m \talloblong n) \talloblong k &= m \talloblong (n \talloblong k) \ ,\\ \emptyset \talloblong m = m &= m \talloblong \emptyset \ .\end{aligned}$$

It is also assumed that monadic bind distributes into $(\talloblong)$ from the end, while $\emptyset$ is a right zero for $(=\!\ll)$:

$$f =\!\ll (m_1 \talloblong m_2) = (f =\!\ll m_1) \talloblong (f =\!\ll m_2) \ , \tag{8}$$

$$f =\!\ll \emptyset = \emptyset \ . \tag{9}$$

For our purpose in this section, we also assume that $(\talloblong)$ is commutative ($m \talloblong n = n \talloblong m$) and idempotent ($m \talloblong m = m$). Implementation of such non-deterministic monads have been studied by Fischer (2011).

When mixed with other effects, the following laws holds for some monads with non-determinism, but not all:

$$(\lambda x \to f_1\, x \talloblong f_2\, x) =\!\ll m = (f_1 =\!\ll m) \talloblong (f_2 =\!\ll m) \ , \tag{10}$$

$$\emptyset \ll m = \emptyset \ . \tag{11}$$

With some implementations of the monad, it is likely that in the lefthand side of (10), the effect of m happens once, while in the righthand side it happens twice. In (11), the m on the lefthand side may incur some effects that do not happen in the righthand side. The results in this section do not depend on (10) and (11). In the next few sections we will discuss situations when they hold and do not hold.

3.1 Warming Up: Permutation and Insertion

As a warm-up example, the following function *perm* non-deterministically computes a permutation of its input, using an auxiliary function *insert* that inserts an element to an arbitrary position in a list:

```
perm            :: N ∈ ε ⇒ [a] → M_ε [a]
perm []         = return []
perm (x : xs)   = insert x =≪ perm xs  ,

insert          :: N ∈ ε ⇒ a → [a] → M_ε [a]
insert x []     = return [x]
insert x (y:xs) = return (x : y : xs) ⫿ ((y:) ⟨$⟩ insert x xs)  .
```

Thus, *perm* $[0,1,2]$ evaluates to one of the six possible lists $[0,1,2]$, $[0,2,1]$, $[1,0,2]$, $[1,2,0]$, $[2,0,1]$, or $[2,1,0]$.

It is not hard for one to formulate the following relationship between *map* and *perm*:

Lemma 3.1
$perm \cdot map\,f = map\,f \mathbin{\langle\bullet\rangle} perm.$

The lemma is true because $map\,f$ is a pure computation — in reasoning about monadic programs it is helpful, and sometimes essential, to identify its pure segments, because these are the parts where more properties are applicable. Note that the composition $(\cdot)$ on the lefthand side is turned into $(\langle\bullet\rangle)$ once we move $map\,f$ leftwards. Proof of this lemma is left as an exercise. Instead, we prove an auxiliary lemma needed to prove Lemma 3.1, to give the readers some feel of proving properties of monadic programs: [2]

Lemma 3.2
$insert\;(f\;x) \cdot map\,f = map\,f \mathbin{\langle\bullet\rangle} insert\;x.$

Proof
Prove by induction on xs that $map\,f \mathbin{\langle\$\rangle} insert\;x\;xs = insert\;(f\;x)\;(map\,f\;xs)$ for all xs. We present only the inductive case $xs := y : xs$:

$$
\begin{array}{ll}
& map\,f \mathbin{\langle\$\rangle} insert\;x\;(y:xs) \\
= & \{\text{ definition of } insert \} \\
& map\,f \mathbin{\langle\$\rangle} (return\;(x:y:xs) \mathbin{[\!]} ((y:) \mathbin{\langle\$\rangle} insert\;x\;xs)) \\
= & \{\text{ by (8) and definition of } (\langle\$\rangle) \} \\
& (map\,f \mathbin{\langle\$\rangle} return\;(x:y:xs)) \mathbin{[\!]} \\
& \quad (map\,f \mathbin{\langle\$\rangle} ((y:) \mathbin{\langle\$\rangle} insert\;x\;xs)) \quad .
\end{array}
$$

The first branch, by definition of $(\langle\$\rangle)$ and monad law (1), simplifies to $return\;(map\,f\;(x:y:xs))$. For the second branch we reason:

$$
\begin{array}{ll}
& map\,f \mathbin{\langle\$\rangle} ((y:) \mathbin{\langle\$\rangle} insert\;x\;xs) \\
= & \{\text{ by (6) }\} \\
& (map\,f \cdot (y:)) \mathbin{\langle\$\rangle} insert\;x\;xs \\
= & \{\text{ definition of } map \} \\
& ((f\;y:) \cdot map\,f) \mathbin{\langle\$\rangle} insert\;x\;xs \\
= & \{\text{ by (6) }\} \\
& (f\;y:) \mathbin{\langle\$\rangle} (map\,f \mathbin{\langle\$\rangle} insert\;x\;xs) \\
= & \{\text{ induction }\} \\
& (f\;y:) \mathbin{\langle\$\rangle} insert\;(f\;x)\;(map\,f\;xs) \quad .
\end{array}
$$

Thus we have

$$
\begin{array}{ll}
& map\,f \mathbin{\langle\$\rangle} insert\;x\;(y:xs) \\
= & \{\text{ calculation above }\} \\
& return\;(f\;x : f\;y : map\,f\;xs) \mathbin{[\!]} \\
& \quad ((f\;y:) \mathbin{\langle\$\rangle} (insert\;(f\;x)\;(map\,f\;xs))) \\
= & \{\text{ definition of } insert \text{ and } map \} \\
& insert\;(f\;x)\;(map\,f\;(y:xs)) \quad .
\end{array}
$$

□

[2] Lemma 3.1 and 3.2 are in fact free theorems of *perm* and *insert* (Voigtländer, 2009). They serve as good exercises, nevertheless.

One may have noticed that the style of proof is familiar: replace *return x* by $[x]$ and $(\llbracket\rrbracket)$ by $(+\!\!+)$, the proof is more-or-less what one would do for a list version of *insert*. This is exactly the point: the style of proofs we use to do for pure programs still works for monadic programs, as long as the monad satisfies the demanded laws, be it a list, a more advanced implementation of non-determinism, or a monad having other effects.

3.2 Example: Spark Aggregation

Spark (Zaharia *et al.*, 2012) is a popular platform for scalable distributed data-parallel computation based on a flexible programming environment with high-level APIs, considered by many as the successor of MapReduce. Programming in Spark, however, can be tricky. Since sub-results are computed across partitions concurrently, the order of their applications varies on different executions. Aggregation in Spark is therefore inherently non-deterministic. It has been reported (Chen *et al.*, 2017) that, computing the integral of x^{73} from $x = -2$ to $x = 2$, which should be 0, using a function in the Spark machine learning library, yields results ranging from -8192.0 to 12288.0 in different runs. It is thus desirable to find out conditions, which Spark's documentation does not specify formally, under which a Spark computation yields deterministic outcomes. In this section we present a model of Spark aggregation using non-deterministic monad, and derive conditions such that an aggregation is actually deterministic.

Distributed collections of data are represented by *Resilient Distributed Datasets* (RDDs) in Spark. Informally, an RDD is a collection of data entries; these data entries are further divided into partitions stored on different machines. Abstractly, an RDD can be seen as a list of lists:

$$\begin{aligned}&\textbf{type}\ \mathsf{Partition}\ a = [a]\ ,\\ &\textbf{type}\ \mathsf{RDD}\ a \quad\ = [\mathsf{Partition}\ a]\ ,\end{aligned}$$

where each Partition may be stored in a different machine.

The *aggregate* combinator intends to be a parallel implementation *foldl*. It processes an RDD in two levels: each partition is processed locally on one machine by *foldl* $(\otimes)$ z, before the sub-results are communicated and combined. This second step can be thought of as another *foldl* with $(\oplus)$. Spark programmers like to assume that their programs are deterministic. To exploit concurrency, however, the sub-results from each machine might be processed in arbitrary order and the result could be non-deterministic. This is modeled by the use of *perm*:

$$\begin{aligned}&\mathit{aggregate} :: \mathsf{N} \in \varepsilon \Rightarrow b \to (b \to a \to b) \to (b \to b \to b) \to \mathsf{RDD}\ a \to \mathsf{M}_\varepsilon\ b\\ &\mathit{aggregate}\ z\ (\otimes)\ (\oplus) =\\ &\quad \mathit{foldl}\ (\oplus)\ z \mathbin{\langle\bullet\rangle} (\mathit{perm} \cdot \mathit{map}\ (\mathit{foldl}\ (\otimes)\ z))\ \ .\end{aligned}$$

It is clear from the types that *foldl* $(\otimes)$ z and *foldl* $(\oplus)$ z are pure computations, and non-determinism is introduced solely by *perm*. The aim is to derive conditions under which *aggregate* produces deterministic outcomes.

Definition 3.3
A function $f :: \mathsf{N} \in \varepsilon \Rightarrow a \to \mathsf{M}_\varepsilon\ b$ is said to be deterministic if there exists $g :: a \to b$ such that $f = \mathit{return} \cdot g$.

The following lemma suggests a condition that makes a *foldl* after a *perm* deterministic:

Lemma 3.4
Given $(\odot) :: b \to a \to b$, we have

$$\mathit{foldl}\ (\odot)\ z \mathbin{\langle\bullet\rangle} \mathit{perm} = \mathit{return} \cdot \mathit{foldl}\ (\odot)\ z\ \ ,$$

if $(w \odot x) \odot y = (w \odot y) \odot x$ for all $x, y :: a$ and $w :: b$.

Again, we present the proof of an auxiliary lemma, with which the proof of Lemma 3.4 is relatively routine:

Lemma 3.5
Given $(\odot) :: b \to a \to b$, we have

$$\mathit{foldl}\ (\odot)\ z \mathbin{\langle\bullet\rangle} \mathit{insert}\ x = \mathit{return} \cdot \mathit{foldl}\ (\odot)\ z \cdot (+\!\!+[x])\ \ ,$$

if $(w \odot x) \odot y = (w \odot y) \odot x$ for all $x, y :: a$ and $w :: b$.

Proof
Prove by induction on xs that $\mathit{foldl}\ (\odot)\ z \mathbin{\langle\$\rangle} \mathit{insert}\ x\ xs = \mathit{return}\ (\mathit{foldl}\ (\odot)\ z\ (xs +\!\!+ [x]))$. We present only the case $xs := xs +\!\!+ [y]$. Since we are proving a property about *foldl*, we use a *foldl*-based characterization of *insert* that is equivalent to the one presented earlier in this section.

$$\begin{array}{ll}
& \mathit{foldl}\ (\odot)\ z \mathbin{\langle\$\rangle} \mathit{insert}\ x\ (xs +\!\!+ [y]) \\
= & \quad \{\ \text{definitions of } (\langle\$\rangle) \text{ and } \mathit{insert}\ \} \\
& \mathit{foldl}\ (\odot)\ z \mathbin{\langle\$\rangle} (\mathit{return}\ (xs +\!\!+ [y,x]) \mathbin{[\!]} ((+\!\!+[y]) \mathbin{\langle\$\rangle} \mathit{insert}\ x\ xs)) \\
= & \quad \{\ \text{by (8) and (1)}\ \} \\
& \mathit{return}\ (\mathit{foldl}\ (\odot)\ z\ (xs +\!\!+ [y,x])) \mathbin{[\!]} \\
& \quad ((\mathit{foldl}\ (\odot)\ z \cdot (+\!\!+[y])) \mathbin{\langle\$\rangle} \mathit{insert}\ x\ xs)\ \ .
\end{array}$$

Focus on the second branch:

$$\begin{array}{ll}
& (\mathit{foldl}\ (\odot)\ z \cdot (+\!\!+[y])) \mathbin{\langle\$\rangle} \mathit{insert}\ x\ xs \\
= & \{\ \mathit{foldl}\ (\odot)\ z\ (ws +\!\!+ [w]) = \mathit{foldl}\ (\odot)\ z\ ws \odot w\ \} \\
& ((\odot y) \cdot \mathit{foldl}\ (\odot)\ z) \mathbin{\langle\$\rangle} \mathit{insert}\ x\ xs \\
= & \{\ \text{by (4)}\ \} \\
& (\odot y) \mathbin{\langle\$\rangle} (\mathit{foldl}\ (\odot)\ z \mathbin{\langle\$\rangle} \mathit{insert}\ x\ xs) \\
= & \{\ \text{induction}\ \} \\
& (\odot y) \mathbin{\langle\$\rangle} \mathit{return}\ (\mathit{foldl}\ (\odot)\ z\ (xs +\!\!+ [x])) \\
= & \{\ \text{monad law (1)}\ \} \\
& \mathit{return}\ (\mathit{foldl}\ (\odot)\ z\ (xs +\!\!+ [x]) \odot y) \\
= & \{\ \mathit{foldl}\ (\odot)\ z\ (ws +\!\!+ [w]) = \mathit{foldl}\ (\odot)\ z\ ws \odot w\ \} \\
& \mathit{return}\ ((\mathit{foldl}\ (\odot)\ z\ xs \odot x) \odot y) \\
= & \{\ \text{since } (w \odot x) \odot y = (w \odot y) \odot x\ \} \\
& \mathit{return}\ (\mathit{foldl}\ (\odot)\ z\ (xs +\!\!+ [y,x]))\ \ .
\end{array}$$

Thus, by idempotence of $(\mathbin{[\!]})$, we have

$$(\mathit{foldl}\ (\odot)\ z \mathbin{\langle\bullet\rangle} \mathit{insert}\ x)\ (xs +\!\!+ [y]) = \mathit{return}\ (\mathit{foldr}\ (\odot)\ z\ (xs +\!\!+ [y,x]))\ \ .$$

□

We are now ready to state the main results on *aggregate*.

Theorem 3.6

Given $(\otimes) :: b \to a \to b$ and $(\oplus) :: b \to b \to b$,

$$aggregate\ z\ (\otimes)\ (\oplus) = return \cdot foldl\ (\oplus)\ z \cdot map\ (foldl\ (\otimes)\ z) \quad ,$$

if $(\oplus)$ is associative and commutative, with z as its identity.

Proof

We reason:

$$\begin{aligned}
& aggregate\ z\ (\otimes)\ (\oplus) \\
= \quad & \{ \text{ definition of } aggregate \ \} \\
& foldl\ (\oplus)\ z \mathbin{\langle\bullet\rangle} (perm \cdot map\ (foldl\ (\otimes)\ z)) \\
= \quad & \{ \text{ Lemma 3.1 } \} \\
& foldl\ (\oplus)\ z \mathbin{\langle\bullet\rangle} (map\ (foldl\ (\otimes)\ z) \mathbin{\langle\bullet\rangle} perm) \\
= \quad & \{ \text{ by (7) } \} \\
& (foldl\ (\oplus)\ z \cdot map\ (foldl\ (\otimes)\ z)) \mathbin{\langle\bullet\rangle} perm \\
= \quad & \{ \text{ Lemma 3.4 } \} \\
& return \cdot foldl\ (\oplus)\ z \cdot map\ (foldl\ (\otimes)\ z) \quad .
\end{aligned}$$

The last step holds because by *fold*-*map* fusion, for all h,

$$\begin{aligned}
& foldl\ (\oplus)\ z \cdot map\ h = foldl\ (\odot)\ z \\
& \quad \textbf{where}\ w \odot xs = w \oplus h\ xs \quad ,
\end{aligned}$$

and $(\odot)$ satisfies that $(w \odot xs) \odot ys = (w \odot ys) \odot xs$ if $(\oplus)$ is associative and commutative.
□

4 Example: The *n* Queens Problem

Reasoning about monadic programs gets more interesting when more than one effect is involved. In the next few sections, we consider programs that are both stateful and non-deterministic. Backtracking algorithms make good examples of such programs, and the 8 queens problem, also dealt with by Gibbons and Hinze (2011), is among the most well-known examples of backtracking.[3]

In this section we present a specification of the problem, as well as some initial calculation that will be used in later sections. The specification is non-deterministic, but not stateful, having the form of a fold after an unfold. In Section 5 and Section 6 we derive two non-deterministic *and* stateful programs, under different assumption of the interaction between non-determinism and state.

	0	1	2	3	4	5	6	7
0	.	.	.	.	.	*Q*	.	.
1	.	.	.	*Q*	.	.	.	.
2	.	.	.	.	.	.	*Q*	.
3	*Q*	.	.	.	.	.	.	.
4	.	.	.	.	.	.	.	*Q*
5	.	*Q*	.	.	.	.	.	.
6	.	.	.	.	*Q*	.	.	.
7	.	.	*Q*	.	.	.	.	.

(a)

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	.	.	.
1	1	2	3	4	.	.	.	.
2	2	3	4	.	.	.	.	.
3	3	4	.	.	.	.	.	.
4	4	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	12
6	.	.	.	.	.	.	12	13
7	.	.	.	.	.	12	13	14

(b)

	0	1	2	3	4	5	6	7
0	0	−1	.	.	.	−5	−6	−7
1	.	0	−1	.	.	.	−5	−6
2	.	.	0	−1	.	.	.	−5
3	3	.	.	0	.	.	.	.
4	4	3	.	.	0	.	.	.
5	5	4	3	.	.	0	.	.
6	6	5	4	3	.	.	0	.
7	7	6	5	4	3	.	.	0

(c)

Fig. 3: (a) This placement can be represented by $[3,5,7,1,6,0,2,4]$. (b) Up diagonals. (c) Down diagonals.

4.1 Specification

The aim of the puzzle is to place *n* queens on a *n* by *n* chess board such that no two queens can attack each other. Given *n*, we number the rows and columns by $[0\,..\,n-1]$. Since all queens should be placed on distinct rows and distinct columns, a potential solution can be represented by a permutation *xs* of the list $[0\,..\,n-1]$, such that $xs\,!!\,i = j$ denotes that the queen on the *i*th column is placed on the *j*th row (see Figure 3(a)). In this representation queens cannot be put on the same row or column, and the problem is reduced to filtering, among permutations of $[0\,..\,n-1]$, those placements in which no two queens are put on the same diagonal. The specification can be written as:

$$\begin{array}{l} \mathit{queens} :: \mathsf{N} \in \varepsilon \Rightarrow \mathsf{Int} \to \mathsf{M}_\varepsilon\;[\mathsf{Int}] \\ \mathit{queens}\;n = \mathit{assert}\;\mathit{safe} \mathrel{=\!\!\ll} \mathit{perm}\;[0\,..\,n-1] \;\;, \end{array}$$

where the pure function $\mathit{safe} :: [\mathsf{Int}] \to \mathsf{Bool}$ determines whether no queens are on the same diagonal. The monadic function $\mathit{assert}\;p\;x$ returns x if $p\;x$ holds, and fails otherwise:

$$\begin{array}{l} \mathit{assert} :: \mathsf{N} \in \varepsilon \Rightarrow (a \to \mathsf{Bool}) \to a \to \mathsf{M}_\varepsilon\;a \\ \mathit{assert}\;p\;x = \mathit{return}\;x \ll \mathit{guard}\;(p\;x) \;\;, \end{array}$$

where *guard* is a standard monadic function defined by:

$$\begin{array}{l} \mathit{guard} :: \mathsf{N} \in \varepsilon \Rightarrow \mathsf{Bool} \to \mathsf{M}_\varepsilon\;() \\ \mathit{guard}\;b = \mathbf{if}\;b\;\mathbf{then}\;\mathit{return}\;()\;\mathbf{else}\;\emptyset \;\;. \end{array}$$

The function *perm* has been defined in Section 3.1 but will be redefined later.

Representing a placement as a permutation also allows an easy way to check whether two queens are put on the same diagonal. A 8 by 8 chess board has 15 *up diagonals* (those running between bottom-left and top-right). Let them be indexed by $[0\,..\,14]$ (see Figure 3(b)). If we apply $\mathit{zipWith}\;(+)\;[0\,..]$ to a permutation, we get the indices of the up-diagonals where the chess pieces are placed. Similarly, there are 15 *down diagonals* (those running between top-left and bottom right). By applying $\mathit{zipWith}\;(-)\;[0\,..]$ to a permutation, we get

[3] Curiously, Gibbons and Hinze (2011) did not finish their derivation and stopped at a program that exhaustively generates all permutations and tests each of them. Perhaps it was sufficient to demonstrate their point.

the indices of their down-diagonals (indexed by $[-7\,..\,7]$. See Figure 3(c)). A placement is safe if the diagonals contain no duplicates:

$$\begin{aligned}
&\mathit{ups}, \mathit{downs} :: [\mathsf{Int}] \to [\mathsf{Int}]\\
&\mathit{ups}\ \ \ \ \ \mathit{xs} = \mathit{zipWith}\ (+)\ [0\,..]\ \mathit{xs}\ \ ,\\
&\mathit{downs}\ \mathit{xs} = \mathit{zipWith}\ (-)\ [0\,..]\ \mathit{xs}\ \ ,\\
&\mathit{safe}\ \ \ \ :: [\mathsf{Int}] \to \mathsf{Bool}\\
&\mathit{safe}\ \mathit{xs} = \mathit{nodup}\ (\mathit{ups}\ \mathit{xs}) \wedge \mathit{nodup}\ (\mathit{downs}\ \mathit{xs})\ \ ,
\end{aligned}$$

where $\mathit{nodup} :: \mathsf{Eq}\ a \Rightarrow [a] \to \mathsf{Bool}$ determines whether there is no duplication in a list.

This specification of *queens* generates all the permutations, before checking them one by one, in two separate phases. We wish to fuse the two phases and produce a faster implementation. The overall idea is to transform *assert safe* into a fold, let *perm* be an unfold, and fuse the two phases into a *hylomorphism*. During the fusion, some non-safe choices can be pruned off earlier, speeding up the computation.

In the rest of this section, we present some general calculations and theorems that can be reused in other sections.

4.2 Safety Check as a Stateful foldr

In this section we try to express *safe* in terms of a stateful *foldr*. For that we will need an inductive definition of *safe*, and to derive such a definition we need to generalise *safe* to *safeAcc* below:

$$\begin{aligned}
&\mathit{safeAcc}\ (i, \mathit{us}, \mathit{ds})\ \mathit{xs} = \mathit{nodup}\ \mathit{us}' \wedge \mathit{nodup}\ \mathit{ds}' \wedge\\
&\qquad\qquad\qquad\qquad\quad \mathit{all}\ (\notin \mathit{us})\ \mathit{us}' \wedge \mathit{all}\ (\notin \mathit{ds})\ \mathit{ds}'\ \ ,\\
&\quad \textbf{where}\ \mathit{us}' = \mathit{zipWith}\ (+)\ [i\,..]\ \mathit{xs}\\
&\qquad\qquad\ \ \mathit{ds}' = \mathit{zipWith}\ (-)\ [i\,..]\ \mathit{xs}\ \ ,
\end{aligned}$$

such that $\mathit{safe} = \mathit{safeAcc}\ (0, [\,], [\,])$. Recall that the standard list function *scanl* applies *foldl* to all prefixes of a list. By plain functional calculation, one may conclude that *safeAcc* can be defined using a variation of *scanl*:

$$\begin{aligned}
&\mathit{safeAcc}\ (i, \mathit{us}, \mathit{ds}) = \mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ (i, \mathit{us}, \mathit{ds})\ \ ,\\
&\quad \textbf{where}\ (i, \mathit{us}, \mathit{ds}) \oplus x = (i+1, (i+x : \mathit{us}), (i-x : \mathit{ds}))\\
&\qquad\qquad \mathit{ok}\ (i, (x : \mathit{us}), (y : \mathit{ds})) = x \notin \mathit{us} \wedge y \notin \mathit{ds}\ \ ,
\end{aligned}$$

where $\mathit{all}\ p = \mathit{foldr}\ (\wedge)\ \mathsf{True} \cdot \mathit{map}\ p$ and scanl_{+} is like *scanl*, but applies *foldl* to all non-empty prefixes of a list:

$$\begin{aligned}
&\mathit{scanl}_{+} :: (b \to a \to b) \to b \to [a] \to [b]\\
&\mathit{scanl}_{+}\ (\oplus)\ \mathit{st}\ [\,]\qquad\ \ = [\,]\\
&\mathit{scanl}_{+}\ (\oplus)\ \mathit{st}\ (x : \mathit{xs}) = (\mathit{st} \oplus x) : \mathit{scanl}_{+}\ (\oplus)\ (\mathit{st} \oplus x)\ \mathit{xs}\ \ .
\end{aligned}$$

Operationally, *safeAcc* examines the list from left to right, while keeping a state $(i, \mathit{us}, \mathit{ds})$, where i is the current position being examined, while *us* and *ds* are respectively indices of all the up and down diagonals encountered so far. Indeed, in a function call $\mathit{scanl}_{+}\ (\oplus)\ \mathit{st}$, the value *st* can be seen as a "state" that is explicitly carried around. This naturally leads to the question: can we convert a scanl_{+} to a monadic program that stores *st* in its state?

State The state effect provides two operators: $get :: \mathsf{S}\ s \in \varepsilon \Rightarrow \mathsf{M}_\varepsilon\ s$ retrieves the state, while $put :: \mathsf{S}\ s \in \varepsilon \Rightarrow s \to \mathsf{M}_\varepsilon\ ()$ overwrites the state by the given value. They are supposed to satisfy the laws:

$$put\ st \gg put\ st' = put\ st' \ , \tag{12}$$

$$put\ st \gg get = put\ st \gg return\ st \ , \tag{13}$$

$$get \gg\!= put = return\ () \ , \tag{14}$$

$$get \gg\!= (\lambda st \to get \gg\!= k\ st) = get \gg\!= (\lambda st \to k\ st\ st) \ . \tag{15}$$

In this section we mostly use the usual left-to-right $(\gg\!=)$ and $(\gg)$, to be consistent with the direction of λ-abstraction. We define:

$$overwrite\ st\ x = put\ st \gg return\ x \ ,$$

such that $m \gg\!= overwrite\ st$ overwrites the state to st while returning the result of m.

From $scanl_+$ **to monadic** *foldr* The following function traverses the given list while maintaining a monadic state updated by operator $(\oplus)$:

```
$loop_+ :: \mathsf{S}\ s \in \varepsilon \Rightarrow (s \to a \to s) \to s \to [a] \to \mathsf{M}_\varepsilon\ [s]$
$loop_+\ (\oplus)\ st\ xs = put\ st \gg foldr\ (\otimes)\ (return\ [\,])\ xs$
  $\mathbf{where}\ x \otimes m = get \gg\!= \lambda st \to \mathbf{let}\ st' = st \oplus x$
                        $\mathbf{in}\ (st'{:}) \mathbin{\langle\$\rangle} (put\ st' \gg m)$ ,
```

We can convert $scanl_+$ to $loop_+$ in the sense below. Note that *get* and *overwrite* retrieves and restores the state in the context.

Theorem 4.1
For all $(\oplus) :: (s \to a \to s)$, $st :: s$, and $xs :: [a]$,

$$\begin{array}{l} return\ (scanl_+\ (\oplus)\ st\ xs) = \\ \quad get \gg\!= \lambda ini \to loop_+\ (\oplus)\ st\ xs \gg\!= overwrite\ ini \ , \end{array}$$

Proof
By induction on xs. We present the case $xs := x : xs$.

$$\begin{array}{cl}
 & get \gg\!= \lambda ini \to loop_+\ (\oplus)\ st\ (x : xs) \gg\!= overwrite\ ini \\
= & \{ \text{ expanding definitions, let } st' = st \oplus x \ \} \\
 & get \gg\!= \lambda ini \to put\ st \gg get \gg\!= \lambda st \to \\
 & ((st'{:}) \mathbin{\langle\$\rangle} (put\ st' \gg foldr\ (\otimes)\ (return\ [\,])\ xs)) \gg\!= \lambda r \to \\
 & put\ ini \gg return\ r \\
= & \{ \text{ by } put\text{-}get\ (13) \ \} \\
 & get \gg\!= \lambda ini \to put\ st \gg \\
 & ((st'{:}) \mathbin{\langle\$\rangle} (put\ st' \gg foldr\ (\otimes)\ (return\ [\,])\ xs)) \gg\!= \lambda r \to \\
 & put\ ini \gg return\ r \\
= & \{ \text{ definition of } (\langle\$\rangle) \text{ and monad laws } \} \\
 & (st'{:}) \mathbin{\langle\$\rangle} (get \gg\!= \lambda ini \to put\ st \gg put\ st' \gg \\
 & \qquad foldr\ (\otimes)\ (return\ [\,])\ xs \gg\!= \lambda r \to \\
 & \qquad put\ ini \gg return\ r)
\end{array}$$

$$
\begin{aligned}
&= \quad \{\text{ by } \mathit{put\text{-}put}\ (12)\ \}\\
&\quad (st'\!:) \mathbin{\langle\$\rangle} (get \gg\!= \lambda ini \to put\ st' \gg foldr\ (\otimes)\ (return\ [\,])\ xs\\
&\qquad\qquad \gg\!= \lambda r \to put\ ini \gg return\ r)\\
&= \quad \{\text{ definitions of } loop_{+} \text{ and } overwrite\ \}\\
&\quad (st'\!:) \mathbin{\langle\$\rangle} (get \gg\!= \lambda ini \to loop_{+}\ (\oplus)\ st'\ xs \gg overwrite\ ini)\\
&= \quad \{\text{ induction }\}\\
&\quad (st'\!:) \mathbin{\langle\$\rangle} return\ (scanl_{+}\ (\oplus)\ st'\ xs)\\
&= return\ ((st \oplus x) : scanl_{+}\ (\oplus)\ (st \oplus x)\ xs)\\
&= return\ (scanl_{+}\ (\oplus)\ st\ (x : xs)) \quad .
\end{aligned}
$$

□

This proof is instructive due to the use of properties (12) and (13), and that $(st'\!:)$, being a pure function, can be easily moved around.

We have turned $scanl_{+}$ into a stateful *foldr*. Non-determinism is not yet involved. In Section 5 and 6 we will turn *assert* (*safeAcc st*) to a *foldr* that is stateful and non-deterministic, to be fused with *perm* to form a hylomorphism.

4.3 Permutation as an Unfold

Now that *safeAcc* examines its input from left to right, it is preferable that *perm* also generates the list from left to right. The function *select* non-deterministically splits a list into a pair containing one chosen element and the rest:

$$
\begin{aligned}
&select :: \mathsf{N} \in \varepsilon \Rightarrow [a] \to \mathsf{M}_{\varepsilon}\ (a,[a]) \quad .\\
&select\ [\,] \qquad\ = \emptyset\\
&select\ (x : xs) = return\ (x, xs) \mathbin{[\!]} ((id \times (x:)) \mathbin{\langle\$\rangle} select\ xs) \quad .
\end{aligned}
$$

For example, *select* $[1,2,3]$ yields one of $(1,[2,3])$, $(2,[1,3])$ and $(3,[1,2])$. The function call *unfoldM p f y* generates a list $[a]$ from a seed $y :: b$. If $p\ y$ holds, the generation stops. Otherwise an element and a new seed is generated using f. It is like the usual *unfoldr* apart from that f, and thus the result, is monadic:

$$
\begin{aligned}
&unfoldM :: (b \to \mathsf{Bool}) \to (b \to \mathsf{M}_{\varepsilon}\ (a,b)) \to b \to \mathsf{M}_{\varepsilon}\ [a]\\
&unfoldM\ p\ f\ y \mid p\ y \qquad\quad = return\ [\,]\\
&\qquad\qquad\qquad \mid otherwise = f\ y \gg\!= \lambda(x,z) \to (x:) \mathbin{\langle\$\rangle} unfoldM\ p\ f\ z \quad .
\end{aligned}
$$

Given these definitions, *perm* can be defined by:

$$
\begin{aligned}
&perm :: \mathsf{N} \in \varepsilon \Rightarrow [a] \to \mathsf{M}_{\varepsilon}\ [a]\\
&perm = unfoldM\ null\ select \quad .
\end{aligned}
$$

4.4 Monadic Hylo-Fusion

If *assert safe* can be turned into a fold, with *perm* being an unfold, the natural thing to do is to fuse them into a hylomorphism (Meijer *et al.*, 1991). In a pure setting, it is known that, provided that the unfolding phase terminates, $foldr\ (\otimes)\ e \cdot unfoldr\ p\ f$ is the unique solution of *hylo* in the equation below (Hinze *et al.*, 2015):

$$\begin{aligned}&\mathit{hylo}\; y \mid p\; y &&= e\\ &\qquad\mid \mathit{otherwise} &&= \mathbf{let}\, f\; y = (x,z)\;\mathbf{in}\; x \otimes \mathit{hylo}\; z\quad .\end{aligned}$$

For the monadic fold and unfold defined previously, a similar result holds — with a side condition. Given $(\otimes) :: b \to \mathsf{M}_\varepsilon\; c \to \mathsf{M}_\varepsilon\; c$, $e :: c$, $p :: a \to \mathsf{Bool}$, and $f :: a \to \mathsf{M}_\varepsilon\; (b,a)$ (with no restriction on ε), consider the expression:

$$\mathit{foldr}\; (\otimes)\; (\mathit{return}\; e) \Leftarrow\!\!\!\!< \mathit{unfoldM}\; p\, f \;::\; a \to \mathsf{M}_\varepsilon\; c\quad .$$

The following theorem says that this combination of folding and unfolding can be fused into one if f eventually terminates (in the sense explained at the end of this section) and for all x and m, $(x\otimes)$ commutes with m in the sense that $x \otimes m = (m \gg\!= \lambda y \to x \otimes \mathit{return}\; y)$. That is, it does not matter whether the effects of m happens inside or outside $(x\otimes)$.

Theorem 4.2
We have that $\mathit{foldr}\; (\otimes)\; (\mathit{return}\; e) \Leftarrow\!\!\!\!< \mathit{unfoldM}\; p\, f = \mathit{hyloM}\; (\otimes)\; e\; p\, f$, defined by:

$$\begin{aligned}&\mathit{hyloM}\; (\otimes)\; e\; p\, f\; y\\ &\quad\mid p\; y &&= \mathit{return}\; e\\ &\quad\mid \mathit{otherwise} &&= f\; y \gg\!= \lambda(x,z) \to x \otimes \mathit{hyloM}\; (\otimes)\; e\; p\, f\; z\quad ,\end{aligned}$$

if $x \otimes m = m \gg\!= ((x\otimes)\cdot \mathit{return})$ for all $m :: \mathsf{M}_\varepsilon\; c$, and that the relation $(\neg \cdot p)\,?\cdot \mathit{snd} \cdot (=\!\ll) \cdot f$ is well-founded. (See the note below.)

Proof
We start with showing that $\mathit{foldr}\; (\otimes)\; (\mathit{return}\; e) \Leftarrow\!\!\!\!< \mathit{unfoldM}\; p\, f$ is a fixed-point of the recursive equations of *hyloM*. When $p\; y$ holds, it is immediate that

$$\mathit{foldr}\; (\otimes)\; (\mathit{return}\; e) =\!\ll \mathit{return}\; [\,] \;=\; \mathit{return}\; e\quad .$$

When $\neg\, (p\; y)$, we reason:

$$\begin{aligned}&\mathit{unfoldM}\; p\, f\; y \gg\!= \mathit{foldr}\; (\otimes)\; (\mathit{return}\; e)\\ =\;&\quad\{\text{ definition of } \mathit{unfoldM},\ \neg\,(p\; y)\ \}\\ &(f\; y \gg\!= (\lambda(x,z) \to (x{:}) \mathbin{\langle\$\rangle} \mathit{unfoldM}\; p\, f\; z)) \gg\!=\\ &\mathit{foldr}\; (\otimes)\; (\mathit{return}\; e)\\ =\;&\quad\{\text{ monadic law (1) and } \mathit{foldr}\ \}\\ &f\; y \gg\!= (\lambda(x,z) \to \mathit{unfoldM}\; p\, f\; z \gg\!= \lambda xs \to\\ &\qquad\quad x \otimes \mathit{foldr}\; (\otimes)\; (\mathit{return}\; e)\; xs)\quad .\end{aligned}$$

We focus on the expression inside the λ abstraction:

$$\begin{aligned}&\mathit{unfoldM}\; p\, f\; z \gg\!= (\lambda xs \to x \otimes \mathit{foldr}\; (\otimes)\; (\mathit{return}\; e)\; xs)\\ =\;&\quad\{\text{ assumption: } x \otimes m = m \gg\!= ((x\otimes)\cdot \mathit{return})\text{, see below }\}\\ &\mathit{unfoldM}\; p\, f\; z \gg\!= (\mathit{foldr}\; (\otimes)\; (\mathit{return}\; e) >\!=\!> ((x\otimes)\cdot \mathit{return}))\\ =\;&\quad\{\text{ since } m \gg\!= (f >\!=\!> g) = (m \gg\!= f) \gg\!= g\ \}\\ &(\mathit{unfoldM}\; p\, f\; z \gg\!= \mathit{foldr}\; (\otimes)\; (\mathit{return}\; e)) \gg\!= ((x\otimes)\cdot \mathit{return})\\ =\;&\quad\{\text{ by assumption } x \otimes m = m \gg\!= ((x\otimes)\cdot \mathit{return})\ \}\\ &x \otimes (\mathit{unfoldM}\; p\, f\; z \gg\!= \mathit{foldr}\; (\otimes)\; (\mathit{return}\; e))\quad .\end{aligned}$$

To understand the first step, note that $h\; xs \gg\!= ((x\otimes)\cdot \mathit{return}) \;=\; (h >\!=\!> ((x\otimes)\cdot \mathit{return}))\; xs$.

Now that $\mathit{unfoldM}\ p\ f\ z \gg= \mathit{foldr}\ (\otimes)\ (\mathit{return}\ e)$ is a fixed-point, we may conclude that it equals $\mathit{hyloM}\ (\otimes)\ e\ p\ f$ if the latter has a unique fixed-point. See the note below. □

Note Let q be a predicate, $q?$ is a relation defined by $\{(x,x) \mid q\ x\}$. The parameter y in *unfoldM* is called the *seed* used to generate the list. The relation $(\neg \cdot p)\,? \cdot \mathit{snd} \cdot (=\ll) \cdot f$ maps one seed to the next seed. If it is *well-founded*, intuitively speaking, the seed generation cannot go on forever and p will eventually hold. It is known that inductive types (those can be folded) and coinductive types (those can be unfolded) do not coincide in SET. To allow a fold to be composed after an unfold, typically one moves to a semantics based on complete partial orders. However, it was shown (Doornbos & Backhouse, 1995) that, in Rel, when the relation generating seeds is well-founded, hylo-equations do have unique solutions. One may thus stay within a set-theoretic semantics. Such an approach is recently explored again (Hinze *et al.*, 2015).

5 Non-Determinism with Local State

In this section and the next, we convert $\mathit{assert}\ (\mathit{safeAcc}\ \mathit{st})$ to a *foldr* that is both non-deterministic and stateful, before deriving a more efficient algorithm by fusion.

We assume in this section that (10) and (11) hold. With them, we get many useful properties, which make this stateful non-determinism monad preferred for program calculation and reasoning. In particular, non-determinism commutes with other effects.

Definition 5.1
Let $m :: \mathsf{M}_\varepsilon\ a$ where x does not occur free, and $n :: \mathsf{M}_\delta\ b$ where y does not occur free. We say m and n commute if

$$\begin{array}{r}m \gg= \lambda x \to n \gg= \lambda y \to f\ x\ y\ = \\ n \gg= \lambda y \to m \gg= \lambda x \to f\ x\ y\ \ .\end{array} \qquad (16)$$

(Notice that m and n can also be typed as $\mathsf{M}_{\varepsilon\cup\delta}$.) We say that m commutes with effect δ if m commutes with any n of type $\mathsf{M}_\delta\ b$, and that effects ε and δ commute if any $m :: \mathsf{M}_\varepsilon\ a$ and $n :: \mathsf{M}_\delta\ b$ commute.

Theorem 5.2
If (10) and (11) hold in addition to the monad laws stated before, non-determinism commutes with any effect ε.

Proof
By induction on syntax of n. □

Operationally, one consequence of (10) is that $\mathit{get} \gg= (\lambda s \to f_1\ s \mathbin{[\!]} f_2\ s) = (\mathit{get} \gg= f_1 \mathbin{[\!]} \mathit{get} \gg= f_2)$ — f_1 and f_2 get the same state regardless of whether *get* is performed outside or inside the non-deterministic branch. It implies that each non-deterministic branch has its own copy of the state (thus the effect in f_1 does not alter the state in f_2). One monad having such property is $\mathsf{M}_{\{\mathsf{N},\mathsf{S}\ s\}}\ a = s \to [(a,s)]$, which is the same monad one gets by $\mathsf{StateT}\ s\ (\mathsf{ListT}\ \mathsf{Identity})$ in the Monad Transformer Library (Gill & Kmett, 2014). With effect handling, the monad meets the requirements if we run the handler for state before that for list.

5.1 Safety in a Stateful, Non-Deterministic Fold

Recall that $\mathit{queens}\ n = \mathit{assert}\ \mathit{safe} =\!\!\ll \mathit{perm}\ [0\,..\,n-1]$, where *safe* can be generalised to $\mathit{safeAcc}\ (0,[\,],[\,])$, which can in turn be defined by $\mathit{safeAcc}\ (i,\mathit{us},\mathit{ds}) = \mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ (i,\mathit{us},\mathit{ds})$. In this section we aim to turn an assertion of the form $\mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ \mathit{st})$ to a sateful and non-deterministic *foldr*.

We calculate, for all *ok*, $(\oplus)$, *st*, and *xs*:

$$\begin{array}{ll}
& \mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ \mathit{st})\ \mathit{xs} \\
= & \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ (\mathit{scanl}_{+}\ (\oplus)\ \mathit{st}\ \mathit{xs})) \gg \mathit{return}\ \mathit{xs} \\
= & \mathit{return}\ (\mathit{scanl}_{+}\ (\oplus)\ \mathit{st}\ \mathit{xs}) \gg\!= \lambda \mathit{ys} \rightarrow \\
& \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ \mathit{xs} \\
= & \quad \{\ \text{Theorem 4.1, definition of } \mathit{loop}_{+}\ \} \\
& \mathit{get} \gg\!= \lambda \mathit{ini} \rightarrow \mathit{loop}_{+}\ (\oplus)\ \mathit{st}\ \mathit{xs} \gg\!= \mathit{overwrite}\ \mathit{ini} \gg\!= \lambda \mathit{ys} \rightarrow \\
& \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ \mathit{xs} \\
= & \quad \{\ \text{non-determinism commutes with state}\ \} \\
& \mathit{get} \gg\!= \lambda \mathit{ini} \rightarrow \mathit{loop}_{+}\ (\oplus)\ \mathit{st}\ \mathit{xs} \gg\!= \lambda \mathit{ys} \rightarrow \\
& \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ \mathit{xs} \gg\!= \mathit{overwrite}\ \mathit{ini}
\end{array}$$

Recall that $\mathit{loop}_{+}\ (\oplus)\ \mathit{st}\ \mathit{xs} = \mathit{put}\ \mathit{st} \gg \mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs}$. Consider the internal *foldr*:

Theorem 5.3
Assume that state and non-determinism commute. Let $(\otimes)$ be defined as that in loop_{+} for any given $(\oplus) :: s \rightarrow a \rightarrow s$. We have that for all $\mathit{ok} :: s \rightarrow \mathsf{Bool}$ and $\mathit{xs} :: [a]$:

$$\begin{array}{l}
\mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs} \gg\!= \lambda \mathit{ys} \rightarrow \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ \mathit{xs} = \\
\quad \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ \mathit{xs} \quad , \\
\quad \mathbf{where}\ x \odot m = \mathit{get} \gg\!= \lambda \mathit{st} \rightarrow \mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg \\
\qquad\qquad\qquad\quad \mathit{put}\ (\mathit{st} \oplus x) \gg ((x{:})\ \langle\$\rangle\ m) \quad .
\end{array}$$

Proof
Unfortunately we cannot use a *foldr* fusion, since *xs* occurs free in $\lambda \mathit{ys} \rightarrow \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ \mathit{xs}$. Instead we use a simple induction on *xs*. For the case $\mathit{xs} := x : \mathit{xs}$:

$$\begin{array}{ll}
& (x \otimes \mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs}) \gg\!= \lambda \mathit{ys} \rightarrow \\
& \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ (x : \mathit{xs}) \\
= & \quad \{\ \text{definition of } (\otimes)\ \} \\
& \mathit{get} \gg\!= \lambda \mathit{st} \rightarrow \\
& (((\mathit{st} \oplus x){:})\ \langle\$\rangle\ (\mathit{put}\ (\mathit{st} \oplus x) \gg \mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs})) \gg\!= \lambda \mathit{ys} \rightarrow \\
& \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ (x : \mathit{xs}) \\
= & \quad \{\ \text{monad laws and definition of } (\langle\$\rangle)\ \} \\
& \mathit{get} \gg\!= \lambda \mathit{st} \rightarrow \mathit{put}\ (\mathit{st} \oplus x) \gg \\
& \mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs} \gg\!= \lambda \mathit{ys} \rightarrow \\
& \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ (\mathit{st} \oplus x : \mathit{ys})) \gg \mathit{return}\ (x : \mathit{xs}) \\
= & \quad \{\ \text{since } \mathit{guard}\ (p \wedge q) = \mathit{guard}\ q \gg \mathit{guard}\ p\ \} \\
& \mathit{get} \gg\!= \lambda \mathit{st} \rightarrow \mathit{put}\ (\mathit{st} \oplus x) \gg \\
& \mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs} \gg\!= \lambda \mathit{ys} \rightarrow \\
& \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg
\end{array}$$

$$
\begin{aligned}
&\quad \mathit{return}\ (x : xs) \\
=&\quad \{ \text{nondeterminism commutes with state} \} \\
&\quad \mathit{get} \gg\!= \lambda \mathit{st} \to \mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg \mathit{put}\ (\mathit{st} \oplus x) \gg \\
&\quad \mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs} \gg\!= \lambda \mathit{ys} \to \\
&\quad \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ (x : xs) \\
=&\quad \{ \text{monad laws and definition of } (\langle\$\rangle) \} \\
&\quad \mathit{get} \gg\!= \lambda \mathit{st} \to \mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg \mathit{put}\ (\mathit{st} \oplus x) \gg \\
&\quad (x{:}) \langle\$\rangle (\mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs} \gg\!= \lambda \mathit{ys} \to \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ \mathit{xs}) \quad .
\end{aligned}
$$

□

This proof is instructive due to extensive use of commutativity.

5.2 *n* Queens Solved

Return to the specification of *queens*:

$$
\begin{aligned}
&\quad \mathit{queens}\ n \\
=&\ \mathit{assert}\ \mathit{safe} =\!\ll \mathit{perm}\ [0\,..\,n-1] \\
=&\quad \{ \text{Theorem 4.1 and 5.3} \} \\
&\quad \mathit{perm}\ [0\,..\,n-1] \gg\!= \lambda \mathit{xs} \to \\
&\quad \mathit{get} \gg\!= \lambda \mathit{ini} \to \mathit{put}\ (0, [\,], [\,]) \gg \\
&\quad \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ \mathit{xs} \gg\!= \mathit{overwrite}\ \mathit{ini} \\
=&\quad \{ \text{nondeterminism commutes with state} \} \\
&\quad \mathit{get} \gg\!= \lambda \mathit{ini} \to \mathit{put}\ (0, [\,], [\,]) \gg \\
&\quad \mathit{perm}\ [0\,..\,n-1] \gg\!= \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,]) \gg\!= \mathit{overwrite}\ \mathit{ini} \quad .
\end{aligned}
$$

Define $\mathit{queensBody}\ \mathit{xs} = \mathit{perm}\ \mathit{xs} \gg\!= \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])$. We have

$$
\begin{aligned}
\mathit{queens}\ n = \mathit{get} \gg\!= \lambda \mathit{ini} \to \mathit{put}\ (0, [\,], [\,]) \gg \\
\mathit{queensBody}\ [0\,..\,n-1] \gg\!= \mathit{overwrite}\ \mathit{ini} \quad .
\end{aligned}
$$

Consider $\mathit{perm} \gg\!\!=\!> \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])$. The function *select* cannot be applied forever since the length of the given list decreases after each call. To apply Theorem 4.2, we need $x \odot m = m \gg\!= ((x\odot) \cdot \mathit{return})$ to hold. Fortunately it is indeed the case. In the lemma below we slightly generalise $(\odot)$ in Theorem 5.3:

Lemma 5.4
Assuming that state and non-determinism commute, and $m \gg\!= \emptyset = \emptyset$. Given $p :: a \to s \to \mathsf{Bool}$, $\mathit{next} :: a \to s \to s$, $\mathit{res} :: a \to b \to b$, define $(\odot) :: a \to \mathsf{M}_\varepsilon\ b \to \mathsf{M}_\varepsilon\ b$ with $\{\mathsf{N}, \mathsf{S}\ s\} \subseteq \varepsilon$:

$$
\begin{aligned}
x \odot m = \mathit{get} \gg\!= \lambda \mathit{st} \to \mathit{guard}\ (p\ x\ \mathit{st}) \gg \\
\mathit{put}\ (\mathit{next}\ x\ \mathit{st}) \gg (\mathit{res}\ x \langle\$\rangle m) \quad .
\end{aligned}
$$

We have $x \odot m = ((x\odot) \cdot \mathit{return}) =\!\ll m$.

Proof
Routine, using commutativity of state and non-determinism. □

Thus we have $\mathit{queensBody} = \mathit{hyloM}\ (\odot)\ [\,]\ \mathit{null}\ \mathit{select}$. Expanding the definition we get:

$$
\begin{array}{l}
\mathit{queensBody} :: \{\mathsf{N}, \mathsf{S}\ (\mathsf{Int}, [\mathsf{Int}], [\mathsf{Int}])\} \subseteq \varepsilon \Rightarrow [\mathsf{Int}] \to \mathsf{M}_\varepsilon\ [\mathsf{Int}] \\
\mathit{queensBody}\ [\,] = \mathit{return}\ [\,] \\
\mathit{queensBody}\ \mathit{xs} = \mathit{select}\ \mathit{xs} \gg\!= \lambda(x, \mathit{ys}) \to \\
\qquad\qquad \mathit{get} \gg\!= \lambda \mathit{st} \to \mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg \\
\qquad\qquad \mathit{put}\ (\mathit{st} \oplus x) \gg ((x{:})\ \langle\$\rangle\ \mathit{queensBody}\ \mathit{ys}) \quad , \\
\quad \textbf{where}\ (i, \mathit{us}, \mathit{ds}) \oplus x = (1 + i, (i + x) : \mathit{us}, (i - x) : \mathit{ds}) \\
\qquad\quad \mathit{ok}\ (\_, u : \mathit{us}, d : \mathit{ds}) = (u \notin \mathit{us}) \wedge (d \notin \mathit{ds}) \quad .
\end{array}
$$

Thus completes the derivation of our first algorithm for the *n*-queens problem.

6 Non-Determinism with Global State

For a monad with both non-determinism and state, property (10) implies that each non-deterministic branch has its own state. This is not costly for states consisting of linked data structures, for example the state $(\mathsf{Int}, [\mathsf{Int}], [\mathsf{Int}])$ in the *n*-queens problem. In some applications, however, the state might be represented by data structures, e.g. arrays, that are costly to duplicate. For such practical concerns, it is worth considering the situation when all non-deterministic branches share one global state.

Non-deterministic monads with a global state, however, turns out to be tricky. One might believe that $\mathsf{M}\ a = s \to ([a], s)$ is a natural implementation of such a monad. The usual, naive implementation of $(\gg\!=)$ using this representation, however, does not satisfy (8), consequently violates monad laws, and is therefore not even a monad. The type $\mathsf{ListT}\ (\mathsf{State}\ s)$ generated using the now standard Monad Transformer Library expands to essentially the same implementation, and is flawed in the same way. More careful implementations of ListT, which does satisfy (8) and the monad laws, have been proposed (Gale, 2007; Volkov, 2014). It is nevertheless surprising that the problem could have persisted quite a while before being noticed. Effect handlers, such as that of Wu (2012) and Kiselyov and Ishii (2015), do produce correct implementations by running the handler for non-determinism before that of state.

Even after we do have a non-deterministic, global-state passing implementation that is a monad, its semantics can sometimes be surprising. In $m_1 \mathbin{[\!]} m_2$, the computation m_2 receives the state computed by m_1. Thus $(\mathbin{[\!]})$ is still associative, but certainly cannot be commutive, nor idempotent. In backtracking algorithms that keeps a global state, it is a common pattern to 1. advance the current state to its next step, 2. recursively search for solutions, and 3. roll back the state to the previous step. To implement a monadic program, one might come up with something like the code below:

$$\mathit{modify}\ \mathit{next} \gg \mathit{solve} \gg\!= \mathit{overwriteBy}\ \mathit{prev} \quad .$$

where *next* advances the state, *prev* undoes the modification of *next*, and *modify* and *overwriteBy* are defined by:

$$
\begin{array}{l}
\mathit{modify}\ f \qquad\quad = \mathit{get} \gg\!= (\mathit{put} \cdot f) \quad , \\
\mathit{overwriteBy}\ f\ v = \mathit{get} \gg\!= (\mathit{put} \cdot f) \gg \mathit{return}\ v \quad .
\end{array}
$$

Let the initial state be *st* and assume that *solve* found three choices $m_1 \mathbin{[\!]} m_2 \mathbin{[\!]} m_3$. The intention is that all of m_1, m_2, and m_3 start running with state $\mathit{next}\ \mathit{st}$, and the state is restored to $\mathit{prev}\ (\mathit{next}\ \mathit{st}) = \mathit{st}$ afterwards. By (8), however,

$$\begin{array}{l} \mathit{modify}\ \mathit{next} \gg (m_1 \mathbin{[\!]} m_2 \mathbin{[\!]} m_3) \gg \mathit{overwriteBy}\ \mathit{prev} = \\ \quad \mathit{modify}\ \mathit{next} \gg ((m_1 \gg \mathit{overwriteBy}\ \mathit{prev}) \mathbin{[\!]} \\ \qquad\qquad\qquad\qquad (m_2 \gg \mathit{overwriteBy}\ \mathit{prev}) \mathbin{[\!]} \\ \qquad\qquad\qquad\qquad (m_3 \gg \mathit{overwriteBy}\ \mathit{prev}))\ \ , \end{array}$$

which is fine in the semantics in Section 5 but, with a shared state, means that m_2 starts with state *st*, after which the state is rolled back too early to *prev st*. The computatin m_3 starts with *prev st*, after which the state is rolled too far to *prev* (*prev st*).

In fact, one cannot guarantee that *overwriteBy prev* is always executed — if *solve* fails, we get $\mathit{modify}\ \mathit{next} \gg \mathit{solve} \ggg\!= \mathit{overwriteBy}\ \mathit{prev} = \mathit{modify}\ \mathit{next} \gg \emptyset \ggg\!= \mathit{overwriteBy}\ \mathit{prev} = \mathit{modify}\ \mathit{next} \gg \emptyset$. Thus the state is advanced to *next st*, but not rolled back to *st*.

We need a way to say that "*modify next* and *modify prev* are run exactly once, respectively before and after all non-deterministic branches in *solve*." For that we should perhaps propose a special operator, with its associated algebraic rules. Fortunately, for the purpose of this pearl, there is a curious hack. Define

$$\begin{array}{l} \mathit{side} :: \mathsf{N} \in \varepsilon \Rightarrow \mathsf{M}_\varepsilon\ a \to \mathsf{M}_\varepsilon\ b \\ \mathit{side}\ m = m \gg \emptyset\ \ . \end{array}$$

Since non-deterministic branches are executed sequentially, the program

$$\mathit{side}\ (\mathit{modify}\ \mathit{next}) \mathbin{[\!]} m_1 \mathbin{[\!]} m_2 \mathbin{[\!]} m_3 \mathbin{[\!]} \mathit{side}\ (\mathit{modify}\ \mathit{prev})$$

executes *modify next* and *modify prev* once, respectively before and after all the non-deterministic branches, even if they fail. Note that *side m* does not generate a result. Its presence is merely for the side-effect of *m*, hence the name. Note also that the type of *side m* need not be the same as that of *m*.

One should note that this does not implement a "barrier" operator that performs an operation after all non-deterministic branches and before all subsequent operations. The program $(m_1 \mathbin{[\!]} m_2 \mathbin{[\!]} \mathit{side}\ \mathit{cmd}) \gg m_3$, for example, is equivalent to $(m_1 \gg m_3) \mathbin{[\!]} (m_2 \gg m_3) \mathbin{[\!]}$ *side cmd* — effects of *cmd* do not happen before m_3. For the purposes in this pearl, however, this hack is sufficient.

As a side note: the pattern $m \gg \emptyset$ is often used in generators, as explained by Kiselyov (2011), who also mentioned using $([\!])$ to simulate chaining, and showed that it does not work in general.

For the rest of the paper we use the following abbreviations:

$$\begin{array}{l} \mathit{sidePut}\ \mathit{st}\ = \mathit{side}\ (\mathit{put}\ \mathit{st})\qquad , \\ \mathit{sideMod}\ f = \mathit{side}\ (\mathit{modify}\ f)\ \ . \end{array}$$

6.1 Passing and Restoring a Global State

Like in Section 5.1, we aim to compute *assert safe* using a stateful *foldr*.

We do not demand (10) to hold in general, but we still needs it to hold for restricted types of monads. For this section, we only needs it to hold for $m :: \mathsf{M}_\mathsf{N}\ a$:

$$m \ggg\!= (\lambda x \to f_1\ x \mathbin{[\!]} f_2\ x)\ =\ (m \ggg\!= f_1) \mathbin{[\!]} (m \ggg\!= f_2)\ \ ,\quad \text{for } m :: \mathsf{M}_\mathsf{N}\ a, \tag{17}$$

$$\mathit{get} \ggg\!= (\lambda x \to f_1\ x \mathbin{[\!]} f_2\ x)\ =\ (\mathit{get} \ggg\!= f_1) \mathbin{[\!]} (\mathit{get} \ggg\!= f_2)\ \ ,\quad \text{for } f :: a \to \mathsf{M}_\mathsf{N}\ b. \tag{18}$$

Equation (19) demands that a stateless computation commutes with *side*, while (20) allows us to move $(\talloblong\, side\ cmd)$ inside a pure application:

$$m \talloblong side\ cmd \;=\; side\ cmd \talloblong m \;\;, \quad \text{for } m :: \mathsf{M_N}\ a. \tag{19}$$
$$(f \langle\$\rangle\, m) \talloblong side\ cmd \;=\; f \langle\$\rangle\, (m \talloblong side\ cmd) \;\;. \tag{20}$$

Property (21) allows us to join commands in adjacent *side*, while (22) introduces a *get*:

$$side\ cmd_1 \talloblong side\ cmd_2 \;=\; side\ (cmd_1 \gg cmd_2) \;\;, \tag{21}$$
$$side\ (put\ st) \talloblong f\ st \;=\; side\ (put\ st) \talloblong (get \gg\!= f) \;\;. \tag{22}$$

One consequence of (21) and (12) is that $side\ (put\ st) \talloblong side\ (put\ st) = side\ (put\ st)$. More generally, $side\ (put\ st)$ can be duplicated an arbitrary number of times and, since it does not return a value, it can be proved by induction on m that:

$$m \gg side\ (put\ st) = side\ (put\ st) \;\;, \quad \text{if } m :: \mathsf{M_N}\ a \text{ and } m \neq \emptyset. \tag{23}$$

With the properties above, one can prove a lemma analog to Theorem 4.1.

Theorem 6.1
Assume the properties stated in this section. Given $(\oplus), (\ominus) :: s \to a \to s$ such that $(st \oplus x) \ominus x = st$ for all x and st, we have that for all $st :: s$ and $xs :: [a]$:

$$\begin{array}{l}
return\ (scanl_+\ (\oplus)\ st\ xs) \talloblong sidePut\ st = \\
\quad sidePut\ st \talloblong foldr\ (\otimes)\ (return\ [\,])\ xs \;\;, \\
\quad \textbf{where}\ x \otimes m = get \gg\!= \lambda st \to sideMod\ (\oplus x) \talloblong \\
\qquad\qquad\qquad\qquad\qquad\qquad (((st \oplus x){:})\ \langle\$\rangle\ m) \talloblong \\
\qquad\qquad\qquad\qquad\qquad\qquad sideMod\ (\ominus x) \;\;.
\end{array}$$

The computation in Theorem 4.1 retrieves the state of the context and restores it back using *get* and *overwrite*. In contrast, the final state after the computation in Theorem 6.1 will be *st*. On the righthand side of the equation, we start with state *st* and, in each step of *foldr*, advances the state by $(\oplus x)$ before the recursive call, and steps back by $(\ominus x)$ afterwards.

We skip the proof of Theorem 6.1 and instead present a direct proof of the following lemma focusing on $assert\ (all\ ok \cdot scanl_+\ (\oplus)\ st)$:

Theorem 6.2
For all $(\oplus), (\ominus) :: s \to a \to s$ such that $(st \oplus x) \ominus x = st$ for all x and st, we have that for all $st :: s$ and $xs :: [a]$:

$$\begin{array}{l}
assert\ (all\ ok \cdot scanl_+\ (\oplus)\ st)\ xs \talloblong sidePut\ st = \\
\quad sidePut\ st \talloblong foldr\ (\odot)\ (return\ [\,])\ xs \;\;, \\
\textbf{where}\ x \odot m = get \gg\!= \lambda st \to guard\ (ok\ (st \oplus x)) \gg \\
\qquad\qquad\quad (sideMod\ (\oplus x) \talloblong ((x{:})\ \langle\$\rangle\ m) \talloblong sideMod\ (\ominus x)) \;\;.
\end{array}$$

Proof
The base case where $xs := [\,]$ needs (19). For the inductive case $xs := x : xs$ we reason:

$$\begin{array}{ll}
& assert\ (all\ ok \cdot scanl_+\ (\oplus)\ st)\ (x : xs) \talloblong sidePut\ st \\
= & \{\ \text{introduce } sidePut\ st \text{ and } get\text{. See below.}\ \}
\end{array}$$

$$
\begin{aligned}
&\quad \mathit{sidePut}\ \mathit{st} \mathbin{[\!]} (\mathit{get} \gg\!= \lambda \mathit{st} \rightarrow \\
&\quad \mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ \mathit{st})\ (x : \mathit{xs}) \mathbin{[\!]} \mathit{sidePut}\ \mathit{st}) \\
&= \quad \{ \text{ definition of } \mathit{assert} \} \\
&\quad \mathit{sidePut}\ \mathit{st} \mathbin{[\!]} (\mathit{get} \gg\!= \lambda \mathit{st} \rightarrow (\mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{guard}\ (\mathit{all}\ \mathit{ok}\ (\mathit{scanl}_+\ (\oplus)\ (\mathit{st} \oplus x)\ \mathit{xs})) \gg \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (x : \mathit{xs})) \mathbin{[\!]} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{sidePut}\ \mathit{st}) \\
&= \quad \{ \text{ (23), (20), and (17), monad laws } \} \\
&\quad \mathit{sidePut}\ \mathit{st} \mathbin{[\!]} (\mathit{get} \gg\!= \lambda \mathit{st} \rightarrow \mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ ((x{:}) \langle\$\rangle (\mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ (\mathit{st} \oplus x))\ \mathit{xs} \mathbin{[\!]} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{sidePut}\ \mathit{st})))\ \ .
\end{aligned}
$$

In the second step we can introduce *sidePut st* and *get* because, for $f\ \mathit{st} :: \mathsf{M}_{\mathsf{N}}\ a$, we have:

$$
\begin{aligned}
&\quad f\ \mathit{st} \mathbin{[\!]} \mathit{sidePut}\ \mathit{st} \\
&= \quad \{ \text{ by (21) and (12) } \} \\
&\quad f\ \mathit{st} \mathbin{[\!]} \mathit{sidePut}\ \mathit{st} \mathbin{[\!]} \mathit{sidePut}\ \mathit{st} \\
&= \quad \{ \text{ by (19) } \} \\
&\quad \mathit{sidePut}\ \mathit{st} \mathbin{[\!]} f\ \mathit{st} \mathbin{[\!]} \mathit{sidePut}\ \mathit{st} \\
&= \quad \{ \text{ by (22) } \} \\
&\quad \mathit{sidePut}\ \mathit{st} \mathbin{[\!]} (\mathit{get} \gg\!= \lambda \mathit{st} \rightarrow f\ \mathit{st} \mathbin{[\!]} \mathit{sidePut}\ \mathit{st})\ \ .
\end{aligned}
$$

We focus on the sub-expression after $\mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg$:

$$
\begin{aligned}
&\quad (x{:}) \langle\$\rangle (\mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ (\mathit{st} \oplus x))\ \mathit{xs} \mathbin{[\!]} \mathit{sidePut}\ \mathit{st}) \\
&= \quad \{ \text{ (21) and that } (\mathit{st} \oplus x) \ominus x = \mathit{st} \} \\
&\quad (x{:}) \langle\$\rangle (\mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ (\mathit{st} \oplus x))\ \mathit{xs} \mathbin{[\!]} \mathit{sidePut}\ (\mathit{st} \oplus x) \mathbin{[\!]} \mathit{sideMod}\ (\ominus x)) \\
&= \quad \{ \text{ induction } \} \\
&\quad (x{:}) \langle\$\rangle (\mathit{sidePut}\ (\mathit{st} \oplus x) \mathbin{[\!]} \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ \mathit{xs} \mathbin{[\!]} \mathit{sideMod}\ (\ominus x)) \\
&= \quad \{ \text{ (20) } \} \\
&\quad \mathit{sidePut}\ (\mathit{st} \oplus x) \mathbin{[\!]} ((x{:}) \langle\$\rangle \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ \mathit{xs}) \mathbin{[\!]} \mathit{sideMod}\ (\ominus x) \\
&= \quad \{ \text{ (17) and (21) } \} \\
&\quad \mathit{sideMod}\ (\oplus x) \mathbin{[\!]} ((x{:}) \langle\$\rangle \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ \mathit{xs}) \mathbin{[\!]} \mathit{sideMod}\ (\ominus x)\ \ .
\end{aligned}
$$

Back to the main calculation, we have:

$$
\begin{aligned}
&\quad \mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ \mathit{st})\ (x : \mathit{xs}) \mathbin{[\!]} \mathit{sidePut}\ \mathit{st} \\
&= \qquad \{ \text{ calculations above } \} \\
&\quad \mathit{sidePut}\ \mathit{st} \mathbin{[\!]} (\mathit{get} \gg\!= \lambda \mathit{st} \rightarrow \mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg \\
&\qquad \mathit{sideMod}\ (\oplus x) \mathbin{[\!]} ((x{:}) \langle\$\rangle \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ \mathit{xs}) \mathbin{[\!]} \mathit{sideMod}\ (\ominus x)) \\
&= \mathit{sidePut}\ \mathit{st} \mathbin{[\!]} \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ (x : \mathit{xs})\ \ .
\end{aligned}
$$

$\square$

6.2 *n* Queens with a Global State

Recall that $\mathit{queens}\ n = \mathit{perm}\ [0\,..\,n-1] \gg\!= \mathit{assert}\ (\mathit{safeAcc}\ (0, [\,], [\,]))$, where $\mathit{safeAcc}\ \mathit{st} = \mathit{map}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ \mathit{st}\ (\mathit{return}\ [\,])$ and $(i, \mathit{us}, \mathit{ds}) \oplus x = (i+1, (i+x) : \mathit{us}, (i-x) : \mathit{ds})$. The reverse operation of $(\oplus)$ is:

$$(i, us, ds) \ominus x = (i-1, \mathit{tail}\; us, \mathit{tail}\; ds) \;\;.$$

To apply the result in Section 6.1, we define $\mathit{runQueens}\; n = \mathit{queens}\; n \mathbin{[\!]} \mathit{sidePut}\; (0, [\,], [\,])$. In the calculation below we abbreviate $(0, [\,], [\,])$ to *ini*, and assume that $n > 0$:

$$\begin{array}{ll}
& \mathit{queens}\; n \mathbin{[\!]} \mathit{sidePut}\; \mathit{ini} \\
= & (\mathit{perm}\; [0\mathinner{..}n-1] \gg\!= \mathit{assert}\; \mathit{safe}) \mathbin{[\!]} \mathit{sidePut}\; \mathit{ini} \\
= & \quad \{ \text{ since } \mathit{perm}\; [0\mathinner{..}n-1] :: \mathsf{M_N}\; [\mathsf{Int}], \text{ by (23) and (17) } \} \\
& \mathit{perm}\; [0\mathinner{..}n-1] \gg\!= (\lambda xs \rightarrow \mathit{assert}\; \mathit{safe}\; xs \mathbin{[\!]} \mathit{sidePut}\; \mathit{ini}) \\
= & \quad \{ \text{ discussion in Section 6.1 } \} \\
& \mathit{perm}\; [0\mathinner{..}n-1] \gg\!= (\lambda xs \rightarrow \mathit{sidePut}\; \mathit{ini} \mathbin{[\!]} \mathit{foldr}\; (\odot)\; (\mathit{return}\; [\,])\; xs) \\
= & \quad \{ \text{ by (23) and (17) } \} \\
& \mathit{sidePut}\; \mathit{ini} \mathbin{[\!]} (\mathit{perm}\; [0\mathinner{..}n-1] \gg\!= \mathit{foldr}\; (\odot)\; (\mathit{return}\; [\,])) \;\;.
\end{array}$$

Define $\mathit{queensBody}\; xs = \mathit{perm}\; xs \gg\!= \mathit{foldr}\; (\odot)\; (\mathit{return}\; [\,])$. The aim now is to fuse it into a hylomorphism. By Theorem 4.2, we need to show that

$$\begin{array}{ll}
& \mathit{get} \gg\!= \lambda s \rightarrow \mathit{guard}\; (\mathit{ok}\; (s \oplus x)) \gg \\
& ((x{:}) \mathbin{\langle\$\rangle} (\mathit{sideMod}\; (\oplus x) \mathbin{[\!]} m \mathbin{[\!]} \mathit{sideMod}\; (\ominus x))) \\
= & m \gg\!= \lambda v \rightarrow \mathit{get} \gg\!= \lambda s \rightarrow \mathit{guard}\; (\mathit{ok}\; (s \oplus x)) \gg \\
& (x{:}) \mathbin{\langle\$\rangle} (\mathit{sideMod}\; (\oplus x) \mathbin{[\!]} \mathit{return}\; v \mathbin{[\!]} \mathit{sideMod}\; (\ominus x)) \;\;.
\end{array}$$

To prove the property, the main lemma we need is:

Lemma 6.3
With $m :: \mathsf{M_N}\; a$ and $(st \oplus x) \ominus x = st$, we have

$$\begin{array}{ll}
& m \gg\!= \lambda v \rightarrow \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} \mathit{return}\; v \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \\
= & \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} m \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \;\;.
\end{array}$$

Proof
Induction on structure of m. We look at the case for $m := m_1 \mathbin{[\!]} m_2$:

$$\begin{array}{ll}
& m_1 \mathbin{[\!]} m_2 \gg\!= \lambda v \rightarrow \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} \mathit{return}\; v \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \\
= & \quad \{ \text{ by (8) } \} \\
& (m_1 \gg\!= \lambda v \rightarrow \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} \mathit{return}\; v \mathbin{[\!]} \mathit{sideMod}\; (\ominus x)) \mathbin{[\!]} \\
& (m_2 \gg\!= \lambda v \rightarrow \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} \mathit{return}\; v \mathbin{[\!]} \mathit{sideMod}\; (\ominus x)) \\
= & \quad \{ \text{ induction } \} \\
& \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} m_1 \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \mathbin{[\!]} \\
& \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} m_2 \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \\
= & \quad \{ \text{ by (19) } \} \\
& m_1 \mathbin{[\!]} \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \mathbin{[\!]} \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} m_2 \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \\
= & \quad \{ \text{ by (21) and that } (st \oplus x) \ominus x = st \; \} \\
& m_1 \mathbin{[\!]} \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} m_2 \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \\
= & \quad \{ \text{ by (19) } \} \\
& \mathit{sideMod}\; (\oplus x) \mathbin{[\!]} m_1 \mathbin{[\!]} m_2 \mathbin{[\!]} \mathit{sideMod}\; (\ominus x) \;\;.
\end{array}$$

$\square$

We may conclude that $\mathit{runQueens}\; n = \mathit{sidePut}\; (0, [\,], [\,]) \mathbin{[\!]} \mathit{queensBody}\; [0\mathinner{..}n-1]$, and

$$\begin{array}{l}
\mathit{queensBody}\;[\,] = \mathit{return}\;[\,]\\
\mathit{queensBody}\;\mathit{xs} = \mathit{select}\;\mathit{xs} \gg\!= \lambda\,(x,\mathit{ys}) \to\\
\qquad\qquad\qquad \mathit{get} \gg\!= \lambda \mathit{st} \to \mathit{guard}\;(\mathit{ok}\;(\mathit{st} \oplus x)) \gg\\
\qquad\qquad\qquad ((x{:})\;\langle\$\rangle\;(\mathit{sideMod}\;(\oplus x)\;[\!]\;\mathit{queensBody}\;\mathit{ys}\;[\!]\;\mathit{sideMod}\;(\ominus x)))\;\;.
\end{array}$$

6.3 Bonus: A Sudoku Solver

To demonstrate an application where an array is used in the state, we implemented a backtracking, brute-force Sudoku solver. For those who have not yet heard of the puzzle: given is a 9 by 9 grid, some cells being blank and some filled with numbers. The aim is to fill in the blank cells such that each column, each row, and each of the nine 3 by 3 sub-grids (also called “boxes”) contains all of the digits from 1 to 9.

In the specification, we simply try, for each blank cell, each of the 9 digits. Define:

$$\begin{array}{l}
\mathit{allChoices} :: \mathsf{N} \in \varepsilon \Rightarrow \mathsf{Int} \to \mathsf{M}_\varepsilon\;[\mathsf{Int}]\\
\mathit{allChoices} = \mathit{unfoldM}\;(0\,{==})\;(\lambda n \to \mathit{liftList}\;[1\,..\,9] \gg\!= \lambda x \to \mathit{return}\;(x, n-1))\;\;,
\end{array}$$

where $\mathit{liftList} :: \mathsf{N} \in \varepsilon \Rightarrow [a] \Rightarrow \mathsf{M}_\varepsilon\;a$ non-deterministically returns an element in the given list, such that $\mathit{allChoices}\;n$ returns a list of length n whose elements are independently chosen from $[1\,..\,9]$. If a given grid contains n blank cells, a list of length n represents a proposed solution.

To check whether a solution is valid, we inspect the list left-to-right, keeping a state. It is sufficient letting the state be the current partially filled grid. For convenience, and also for simplifying the definition of $(\oplus)$ and $(\ominus)$, we also maintain a zipper of positions:

$$\begin{array}{ll}
\mathbf{type}\;\mathsf{Pos} = (\mathsf{Int},\mathsf{Int})\;\;, & \mathbf{type}\;\mathsf{Grid} = \mathsf{Array}\;\mathsf{Pos}\;\mathsf{Entry}\;\;,\\
\mathbf{type}\;\mathsf{Entry} = \mathsf{Int}\;\;, & \mathbf{type}\;\mathsf{State} = (\mathsf{Grid},[\mathsf{Pos}],[\mathsf{Pos}])\;\;.
\end{array}$$

Therefore, in the state $(\mathit{grid},\mathit{todo},\mathit{done})$, grid is an array representing the current status of the grid (an empty cell being filled 0), todo is a list of blank positions to be filled, and done is the list of positions that were blank but are now filled.

We assume a function $\mathit{collisions} :: \mathsf{Pos} \to [\mathsf{Pos}]$ which, given a position, returns a list of positions that should be checked. That is, $\mathit{collisions}\;i$ returns all the positions that are on the same row, column, and in the same box as i. With that, we may define:

$$\begin{array}{l}
\mathit{safeAcc}\;\mathit{st} = \mathit{all}\;\mathit{ok} \cdot \mathit{scanl}_{+}\;(\oplus)\;\mathit{st}\;\;,\\
\quad \mathbf{where}\;(\mathit{grid}, i : \mathit{is}, \mathit{js}) \oplus x = (\mathit{grid} \mathbin{/\!/} [(i,x)], \mathit{is}, i : \mathit{js})\;\;,\\
\qquad\qquad \mathit{ok}\;(\mathit{grid}, \mathit{is}, i : \mathit{js}) = \mathit{all}\;((\mathit{grid}\,!\,i) \neq)\;(\mathit{map}\;(\mathit{grid}!)\;(\mathit{collisions}\;i))\;\;,
\end{array}$$

where $(!)$ and $(/\!/)$ respectively performs array reading and updating. The reverse operation of $(\oplus)$ is $(\mathit{grid},\mathit{is}, i : \mathit{js}) \ominus x = (\mathit{grid} \mathbin{/\!/} [(i,0)], i : \mathit{is}, \mathit{js})$. A program solving the puzzle can be specified by

$$\begin{array}{l}
\mathit{sudoku}\;\mathit{grid} = (\mathit{assert}\;(\mathit{safeAcc}\;\mathit{initState}) =\!\ll \mathit{allChoices}\;(\mathit{length}\;\mathit{empties}))\;[\!]\\
\qquad\qquad\qquad \mathit{sidePut}\;\mathit{initState}\;\;,\\
\quad \mathbf{where}\;\mathit{initState} = (\mathit{grid}, \mathit{empties}, [\,])\;\;,
\end{array}$$

where grid is the array representing the initial puzzle and $\mathit{empties}$ are the blank positions.

Through a similar derivation, we come up with a backtracking Sudoku solver, whose main computation takes the number of blank cells and returns a list of digits to fill (in the order specified by *empties* in the state). The actual implementation is slightly complicated by conversion from Array to STUArray, the array that supports in-place update in Haskell. In the code below, *guardNoCollision* is specified by $get \gg\!= (guard \cdot ok \cdot (\oplus x))$. Operations *modNext x* and *modPrev x* are respectively specified by *modify* $(\oplus x)$ and *modify* $(\ominus x)$, but alters the array directly.

$$
\begin{aligned}
&solve\ 0 = return\ [\,]\\
&solve\ n = liftList\ [1\,..\,9] \gg\!= \lambda x \to guardNoCollision\ x \gg\\
&\qquad\qquad (x{:})\ \langle\$\rangle\ (side\ (modNext\ x) \mathbin{[\!]} solve\ (n-1) \mathbin{[\!]} side\ (modPrev\ x))\ \ .
\end{aligned}
$$

The handle of an STUArray is usually stored in a reader monad. The actual monad we use is constructed by

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Grid}\ s \quad\ = \mathsf{STUArray}\ s\ \mathsf{Pos}\ \mathsf{Entry}\ \ ,\\
&\textbf{type}\ \mathsf{SudoM}\ s = \mathsf{ListT}\ (\mathsf{StateT}\ ([\mathsf{Pos}],[\mathsf{Pos}])\ (\mathsf{ReaderT}\ (\mathsf{Grid}\ s)\ (\mathsf{ST}\ s)))\ \ ,
\end{aligned}
$$

where ListT is one of the correct implementations (Gale, 2007).

This Sudoku solver is not very effective — a puzzle rated as “hard” could take minutes to solve. For Sudoku, naive brute-force searching does not make a good algorithm. In comparison, Bird (2010) derived a purely functional program, based on constraint refining, that is able to solve the same puzzle in an instant. Nevertheless, the algorithm in this section does the job, and demonstrates that our pattern of derivation is applicable.

7 Conclusion

We have demonstrated several case studies of reasoning and derivation of monadic programs. Using the monad laws and the laws of non-determinism operators, we derived conditions under which a monadic program modelling Spark aggregation is deterministic. The proofs may proceed in the usual algebraic, equational style. To study the interaction between non-determinism and state, we derived two backtracking algorithms from specification. In both cases, the state monad is introduced by converting *assert p*, where *p* uses $scanl_{+}$, to a *foldr* that uses monadic state operations. We present general theorems for such conversion, as well as and theorems that allows fusion of hylomorphism. This pattern is applied to solve the *n*-queens and the Sudoku puzzle. Theorem 4.1, 6.1, etc., can be generalised to specific forms of *foldr* rather than $scanl_{+}$. We believe that it is a general pattern to convert an explicitly-state-passing function to one that uses a state monad.

It turns that in derivations of programs using non-determinism and state, commutativity plays a central role. With the state being local, we have more properties at hand, and commutativity holds more generally. With a shared global state, commutativity holds in limited cases, making the derivation more difficult. There are practical situations when we need the state to be global, and we may need more strategies for deriving such programs.

We discovered a hack that simulates chaining of effects using $([\!])$, which works for at least this class of backtracking algorithms. In general, we may need a proper operator that acts as barrier among stateful, nondeterministic operations.

The theorems in this pearl are restricted to unfold and fold for lists. To generalise them to F-algebras, more abstraction are needed to thread the effects in the functor F. Nevertheless it is an interesting topic to explore. We are also hoping to seeing more derivations of monadic programs, especially those mixing several effects.

References

Bird, Richard S. (2010). *Pearls of functional algorithm design*. Cambridge University Press.

Chen, Yu-Fang, Hong, Chih-Duo, Lengál, Ondřej, Mu, Shin-Cheng, Sinha, Nishant, & Wang, Bow-Yaw. (2017). An executable sequential specification for Spark aggregation. *International conference on networked systems*. Springer-Verlag.

Doornbos, Henk, & Backhouse, Roland Carl. (1995). Induction and recursion on datatypes. *Pages 242–256 of:* Möller, Bernhard (ed), *Mathematics of program construction, 3rd international conference*. LNCS, no. 947. Springer-Verlag.

Fischer, Sebastian, Kiselyov, Oleg, & Shan, Chung-chieh. (2011). Purely functional lazy nondeterministic programming. *Journal of functional programming*, **21**(4-5), 413–465.

Gale, Yitzchak. (2007). *ListT done right alternative*. `https://wiki.haskell.org/ListT_done_right_alternative`.

Gibbons, Jeremy, & Hinze, Ralf. (2011). Just do it: simple monadic equational reasoning. *Pages 2–14 of:* Danvy, Olivier (ed), *International conference on functional programming*. ACM Press.

Gill, Andy, & Kmett, Edward. (2014). *The monad transformer library*. `https://hackage.haskell.org/package/mtl`.

Hinze, Ralf, Wu, Nicolas, & Gibbons, Jeremy. (2015). Conjugate hylomorphisms, or: the mother of all structured recursion schemes. *Pages 527–538 of:* Walker, David (ed), *Symposium on principles of programming languages*. ACM Press.

Hutton, Graham, & Fulger, Diana. (2007). Reasoning about effects: seeing the wood through the trees. *Symposium on trends in functional programming*.

Kiselyov, Oleg. (2011). *Generators: the API for traversal and non-determinism*. `http://okmij.org/ftp/continuations/generators.html`.

Kiselyov, Oleg. (2015). *Laws of monadplus*. `http://okmij.org/ftp/Computation/monads.html\#monadplus`.

Kiselyov, Oleg, & Ishii, Hiromi. (2015). Freer monads, more extensible effects. *Pages 94–105 of:* Reppy, John H (ed), *Symposium on haskell*. ACM Press.

Kiselyov, Oleg, Sabry, Amr, & Swords, Cameron. (2013). Extensible effects: an alternative to monad transformers. *Pages 59–70 of:* Shan, Chung-chieh (ed), *Symposium on Haskell*. ACM Press.

Liang, Sheng, Hudak, Paul, & Jones, Mark. (1995). Monad transformers and modular interpreters. *Pages 333–343 of:* Cytron, Ron K., & Lee, Peter (eds), *Symposium on principles of programming languages*. ACM Press.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional programming with bananas, lenses, envelopes, and barbed wire. *Pages 124–144 of:* Hughes, R. John Muir (ed), *Functional programming languages and computer architecture*. LNCS, no. 523. Springer-Verlag.

Moggi, Eugenio. (1989). Computational lambda-calculus and monads. *Pages 14–23 of:* Parikh, Rohit (ed), *Logic in computer science*. IEEE Computer Society Press.

Voigtländer, Janis. (2009). Free theorems involving type constructor classes. *Pages 173–184 of:* Tolmach, Andrew (ed), *International conference on functional programming*. ACM Press.

Volkov, Nikita. (2014). *The list-t package*. `http://hackage.haskell.org/package/list-t`.

Wadler, Philip. (1992). Monads for functional programming. *Pages 233–264 of:* Broy, Manfred (ed), *Program design calculi: Marktoberdorf summer school*. Springer-Verlag.

Wu, Nicolas, Schrijvers, Tom, & Hinze, Ralf. (2012). Effect handlers in scope. *Pages 1–12 of:* Voigtländer, Janis (ed), *Symposium on haskell*. ACM Press.

Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J., Shenker, Scott, & Stoica, Ion. (2012). Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. Gribble, Steven, & Katabi, Dina (eds), *Networked systems design and implementation*. USENIX.