Abstract. Equational reasoning is among the most important tools that functional programming provides us. Curiously, relatively less 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 the intricate interaction between state and non-determinism. We derive a backtracking algorithm for the n-queens puzzle when each non-deterministic branch has its own local state. For the scenario where a global state is shared, we propose laws the monad should satisfy, and develop programming patterns and techniques to simulate local states.

Reasoning and Derivation of Monadic Programs

A Case Study of Non-determinism and State

Shin-Cheng Mu[1], Tom Schrijvers[2], and Koen Pauwels[2]

[1] Institute of Information Science, Academia Sinica
[2] Department of Computer Science, KU Leuven

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 properties 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 — in a rigorous, mathematically manageable manner. Since the introduction of *monads* into the functional programming community [?,?], 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.

Hutton and Fulger [?] noted that relatively less attention has been paid to reasoning about monadic programs. We believe that the observation is still true today, 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. The validity of these properties, proved using only these laws, is independent from the particular implementation of the monad.

This paper follows the trail of Hutton and Fulger [?] and Gibbons and Hinze [?], aiming to develop theorems and patterns that are useful for reasoning about monadic programs. We focus on two effects — non-determinism and state. The interaction between non-determinism and state is known to be intricate. When each non-deterministic branch has its own local state, we get a relatively well-behaved monad, providing a richer collection of properties to work with. When all the non-deterministic branches share one global state, the properties of the monad is much less intuitive, as we shall see in this paper.

In this paper we consider problem specifications that use a monadic unfold to generate possible solutions, which are filtered using a *scanl*-like predicate. We construct backtracking algorithms for such problems in two scenarios, in which the state is respectively local and global. In the local-state case, we develop theorems that convert a variation of *scanl* to a *foldr* that uses the state monad, as well as theorems constructing hylomorphism. For the case of global state, we study programming patterns that guarantee to restore the initial state after all non-deterministic branches, propose laws the global state monad should satisfy, and show that one may simulate local states using a global state. The algorithms are used to solve the n-queens puzzle, our running example.

2 Monad and Effect Operators

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

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

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

$$(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g) \quad . \tag{3}$$

We also define $m_1 \gg m_2 = m_1 \gg\!= \mathit{const}\ m_2$, which has type $(\gg) :: m\ a \rightarrow m\ b \rightarrow m\ b$. Kleisli composition, denoted by $(>\!\!=\!\!>)$, composes two monadic operations $a \rightarrow \mathsf{M}\ b$ and $b \rightarrow \mathsf{M}\ c$ into an operation $a \rightarrow \mathsf{M}\ c$. The operator $(\langle\$\rangle)$ applies a pure function to a monad.

$$
\begin{array}{l}
(>\!\!=\!\!>) :: (a \rightarrow \mathsf{M}\ b) \rightarrow (b \rightarrow \mathsf{M}\ c) \rightarrow a \rightarrow \mathsf{M}\ c \\
(f >\!\!=\!\!> g)\ x = f\ x \gg\!= g \quad , \\
(\langle\$\rangle) :: (a \rightarrow b) \rightarrow \mathsf{M}\ a \rightarrow \mathsf{M}\ b \\
f \mathbin{\langle\$\rangle} m = m \gg\!= (\mathit{return} \cdot f) \quad .
\end{array}
$$

The following properties can be proved from their definitions and the monad laws:

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

$$(f \mathbin{\langle\$\rangle} m) \gg\!= g = m \gg\!= (g \cdot f) \quad , \tag{5}$$

$$f \mathbin{\langle\$\rangle} (m \gg\!= k) = m \gg\!= (\lambda x \rightarrow f \mathbin{\langle\$\rangle} k\ x) \quad , x \text{ not free in } f. \tag{6}$$

Effect and Effect Operators Monads are used to model effects, and each effect comes with its collection of operators. For example, to model non-determinism we assume two operators $\emptyset$ and $(\|)$ (*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 may involve more than one effect. In Haskell, the type class constraint MonadPlus in the type of a program denotes that the program may use $\emptyset$ or $(\|)$, and possibly other effects, while MonadState s denotes that it may use *get* and *put*. Some theorems in this paper, however, apply only to programs that, for example, use non-determinism and no other effects. To talk about such programs, we use a slightly different notation. We let the type $\mathsf{M}_\epsilon\ a$ denote a monad whose return type is a and, during whose execution, effects in the set ϵ may occur. This paper considers only two effects: non-determinism and state. Non-determinism is denoted by N, for which we assume two operators $\emptyset :: \mathsf{M}_\epsilon\ a$ and $(\talloblong) :: \mathsf{M}_\epsilon\ a \to \mathsf{M}_\epsilon\ a \to \mathsf{M}_\epsilon\ a$, where $\mathsf{N} \in \epsilon$. A state effect is denoted by $\mathsf{S}\ s$, where s is the type of the state, with two operators $get :: \mathsf{M}_\epsilon\ s$ and $put :: s \to \mathsf{M}_\epsilon\ ()$ where $\mathsf{S}\ s \in \epsilon$.

We introduce a small language of effectful programs and a simple type system in Figure 1. If a program has type $\mathsf{M}_\mathsf{N}\ a$, we know that non-determinism is the *only effect* allowed. Inference of effects is not unique, however: a program using $\emptyset$ can be typed as both $\mathsf{M}_\mathsf{N}\ a$ and $\mathsf{M}_{\{\mathsf{N},\mathsf{S}\ \mathsf{Int}\}}\ a$. We sometimes denote the constraint on ϵ in a type-class-like syntax, e.g $\emptyset :: \mathsf{N} \in \epsilon \Rightarrow \mathsf{M}_\epsilon\ a$. All these are merely notational convenience — the point is that the effects a program uses can be determined statically.

$$\begin{array}{lll}
\mathcal{E} &= \text{pure, non-monadic expressons} & \\
\mathcal{F} &= \text{functions returning monadic programs} & \\
\mathcal{P} &= return\ \mathcal{E} \mid \mathcal{P} \gg\!= \mathcal{F} \mid \emptyset \mid \mathcal{P} \talloblong \mathcal{P} \mid get \mid put\ \mathcal{E} & \text{— monadic progs} \\
\mathcal{T} &= a \mid c \mid \mathcal{T} \to \mathcal{T} \mid \mathsf{M}_{\{\mathcal{F}\}}\ \mathcal{T} & \text{— types} \\
\mathcal{F} &= \mathsf{S}\ a \mid \mathsf{S}\ c \mid \mathsf{N} & \text{— effects}
\end{array}$$

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

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

$$\frac{\mathsf{N} \in \epsilon}{\Gamma \vdash \emptyset :: \mathsf{M}_\epsilon\ a} \qquad \frac{\Gamma \vdash m_1 :: \mathsf{M}_\epsilon\ a \quad \Gamma \vdash m_2 :: \mathsf{M}_\epsilon\ a \quad \mathsf{N} \in \epsilon}{\Gamma \vdash m_1 \talloblong m_2 :: \mathsf{M}_\epsilon\ a}$$

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

Fig. 1: A small language and type system for effectful programs.

Total, Finite Programs Like in other literature on program derivation, we assume a set-theoretic semantics in which functions are total. Lists in this paper 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 $\mathsf{M}_\epsilon\ a$ by structural induction over its syntax.

3 Example: The n-Queens Problem

Reasoning about monadic programs gets more interesting when more than one effect is involved. Backtracking algorithms make good examples of programs that are stateful and non-deterministic, and the n-queens problem, also dealt with by Gibbons and Hinze [**?**], is among the most well-known examples of backtracking.[3]

In this section we present a specification of the problem, before transforming it into the form $\mathit{unfoldM}\ p\ f \ggg \mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ \mathit{st})$ (whose components will be defined later), which is the general form of problems we will deal with in this paper. The specification is non-deterministic, but not stateful. In the next few sections we will introduce state into the specification, under different assumptions of the interaction between non-determinism and state.

3.1 Non-Determinism

Since the n-queens problem will be specified by a non-deterministic program, we discuss non-determinism before presenting the specification. 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. What laws they should satisfy, however, can be a tricky issue. As discussed by Kiselyov [**?**], it eventually comes down to what we use the monad for. It is usually expected that $(\talloblong)$ and $\emptyset$ form a monoid. That is, $(\talloblong)$ is associative, with $\emptyset$ as its zero:

$$(m \talloblong n) \talloblong k \;=\; m \talloblong (n \talloblong k)\quad , \tag{7}$$

$$\emptyset \talloblong m \;=\; m \;=\; m \talloblong \emptyset\quad . \tag{8}$$

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

$$\textbf{left-distributivity}:\quad (m_1 \talloblong m_2) \gg\!= f \;=\; (m_1 \gg\!= f) \talloblong (m_2 \gg\!= f)\quad , \tag{9}$$

$$\textbf{left-zero}:\quad \emptyset \gg\!= f \;=\; \emptyset\quad . \tag{10}$$

We will refer to the laws (7), (8), (9), (10) collectively as the *nondeterminism laws*. Other properties regarding $\emptyset$ and $(\talloblong)$ will be introduced when needed.

3.2 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 $[0\mathinner{..}n-1]$. 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 $[0\mathinner{..}n-1]$, such that $\mathit{xs}\ !!\ i = j$ denotes that the queen on the ith column is

[3] Curiously, Gibbons and Hinze [**?**] 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.

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

placed on the jth row (see Figure 2(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 a non-deterministic program:

$$\begin{array}{l} \mathit{queens} :: \mathsf{N} \in \epsilon \Rightarrow \mathsf{Int} \to \mathsf{M}_\epsilon\ [\mathsf{Int}] \\ \mathit{queens}\ n = \mathit{perm}\ [0\,..\,n-1] \mathbin{>\!\!\!>\!\!=} \mathit{filt}\ \mathit{safe} \quad , \end{array}$$

where *perm* non-deterministically computes a permutation of its input, and the pure function $\mathit{safe} :: [\mathsf{Int}] \to \mathsf{Bool}$ determines whether no queens are on the same diagonal. The monadic function $\mathit{filt}\ p\ x$ returns x if $p\ x$ holds, and fails otherwise:

$$\begin{array}{l} \mathit{filt} :: \mathsf{N} \in \epsilon \Rightarrow (a \to \mathsf{Bool}) \to a \to \mathsf{M}_\epsilon\ a \\ \mathit{filt}\ p\ x = \mathit{guard}\ (p\ x) \gg \mathit{return}\ x \quad , \end{array}$$

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

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

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 define *perm* in terms of an unfold, transform *filt safe* into a fold, and fuse the two phases into a *hylomorphism* [**?**]. During the fusion, some non-safe choices can be pruned off earlier, speeding up the computation.

Permutation The monadic function *perm* can be written both as a fold or an unfold. For this problem we choose the latter. The function *select* non-deterministically splits a list into a pair containing one chosen element and the rest:

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

where $(f \times g)\ (x, y) = (f\ \ x, g\ \ y)$. 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:

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

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

$$
\begin{array}{l}
\mathit{perm} :: \mathsf{N} \in \epsilon \Rightarrow [a] \to \mathsf{M}_\epsilon\ [a] \\
\mathit{perm} = \mathit{unfoldM}\ \ \mathit{null}\ \ \mathit{select} \quad .
\end{array}
$$

3.3 Safety Check in a *scanl*

We have yet to defined *safe*. Representing a placement as a permutation allows an easy way to check whether two queens are put on the same diagonal. An 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 2(b)). If we apply *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 *zipWith* $(-)$ $[0\,..]$ to a permutation, we get the indices of their down-diagonals (indexed by $[-7\,..\,7]$. See Figure 2(c)). A placement is safe if the diagonals contain no duplicates:

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

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

The eventual goal is to transform *filt safe* into a *foldr*, to be fused with *perm*, an unfold that generates a list from left to right. In order to do so, it helps if *safe* can be expressed in a computation that processes the list left-to-right, that is, a *foldl* or a *scanl*. To derive such a definition we use the standard trick — introducing accumulating parameters, and generalising *safe* to *safeAcc* below:

$$
\begin{array}{l}
\mathit{safeAcc} :: (\mathsf{Int}, [\mathsf{Int}], [\mathsf{Int}]) \to [\mathsf{Int}] \to \mathsf{Bool} \\
\mathit{safeAcc}\ (i, \mathit{us}, \mathit{ds})\ \mathit{xs} = \mathit{nodup}\ \mathit{us}' \wedge \mathit{nodup}\ \mathit{ds}' \wedge
\end{array}
$$

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

It is a generalisation because $safe = safeAcc\ (0, [\,], [\,])$. By plain functional calculation, one may conclude that *safeAcc* can be defined using a variation of *scanl*:

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

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

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

Operationally, *safeAcc* examines the list from left to right, while keeping a state (i, us, 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 $scanl_{+}\ (\oplus)\ st$, the value st can be seen as a "state" that is explicitly carried around. This naturally leads to the idea: can we convert a $scanl_{+}$ to a monadic program that stores st in its state? This is the goal of the next section.

As a summary of this section, after defining *queens*, we have transformed it into the following form:

$$unfoldM\ p\ f \ggg filt\ (all\ ok \cdot scanl_{+}\ (\oplus)\ st) \quad .$$

This is the form of problems we will consider for the rest of this paper: problems whose solutions are generated by an monadic unfold, before being filtered by an *filt* that takes the result of a $scanl_{+}$.

4 From Pure to Stateful *scanl*

The aim of this section is to turn the filtering phase $filt\ (all\ ok \cdot scanl_{+}\ (\oplus)\ st)$ into a *foldr*. For that we introduce a state monad to pass the state around.

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

$$\begin{aligned}&\mathbf{put\text{-}put}: && put\ st \gg put\ st' = put\ st' \quad , && (11)\\ &\mathbf{put\text{-}get}: && put\ st \gg get = put\ st \gg return\ st \quad , && (12)\\ &\mathbf{get\text{-}put}: && get \ggg put = return\ () \quad , && (13)\\ &\mathbf{get\text{-}get}: && get \ggg (\lambda st \to get \ggg k\ st) = get \ggg (\lambda st \to k\ st\ st) \quad . && (14)\end{aligned}$$

4.1 From $scanl_+$ to monadic *foldr*

Consider the following monadic variation of *scanl*:

$$\begin{aligned}
&\mathit{scanlM} :: \mathsf{S}\ s \in \epsilon \Rightarrow (s \to a \to s) \to s \to [a] \to \mathsf{M}_\epsilon\ [s] \\
&\mathit{scanlM}\ (\oplus)\ st\ xs = put\ st \gg foldr\ (\otimes)\ (return\ [\,])\ xs \\
&\quad \mathbf{where}\ x \otimes m = get \ggg\!= \lambda st \to \mathbf{let}\ st' = st \oplus x \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (st'{:}) \mathbin{\langle\$\rangle} (put\ st' \gg m) \quad ,
\end{aligned}$$

It behaves like $scanl_+$, but stores the accumulated information in a monadic state, which is retrieved and stored in each step. The main body of the computation is implemented using a *foldr*.

To relate $scanl_+$ and *scanlM*, one would like to have $return\ (scanl_+\ (\oplus)\ st\ xs) = scanlM\ (\oplus)\ st\ xs$. However, the lefthand side does not alter the state, while the righthand side does. One of the ways to make the equality hold is to manually backup and restore the state. Define

$$protect\ m \;=\; get \ggg\!= \lambda ini \to m \ggg\!= \lambda x \to put\ ini \gg return\ x \quad ,$$

We have

Theorem 1. *For all* $(\oplus) :: (s \to a \to s)$*,* $st :: s$*, and* $xs :: [a]$*,*

$$return\ (scanl_+\ (\oplus)\ st\ xs) = protect\ (scanlM\ (\oplus)\ st\ xs) \quad .$$

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

$$\begin{aligned}
&\quad protect\ (scanlM\ (\oplus)\ st\ (x : xs)) \\
&= \quad \{\ \text{expanding definitions, let } st' = st \oplus x\ \} \\
&\quad get \ggg\!= \lambda ini \to put\ st \gg get \ggg\!= \lambda st \to \\
&\quad ((st'{:}) \mathbin{\langle\$\rangle} (put\ st' \gg foldr\ (\otimes)\ (return\ [\,])\ xs)) \ggg\!= \lambda r \to \\
&\quad put\ ini \gg return\ r \\
&= \quad \{\ \text{by } put\text{-}get\ (12)\ \} \\
&\quad get \ggg\!= \lambda ini \to put\ st \gg \\
&\quad ((st'{:}) \mathbin{\langle\$\rangle} (put\ st' \gg foldr\ (\otimes)\ (return\ [\,])\ xs)) \ggg\!= \lambda r \to \\
&\quad put\ ini \gg return\ r \\
&= \quad \{\ \text{by } (6)\ \} \\
&\quad (st'{:}) \mathbin{\langle\$\rangle} (get \ggg\!= \lambda ini \to put\ st \gg put\ st' \gg \\
&\qquad\qquad foldr\ (\otimes)\ (return\ [\,])\ xs \ggg\!= \lambda r \to \\
&\qquad\qquad put\ ini \gg return\ r) \\
&= \quad \{\ \text{by } put\text{-}put\ (11)\ \} \\
&\quad (st'{:}) \mathbin{\langle\$\rangle} (get \ggg\!= \lambda ini \to put\ st' \gg foldr\ (\otimes)\ (return\ [\,])\ xs \\
&\qquad\qquad \ggg\!= \lambda r \to put\ ini \gg return\ r) \\
&= \quad \{\ \text{definitions of } scanlM \text{ and } protect\ \} \\
&\quad (st'{:}) \mathbin{\langle\$\rangle} protect\ (scanlM\ (\oplus)\ st'\ xs) \\
&= \quad \{\ \text{induction}\ \} \\
&\quad (st'{:}) \mathbin{\langle\$\rangle} return\ (scanl_+\ (\oplus)\ st'\ xs) \\
&= return\ ((st \oplus x) : scanl_+\ (\oplus)\ (st \oplus x)\ xs) \\
&= return\ (scanl_+\ (\oplus)\ st\ (x : xs)) \quad .
\end{aligned}$$

This proof is instructive due to the use of properties (11) and (12), and that $(st'{:})$, being a pure function, can be easily moved around using (6).

We have learned that $\mathit{scanl}_{+}\ (\oplus)\ st$ can be turned into $\mathit{scanlM}\ (\oplus)\ st$, defined in terms of a stateful *foldr*. In the definition, state is the only effect involved. The next task is to transform $\mathit{filt}\ (\mathit{scanl}_{+}\ (\oplus)\ st)$ into a *foldr*. The operator *filt* is defined using non-determinism. The transform therefore involves the interaction between two effects, a tricky topic this paper tries to deal with.

4.2 Right-Distributivity and Local State

We now digress a little to discuss one form of interaction between non-determinism and state. When mixed with other effects, the following laws hold for some monads with non-determinism, but not all:

$$\textbf{right-distributivity}: \quad m \gg\!\!= (\lambda x \rightarrow f_1\ x \mathbin{[\!]} f_2\ x) \ = \ (m \gg\!\!= f_1) \mathbin{[\!]} (m \gg\!\!= f_2) \quad , \tag{15}$$

$$\textbf{right-zero}: \quad m \gg \emptyset \ = \ \emptyset \quad . \tag{16}$$

With some implementations of the monad, it is likely that in the lefthand side of (15), the effect of m happens once, while in the righthand side it happens twice. In (16), the m on the lefthand side may incur some effects that do not happen in the righthand side.

Having (15) and (16) leads to profound consequences on the semantics and implementation of monadic programs. To begin with, (15) implies that $(\mathbin{[\!]})$ be commutative: let $m = m_1 \mathbin{[\!]} m_2$ and $f_1 = f_2 = \mathit{return}$ in (15). Implementation of such non-deterministic monads have been studied by Fischer [?].

When mixed with state, one consequence of (15) is that $\mathit{get} \gg\!\!= (\lambda s \rightarrow f_1\ s \mathbin{[\!]} f_2\ s) = (\mathit{get} \gg\!\!= f_1 \mathbin{[\!]} \mathit{get} \gg\!\!= f_2)$. That is, f_1 and f_2 get the same state regardless of whether *get* is performed outside or inside the non-deterministic branch. Similarly, (16) implies $\mathit{put}\ s \gg \emptyset = \emptyset$ — when a program fails, the changes it performed on the state can be discarded. These requirements imply that each non-deterministic branch has its own copy of the state. Therefore, we will refer to (15) and (16) as *local state laws* in this paper — even though they do not explicitly mention state operators at all! One monad satisfying the laws is $\mathsf{M}_{\{\mathsf{N},\mathsf{S}\ s\}}\ a = s \rightarrow [(a, s)]$, which is the same monad one gets by $\mathsf{StateT}\ s\ (\mathsf{ListT}\ \mathsf{Identity})$ in the Monad Transformer Library [?]. With effect handling [?,?], the monad meets the requirements if we run the handler for state before that for list.

The advantage of having the local state laws is that 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 1. *Let $m :: \mathsf{M}_\epsilon\ a$ 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*

$$\begin{aligned} m \gg\!\!= \lambda x \rightarrow n \gg\!\!= \lambda y \rightarrow f\ x\ y \ &= \\ n \gg\!\!= \lambda y \rightarrow m \gg\!\!= \lambda x \rightarrow f\ x\ y \quad &. \end{aligned} \tag{17}$$

(Notice that m and n can also be typed as $\mathsf{M}_{\epsilon\cup\delta}$.) 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 $m :: \mathsf{M}_\epsilon\ a$ and $n :: \mathsf{M}_\delta\ b$ commute.

Theorem 2. *If right-distributivity* (15) *and right-zero* (16) *hold in addition to the monad laws stated before, non-determinism commutes with any effect ϵ.*

Proof. By induction on syntax of n.

For the rest of Section 4 and 5, we assume that (15) and (16) hold.

Note We briefly justify proofs by induction on the syntax tree. Finite monadic programs can be represented by the free monad constructed out of *return* and the effect operators, which can be represented by an inductively defined data structure, and interpreted by effect handlers [**?**,**?**]. When we say two programs m_1 and m_2 are equal, we mean that they have the same denotation when interpreted by the effect handlers of the corresponding effects, for example, $\mathit{hdNondet}\ (\mathit{hdState}\ s\ m_1) = \mathit{hdNondet}\ (\mathit{hdState}\ s\ m_2)$, where *hdNondet* and *hdState* are respectively handlers for nondeterminism and state. Such equality can be proved by induction on some sub-expression in m_1 or m_2, which are treated like any inductively defined data structure. A more complete treatment is a work in progress, which cannot be fully covered in this paper. (*End of Note*)

4.3 Filtering Using a Stateful, Non-Deterministic Fold

Having dealt with $\mathit{scanl}_+\ (\oplus)\ \mathit{st}$ in Section 4.1, in this section we aim to turn a filter of the form $\mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ \mathit{st})$ to a stateful and non-deterministic *foldr*.

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

$$\begin{aligned}
&\mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ \mathit{st})\ \mathit{xs} \\
=\ &\mathit{guard}\ (\mathit{all}\ \mathit{ok}\ (\mathit{scanl}_+\ (\oplus)\ \mathit{st}\ \mathit{xs})) \gg \mathit{return}\ \mathit{xs} \\
=\ &\mathit{return}\ (\mathit{scanl}_+\ (\oplus)\ \mathit{st}\ \mathit{xs}) \ggg\!= \lambda \mathit{ys} \rightarrow \\
&\mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ \mathit{xs} \\
=\ &\quad \{\ \text{Theorem 1, definition of } \mathit{protect}\text{, monad law}\ \} \\
&\mathit{get} \ggg\!= \lambda \mathit{ini} \rightarrow \mathit{scanlM}\ (\oplus)\ \mathit{st}\ \mathit{xs} \ggg\!= \lambda \mathit{ys} \rightarrow \mathit{put}\ \mathit{ini} \gg \\
&\mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{return}\ \mathit{xs} \\
=\ &\quad \{\ \text{non-determinism commutes with state}\ \} \\
&\mathit{get} \ggg\!= \lambda \mathit{ini} \rightarrow \mathit{scanlM}\ (\oplus)\ \mathit{st}\ \mathit{xs} \ggg\!= \lambda \mathit{ys} \rightarrow \\
&\mathit{guard}\ (\mathit{all}\ \mathit{ok}\ \mathit{ys}) \gg \mathit{put}\ \mathit{ini} \gg \mathit{return}\ \mathit{xs} \\
=\ &\quad \{\ \text{definition of } \mathit{protect}\text{, monad laws}\ \} \\
&\mathit{protect}\ (\mathit{scanlM}\ (\oplus)\ \mathit{st}\ \mathit{xs} \ggg\!= (\mathit{guard} \cdot \mathit{all}\ \mathit{ok}) \gg \mathit{return}\ \mathit{xs}) \quad .
\end{aligned}$$

Recall that $\mathit{scanlM}\ (\oplus)\ \mathit{st}\ \mathit{xs} = \mathit{put}\ \mathit{st} \gg \mathit{foldr}\ (\otimes)\ (\mathit{return}\ [\,])\ \mathit{xs}$. The following theorem fuses a monadic *foldr* with a *guard* that uses its result.

Theorem 3. *Assume that state and non-determinism commute. Let* $(\otimes)$ *be defined as that in scanlM for any given* $(\oplus) :: s \to a \to s$*. We have that for all* $ok :: s \to \mathsf{Bool}$ *and* $xs :: [a]$*:*

$$\begin{aligned}
&foldr\ (\otimes)\ (return\ [\,])\ xs \mathbin{>\!\!>\!\!=} (guard \cdot all\ ok) \mathbin{>\!\!>} return\ xs = \\
&\quad foldr\ (\odot)\ (return\ [\,])\ xs \quad , \\
&\quad \mathbf{where}\ x \odot m = get \mathbin{>\!\!>\!\!=} \lambda st \to guard\ (ok\ (st \oplus x)) \mathbin{>\!\!>} \\
&\qquad\qquad\qquad put\ (st \oplus x) \mathbin{>\!\!>} ((x{:}) \mathbin{\langle\$\rangle} m) \quad .
\end{aligned}$$

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

$$\begin{aligned}
&\quad (x \otimes foldr\ (\otimes)\ (return\ [\,])\ xs) \mathbin{>\!\!>\!\!=} (guard \cdot all\ ok) \mathbin{>\!\!>} return\ (x : xs) \\
&= \quad \{ \text{ definition of } (\otimes) \ \} \\
&\quad get \mathbin{>\!\!>\!\!=} \lambda st \to \\
&\quad (((st \oplus x){:}) \mathbin{\langle\$\rangle} (put\ (st \oplus x) \mathbin{>\!\!>} foldr\ (\otimes)\ (return\ [\,])\ xs)) \mathbin{>\!\!>\!\!=} \\
&\quad (guard \cdot all\ ok) \mathbin{>\!\!>} return\ (x : xs) \\
&= \quad \{ \text{ monad laws, (5), and (6) } \} \\
&\quad get \mathbin{>\!\!>\!\!=} \lambda st \to put\ (st \oplus x) \mathbin{>\!\!>} \\
&\quad foldr\ (\otimes)\ (return\ [\,])\ xs \mathbin{>\!\!>\!\!=} \lambda ys \to \\
&\quad guard\ (all\ ok\ (st \oplus x : ys)) \mathbin{>\!\!>} return\ (x : xs) \\
&= \quad \{ \text{ since } guard\ (p \wedge q) = guard\ q \mathbin{>\!\!>} guard\ p \ \} \\
&\quad get \mathbin{>\!\!>\!\!=} \lambda st \to put\ (st \oplus x) \mathbin{>\!\!>} \\
&\quad foldr\ (\otimes)\ (return\ [\,])\ xs \mathbin{>\!\!>\!\!=} \lambda ys \to \\
&\quad guard\ (ok\ (st \oplus x)) \mathbin{>\!\!>} guard\ (all\ ok\ ys) \mathbin{>\!\!>} \\
&\quad return\ (x : xs) \\
&= \quad \{ \text{ nondeterminism commutes with state } \} \\
&\quad get \mathbin{>\!\!>\!\!=} \lambda st \to guard\ (ok\ (st \oplus x)) \mathbin{>\!\!>} put\ (st \oplus x) \mathbin{>\!\!>} \\
&\quad foldr\ (\otimes)\ (return\ [\,])\ xs \mathbin{>\!\!>\!\!=} \lambda ys \to \\
&\quad guard\ (all\ ok\ ys) \mathbin{>\!\!>} return\ (x : xs) \\
&= \quad \{ \text{ monad laws and definition of } (\langle\$\rangle) \ \} \\
&\quad get \mathbin{>\!\!>\!\!=} \lambda st \to guard\ (ok\ (st \oplus x)) \mathbin{>\!\!>} put\ (st \oplus x) \mathbin{>\!\!>} \\
&\quad (x{:}) \mathbin{\langle\$\rangle} (foldr\ (\otimes)\ (return\ [\,])\ xs \mathbin{>\!\!>\!\!=} \lambda ys \to guard\ (all\ ok\ ys) \mathbin{>\!\!>} return\ xs) \\
&= \quad \{ \text{ induction } \} \\
&\quad get \mathbin{>\!\!>\!\!=} \lambda st \to guard\ (ok\ (st \oplus x)) \mathbin{>\!\!>} put\ (st \oplus x) \mathbin{>\!\!>} \\
&\quad (x{:}) \mathbin{\langle\$\rangle} foldr\ (\odot)\ (return\ [\,])\ xs \\
&= \quad \{ \text{ definition of } (\odot) \ \} \\
&\quad foldr\ (\odot)\ (return\ [\,])\ (x : xs) \quad .
\end{aligned}$$

This proof is instructive due to extensive use of commutativity.

In summary, we now have this corollary performing $filt\ (all\ ok \cdot scanl_{+}\ (\oplus)\ st)$ using a non-deterministic and stateful foldr:

Corollary 1. *Let* $(\odot)$ *be defined as in Theorem 3. If state and non-determinism commute, we have:*

$$\begin{array}{l} \mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ \mathit{st})\ \mathit{xs} = \\ \quad \mathit{protect}\ (\mathit{put}\ \mathit{st} \gg \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ \mathit{xs}) \quad . \end{array}$$

5 Monadic Hylomorphism

To recap what we have done, we started with a specification of the form $\mathit{unfoldM}\ p\ f\ z \gg\!= \mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ \mathit{st})$, where $f :: b \rightarrow \mathsf{M}_{\mathsf{N}}\ (a, b)$, and have shown that

$$\begin{array}{ll} & \mathit{unfoldM}\ p\ f\ z \gg\!= \mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ \mathit{st}) \\ = & \{ \text{ Corollary 1, with } (\odot) \text{ defined as in Theorem 3 } \} \\ & \mathit{unfoldM}\ p\ f\ z \gg\!= \lambda \mathit{xs} \rightarrow \mathit{protect}\ (\mathit{put}\ \mathit{st} \gg \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])\ \mathit{xs}) \\ = & \{ \text{ nondeterminism commutes with state } \} \\ & \mathit{protect}\ (\mathit{put}\ \mathit{st} \gg \mathit{unfoldM}\ p\ f\ z \gg\!= \mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])) \quad . \end{array}$$

The final task is to fuse $\mathit{unfoldM}\ p\ f$ with $\mathit{foldr}\ (\odot)\ (\mathit{return}\ [\,])$.

5.1 Monadic Hylo-Fusion

In a pure setting, it is known that, provided that the unfolding phase terminates, $\mathit{foldr}\ (\otimes)\ e \cdot \mathit{unfoldr}\ p\ f$ is the unique solution of hylo in the equation below [**?**]:

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

Hylomorphisms with monadic folds and unfolds are a bit tricky. Pardo [**?**] discussed hylomorphism for regular base functors, where the unfolding phase is monadic while the folding phase is pure. As for the case when both phases are monadic, he noted "the drawback ... is that they cannot be always transformed into a single function that avoids the construction of the intermediate data structure."

For our purpose, we focus our attention on lists, and have a theorem fusing the monadic unfolding and folding phases under a side condition. Given $(\otimes) :: b \rightarrow \mathsf{M}_{\epsilon}\ c \rightarrow \mathsf{M}_{\epsilon}\ c$, $e :: c$, $p :: a \rightarrow \mathsf{Bool}$, and $f :: a \rightarrow \mathsf{M}_{\epsilon}\ (b, a)$ (with no restriction on ϵ), consider the expression:

$$\mathit{unfoldM}\ p\ f \ggg \mathit{foldr}\ (\otimes)\ (\mathit{return}\ e) \ :: \ a \rightarrow \mathsf{M}_{\epsilon}\ c \quad .$$

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 $x \otimes m = (m \gg\!= \lambda y \rightarrow x \otimes \mathit{return}\ y)$. That is, it does not matter whether the effects of m happens inside or outside $(x\otimes)$.

Theorem 4. *For all* ϵ, $(\otimes) :: a \rightarrow \mathsf{M}_{\epsilon}\ c \rightarrow \mathsf{M}_{\epsilon}\ c$, $m :: \mathsf{M}_{\epsilon}\ c$, $p :: b \rightarrow \mathsf{Bool}$, $f :: b \rightarrow \mathsf{M}_{\epsilon}\ (a, c)$*, we have that* $\mathit{unfoldM}\ p\ f \ggg \mathit{foldr}\ (\otimes)\ m = \mathit{hyloM}\ (\otimes)\ m\ p\ f$*, defined by:*

$$\begin{array}{l} \mathit{hyloM}\ (\otimes)\ m\ p\ f\ y \\ \quad \mid p\ y \qquad\quad = m \\ \quad \mid \mathit{otherwise} = f\ y \gg\!= \lambda(x,z) \rightarrow x \otimes \mathit{hyloM}\ (\otimes)\ m\ p\ f\ z \quad , \end{array}$$

if $x \otimes n = n \gg\!= ((x\otimes) \cdot \mathit{return})$ *for all* $n :: \mathsf{M}_\epsilon\ c$*, 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 $\mathit{unfoldM}\ p\ f \ggg \mathit{foldr}\ (\otimes)\ m$ is a fixed-point of the recursive equations of hyloM. When $p\ y$ holds, it is immediate that

$$\mathit{return}\ [\,] \gg\!= \mathit{foldr}\ (\otimes)\ m \ = \ m \quad .$$

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

$$\begin{array}{cl} & \mathit{unfoldM}\ p\ f\ y \gg\!= \mathit{foldr}\ (\otimes)\ m \\ = & \{\ \text{definition of } \mathit{unfoldM},\ \neg\ (p\ y)\ \} \\ & (f\ y \gg\!= (\lambda(x,z) \rightarrow (x{:})\ \langle\$\rangle\ \mathit{unfoldM}\ p\ f\ z)) \gg\!= \\ & \mathit{foldr}\ (\otimes)\ m \\ = & \{\ \text{monadic law (1) and } \mathit{foldr}\ \} \\ & f\ y \gg\!= (\lambda(x,z) \rightarrow \mathit{unfoldM}\ p\ f\ z \gg\!= \lambda xs \rightarrow \\ & \qquad\qquad x \otimes \mathit{foldr}\ (\otimes)\ m\ xs) \quad . \end{array}$$

We focus on the expression inside the λ abstraction:

$$\begin{array}{cl} & \mathit{unfoldM}\ p\ f\ z \gg\!= (\lambda xs \rightarrow x \otimes \mathit{foldr}\ (\otimes)\ m\ xs) \\ = & \{\ \text{assumption: } x \otimes n = n \gg\!= ((x\otimes) \cdot \mathit{return}), \text{ see below}\ \} \\ & \mathit{unfoldM}\ p\ f\ z \gg\!= (\mathit{foldr}\ (\otimes)\ m \ggg ((x\otimes) \cdot \mathit{return})) \\ = & \{\ \text{since } m \gg\!= (f \ggg g) = (m \gg\!= f) \gg\!= g\ \} \\ & (\mathit{unfoldM}\ p\ f\ z \gg\!= \mathit{foldr}\ (\otimes)\ m) \gg\!= ((x\otimes) \cdot \mathit{return}) \\ = & \{\ \text{by assumption } x \otimes m = m \gg\!= ((x\otimes) \cdot \mathit{return})\ \} \\ & x \otimes (\mathit{unfoldM}\ p\ f\ z \gg\!= \mathit{foldr}\ (\otimes)\ m) \quad . \end{array}$$

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

Now that $\mathit{unfoldM}\ p\ f\ z \gg\!= \mathit{foldr}\ (\otimes)\ m$ is a fixed-point, we may conclude that it equals $\mathit{hyloM}\ (\otimes)\ m\ p\ f$ if the latter has a unique fixed-point, which is guaranteed by the well-foundedness condition. 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 (where $(=\!\ll)$ is $(\gg\!=)$ written reversed). 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 [**?**] 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 [?]. (*End of Note*)

TODO: LLNCS docs recommend not using vspace command Theorem 4 does

not rely on the *local state laws* (15) and (16), and does not put restriction on ϵ. To apply the theorem to our particular case, we have to show that its preconditions hold for our particular $(\odot)$ — for that we will need (15) and (16). In the lemma below we slightly generalise $(\odot)$ in Theorem 3:

Lemma 1. *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* $(\odot) :: a \to \mathsf{M}_\epsilon\ b \to \mathsf{M}_\epsilon\ b$ *with* $\{\mathsf{N}, \mathsf{S}\ s\} \subseteq \epsilon$*:*

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

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

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

5.2 Summary, and Solving n-Queens

To conclude our derivation, a problem formulated as $\mathit{unfoldM}\ p\ f\ z \gg\!= \mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ st)$ can be solved by a hylomorphism. Define:

$$\begin{aligned} & \mathit{solve} :: \{\mathsf{N}, \mathsf{S}\ s\} \subseteq \epsilon \Rightarrow (b \to \mathsf{Bool}) \to (b \to \mathsf{M}_\epsilon\ (a, b)) \to (s \to \mathsf{Bool}) \to \\ & \qquad\qquad (s \to a \to s) \to s \to b \to \mathsf{M}_\epsilon\ [a] \\ & \mathit{solve}\ p\ f\ \mathit{ok}\ (\oplus)\ st\ z = \mathit{protect}\ (\mathit{put}\ st \gg \mathit{hyloM}\ (\odot)\ (\mathit{return}\ [\,])\ p\ f\ z) \quad , \\ & \quad \textbf{where}\ x \odot m = \mathit{get} \gg\!= \lambda st \to \mathit{guard}\ (\mathit{ok}\ (st \oplus x)) \gg \\ & \qquad\qquad \mathit{put}\ (st \oplus x) \gg ((x{:}) \mathbin{\langle\$\rangle} m) \quad . \end{aligned}$$

Corollary 2. *If the relation* $(\neg \cdot p)\,?\cdot \mathit{snd} \cdot (=\!\ll) \cdot f$ *is well-founded, and* (15) *and* (16) *hold in addition to the other laws, we have*

$$\begin{aligned} & \mathit{unfoldM}\ p\ f\ z \gg\!= \mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ st) = \\ & \quad \mathit{solve}\ p\ f\ \mathit{ok}\ (\oplus)\ st\ z \quad . \end{aligned}$$

n-Queens Solved Recall that

$$\begin{aligned} \mathit{queens}\ n & = \mathit{perm}\ [0\mathinner{..}n-1] \gg\!= \mathit{filt}\ \mathit{safe} \\ & = \mathit{unfoldM}\ \mathit{null}\ \mathit{select}\ [0\mathinner{..}n-1] \gg\!= \mathit{filt}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_+\ (\oplus)\ (0, [\,], [\,])) \quad , \end{aligned}$$

where the auxiliary functions select, ok, $(\oplus)$ are defined in Section 3. The function select cannot be applied forever since the length of the given list decreases after each call. Therefore, Corollary 2 applies, and we have $\mathit{queens}\ n = \mathit{solve}\ \mathit{null}\ \mathit{select}\ \mathit{ok}\ (\oplus)\ (0, [\,], [\,])\ [0\mathinner{..}n-1]$. Expanding the definitions we get:

$$
\begin{array}{l}
\mathit{queens} :: \{\mathsf{N}, \mathsf{S}\ (\mathsf{Int}, [\mathsf{Int}], [\mathsf{Int}])\} \subseteq \epsilon \Rightarrow \mathsf{Int} \rightarrow \mathsf{M}_\epsilon\ [\mathsf{Int}] \\
\mathit{queens}\ n = \mathit{protect}\ (\mathit{put}\ (0, [\,], [\,]) \gg \mathit{queensBody}\ [0 \mathinner{..} n-1]) \quad , \\
\mathit{queensBody} :: \{\mathsf{N}, \mathsf{S}\ (\mathsf{Int}, [\mathsf{Int}], [\mathsf{Int}])\} \subseteq \epsilon \Rightarrow [\mathsf{Int}] \rightarrow \mathsf{M}_\epsilon\ [\mathsf{Int}] \\
\mathit{queensBody}\ [\,] \ \ = \mathit{return}\ [\,] \\
\mathit{queensBody}\ \mathit{xs} = \mathit{select}\ \mathit{xs} \gg\!= \lambda(x, \mathit{ys}) \rightarrow \\
\qquad\qquad\qquad\qquad \mathit{get} \gg\!= \lambda \mathit{st} \rightarrow \mathit{guard}\ (\mathit{ok}\ (\mathit{st} \oplus x)) \gg \\
\qquad\qquad\qquad\qquad \mathit{put}\ (\mathit{st} \oplus x) \gg ((x{:}) \mathbin{\langle\$\rangle} \mathit{queensBody}\ \mathit{ys}) \quad , \\
\quad \mathbf{where}\ (i, \mathit{us}, \mathit{ds}) \oplus x = (1+i, (i+x) : \mathit{us}, (i-x) : \mathit{ds}) \\
\qquad\qquad \mathit{ok}\ (\_, u : \mathit{us}, d : \mathit{ds}) = (u \notin \mathit{us}) \wedge (d \notin \mathit{ds}) \quad .
\end{array}
$$

This 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, the local state laws imply 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, is rather tricky. One might believe that $\mathsf{M}\ a = s \rightarrow ([a], s)$ is a natural implementation of such a monad. The usual, naive implementation of $(\gg\!=)$ using this representation, however, does not satisfy left-distributivity (9), violates monad laws, and is therefore not even a monad. [4]

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 $([\!])$ is still associative, but certainly cannot be commutative. As mentioned in Section 4.2, right-distributivity (15) implies commutativity of $([\!])$. Contravariantly, (15) cannot be true when the state is global. Right-zero (16) does not hold either: $\emptyset$ simply fails, while $\mathit{put}\ s \gg \emptyset$, for example, fails with an altered global state. These significantly limit the properties we may have.

The aim of this section is to appeal to intuition and see what happens when we work with a global state monad: what pitfall we may encounter, and what programming pattern we may use, to motivate the more formal treatment in Section 7.

[4] 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 (9) and the monad laws, have been proposed [**?**,**?**]. Effect handlers (e.g. Wu [**?**] and Kiselyov and Ishii [**?**]) do produce correct implementations if we run the handler for non-determinism before that of state.

6.1 The Global State Law

We have already discussed general laws for nondeterministic monads (laws (7) through (10)), as well as laws which govern the interaction between state and nondeterminism in a local state setting (laws (15) and (16)). For global state semantics, an alternative law is required to govern the interactions between nondeterminism and state. We call this the *global state law*, to be discussed in more detail in Section 7.2.

$$\textbf{put-or}: \quad (\mathit{put}\ s \gg m) \mathbin{[]} n = \ \mathit{put}\ s \gg (m \mathbin{[]} n) \quad , \tag{18}$$

This law allows the lifting of a *put* operation from the left branch of a nondeterministic choice, an operation which does not preserve meaning under local-state semantics: suppose for example that $m = \emptyset$, then by (16) and (8), the left-hand side of the equation is equal to n, whereas by (8), the right-hand side of the equation is equal to $\mathit{put}\ s \gg n$.

By itself, this law leaves us free to choose from a large space of semantic domain implementations with different properties. For example, in any given implementation, the programs $\mathit{return}\ x \mathbin{[]} \mathit{return}\ y$ and $\mathit{return}\ y \mathbin{[]} \mathit{return}\ x$ may be considered semantically identical, or they may be considered semantically distinct. The same goes for the programs $\mathit{return}\ x \mathbin{[]} \mathit{return}\ x$ and $\mathit{return}\ x$, or the programs $(\mathit{put}\ s \gg \mathit{return}\ x) \mathbin{[]} m$ and $(\mathit{put}\ s \gg \mathit{return}\ x) \mathbin{[]} \mathit{put}\ s \gg m$. Additional axioms will be introduced as needed to cover these properties.

6.2 Chaining Using Non-deterministic Choice

In backtracking algorithms that keep a global state, it is a common pattern to 1. update the current state to its next step, 2. recursively search for solutions, and 3. roll back the state to the previous step. To implement such pattern as a monadic program, one might come up with something like the code below:

$$\mathit{modify}\ \mathit{next} \gg \mathit{search} \gg\!= \mathit{modReturn}\ \mathit{prev} \quad .$$

where *next* advances the state, *prev* undoes the modification of *next* ($\mathit{prev} \cdot \mathit{next} = \mathit{id}$), and *modify* and *modReturn* are defined by:

$$\begin{aligned}
&\mathit{modify}\ f && = \mathit{get} \gg\!= (\mathit{put} \cdot f) \quad , \\
&\mathit{modReturn}\ f\ v && = \mathit{modify}\ f \gg \mathit{return}\ v \quad .
\end{aligned}$$

Let the initial state be *st* and assume that *search* 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 (9), however,

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

which, with a global state, means that m_2 starts with state st, after which the state is rolled back too early to $prev\ st$. The computation m_3 starts with $prev\ st$, after which the state is rolled too far to $prev\ (prev\ st)$.

In fact, one cannot guarantee that $modReturn\ prev$ is always executed — if $search$ fails, we get $modify\ next \gg search \gg\!= modReturn\ prev = modify\ next \gg \emptyset \gg\!= modReturn\ prev = modify\ next \gg \emptyset$. Thus the state is advanced to $next\ st$, but not rolled back to st.

We need a way to say that "*modify next* and *modReturn prev* are run exactly once, respectively before and after all non-deterministic branches in *solve*." Fortunately, we have discovered a curious technique. Define

$$\begin{aligned}&side :: \mathsf{N} \in \epsilon \Rightarrow \mathsf{M}_\epsilon\ a \to \mathsf{M}_\epsilon\ b\\ &side\ m = m \gg \emptyset \quad .\end{aligned}$$

Since non-deterministic branches are executed sequentially, the program

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

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.

The reader might wonder: now that we are using $([\!])$ as a sequencing operator, does it simply coincide with $(\gg)$? Recall that we still have left-distributivity (9) and, therefore, $(m_1 \mathbin{[\!]} m_2) \gg n$ equals $(m_1 \gg n) \mathbin{[\!]} (m_2 \gg n)$. That is, $([\!])$ acts as "insertion points", where future code followed by $(\gg)$ can be inserted into! This is certainly a dangerous feature, whose undisciplined use can lead to chaos. However, we will exploit this feature and develop a safer programming pattern in the next section.

6.3 State-Restoring Operations

The discussion above suggests that one can implement backtracking, in a global-state setting, by using $([\!])$ and *side* appropriately. We can even go a bit further by defining the following variations of *put*, which restores the original state when it is backtracked over:

$$\begin{aligned}&put_{\mathsf{R}} :: \{\mathsf{S}\ s, \mathsf{N}\} \subseteq \epsilon \Rightarrow s \to \mathsf{M}_\epsilon\ ()\\ &put_{\mathsf{R}}\ s = get \gg\!= \lambda s_0 \to put\ s \mathbin{[\!]} side\ (put\ s_0) \quad .\end{aligned}$$

To help build understanding for put_{R}, Figure 3 shows the flow of execution for the expression $(put_{\mathsf{R}}\ t \gg ret\ x) \mathbin{[\!]} ret\ y$. Initially, the state is s; it gets modified to t at the $put\ t$ node after which the value x is output with the working state t. Then, we backtrack (since we're using global-state semantics, the state modification caused by $put\ t$ is not reversed), arriving at $put\ s$, which resets the state to s, immediately fails, and backtracks to the right branch of the topmost $([\!])$. There the value y is output with working state s.

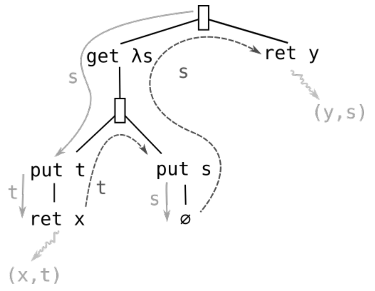

Fig. 3: Illustration of state-restoring put

For some further intuition about put_{R}, consider $put_{\mathsf{R}}\ s \gg comp$ where $comp$ is some arbitrary computation:

$$\begin{aligned}
&\quad put_{\mathsf{R}}\ s \gg comp \\
&= (get \gg\!= \lambda s_0 \rightarrow put\ s \mathbin{[\!]} side\ (put\ s_0)) \gg comp \\
&= \quad \{ \text{ monad law, left-distributivity (9) } \} \\
&\quad get \gg\!= \lambda s_0 \rightarrow (put\ s \gg comp) \mathbin{[\!]} (side\ (put\ s_0) \gg comp) \\
&= \quad \{ \text{ by (10), } \emptyset \gg comp = \emptyset \ \} \\
&\quad get \gg\!= \lambda s_0 \rightarrow (put\ s \gg comp) \mathbin{[\!]} side\ (put\ s_0) \quad .
\end{aligned}$$

Thanks to left-distributivity (9), $(\gg comp)$ is promoted into $([\!])$. Furthermore, the $(\gg comp)$ after $side\ (put\ s_0)$ is discarded by (10). In words, $put_{\mathsf{R}}\ s \gg comp$ saves the current state, computes $comp$ using state s, and restores the saved state! The subscript R stands for "restore." Note also that $(put_{\mathsf{R}}\ s \gg m_1) \gg m_2) = put_{\mathsf{R}}\ s \gg (m_1 \gg m_2)$ — the state restoration happens in the end.

The behaviour of put_{R}, however, is still rather tricky. It is instructive comparing

(a) $return\ x$,
(b) $put\ s \gg return\ x$, and
(c) $put_{\mathsf{R}}\ s \gg return\ x$.

When run in initial state s_0, they all yield x as the result. The final states after running (a), (b) and (c) are s_0, s and s_0, respectively. However, (c) does *not* behave identically to (a) in all contexts! For example, in the context $(\gg get)$, we can tell them apart: $return\ x \gg get$ returns s_0, while $put_{\mathsf{R}}\ s \gg return\ x \gg get$ returns s, even though the program yields final state s_0.

We wish that put_{R}, when run with a global state, satisfies laws (11) through (16) — the state laws and the *local* state laws. If so, one could take a program written for a local state monad, replace all occurrences of put by put_{R}, and run the program with a global state. Unfortunately this is not the case: put_{R} does satisfy *put-put* (11) and *put-get* (13), but *get-put* (12) fails — $get \gg\!= put_{\mathsf{R}}$ and $return$ () can be told apart by some contexts, for example $(\gg put\ t)$. To see that, we calculate:

$$
\begin{array}{ll}
 & (\mathit{get} \ggg\!= \mathit{put}_{\mathsf{R}}) \gg \mathit{put}\ t \\
= & (\mathit{get} \ggg\!= \lambda s \to \mathit{get} \ggg\!= \lambda s_0 \to \mathit{put}\ s \mathbin{[\!]} \mathit{side}\ (\mathit{put}\ s_0)) \gg \mathit{put}\ t \\
= & \quad \{\ \mathit{get}\text{-}\mathit{get}\ \} \\
 & (\mathit{get} \ggg\!= \lambda s \to \mathit{put}\ s \mathbin{[\!]} \mathit{side}\ (\mathit{put}\ s)) \gg \mathit{put}\ t \\
= & \quad \{\ \text{monad laws, left-distributivity}\ \} \\
 & \mathit{get} \ggg\!= \lambda s \to (\mathit{put}\ s \gg \mathit{put}\ t) \mathbin{[\!]} \mathit{side}\ (\mathit{put}\ s) \\
= & \quad \{\ \mathit{put}\text{-}\mathit{put}\ \} \\
 & \mathit{get} \ggg\!= \lambda s \to \mathit{put}\ t \mathbin{[\!]} \mathit{side}\ (\mathit{put}\ s) \quad .
\end{array}
$$

Meanwhile, $\mathit{return}\ () \gg \mathit{put}\ t = \mathit{put}\ t$, which does not behave in the same way as $\mathit{get} \ggg\!= \lambda s \to \mathit{put}\ t \mathbin{[\!]} \mathit{side}\ (\mathit{put}\ s)$ when $s \neq t$.

In a global-state setting, the left-distributivity law (9) makes it tricky to reason about combinations of $([\!])$ and $(\ggg\!=)$ operators. Suppose we have a program $(m \mathbin{[\!]} n)$, and we construct an extended program by binding a continuation f to it such that we get $(m \mathbin{[\!]} n) \ggg\!= f$ (where f might modify the state). Under global-state semantics, the evaluation of the right branch is influenced by the state modifications performed by evaluating the left branch. So by (9), this means that when we get to evaluating the n subprogram in the extended program, it will do so with a different initial state (the one obtained after running $m \ggg\!= f$) compaired against the initial state in the original program (the one obtained by running m). In other words, placing our program in a different context changed the meaning of one of its subprograms. So it is difficult to reason about programs compositionally in this setting — some properties hold only when we take the entire program into consideration.

It turns out that all properties we need do hold, provided that *all* occurrences of *put* are replaced by $\mathit{put}_{\mathsf{R}}$ — problematic contexts such as $\mathit{put}\ t$ above are thus ruled out. However, that "all *put* are replaced by $\mathit{put}_{\mathsf{R}}$" is a global property, and to properly talk about it we have to formally define contexts, which is what we will do in Section 7.

7 Laws and Translation for Global State Monad

In this section we give a more formal treatment of non-deterministic global state monad. We propose laws such a monad should satisfy — to the best of our knowledge, we are the first to propose these laws. The laws turn out to be rather intricate. To make sure that there exists a model, an implementation is proposed in the appendix, and it is verified in Coq that the laws and some additional theorems are satisfied.

The ultimate goal, however, is to show the following property: given a program written for a local-state monad, if we replace all occurrences of *put* by $\mathit{put}_{\mathsf{R}}$, the resulting program yields the same result when run with a global-state monad. This allows us to painlessly port our previous algorithm to work with a global state. To show this we first introduce a syntax for nondeterministic and stateful monadic programs and contexts. Then we imbue these programs with global-state semantics. Finally we define the function that performs the translation just described, and prove that this translation is correct.

7.1 Programs and Contexts

```
data Prog a where
  Return :: a → Prog a
  ∅      :: Prog a
  (⫿)    :: Prog a → Prog a → Prog a
  Get    :: (S → Prog a) → Prog a
  Put    :: S → Prog a → Prog a
```

(a)

```
data Env (l :: [∗]) where
  Nil :: Env '[]
  Cons :: a → Env l → Env (a : l)
type OProg e a = Env e → Prog a
```

(b)

```
data Ctx e1 a e2 b where
  □      :: Ctx e a e a
  COr1   :: Ctx e1 a e2 b → OProg e2 b
         → Ctx e1 a e2 b
  COr2   :: OProg e2 b → Ctx e1 a e2 b
         → Ctx e1 a e2 b
  CPut   :: (Env e2 → S) → Ctx e1 a e2 b
         → Ctx e1 a e2 b
  CGet   :: (S → Bool) → Ctx e1 a (S : e2) b
         → (S → OProg e2 b) → Ctx e1 a e2 b
  CBind1 :: Ctx e1 a e2 b → (b → OProg e2 c)
         → Ctx e1 a e2 c
  CBind2 :: OProg e2 a → Ctx e1 b (a : e2) c
         → Ctx e1 b e2 c
```

(c)

```
  run   :: Prog a → Dom a
  ⟨ret⟩ :: a → Dom a
  ⟨∅⟩   :: Dom a
  ⟨⫿⟩   :: Dom a → Dom a → Dom a
  ⟨get⟩ :: (S → Dom a) → Dom a
  ⟨put⟩ :: S → Dom a → Dom a
```

(d)

Fig. 4: (a) Syntax for programs. (b) Environments and open programs. (c) Syntax for contexts. (d) Semantic domain.

In the previous sections we have been mixing syntax and semantics, which we avoid in this section by defining the program syntax as a free monad. This way we avoid the need for a type-level distinction between programs with local-state semantics and programs with global-state semantics. Figure 4(a) defines a syntax for nondeterministic, stateful, closed programs Prog, where the Get and Put constructors take continuations as arguments, and the (≫=) operator is defined as follows:

$$
\begin{array}{lcl}
(\gg\!=) :: \mathsf{Prog}\ a \to (a \to \mathsf{Prog}\ b) \to \mathsf{Prog}\ b \\
\mathsf{Return}\ x \gg\!= f = f\ x \\
\emptyset \qquad\quad\ \gg\!= f = \emptyset \\
(m \mathbin{[\!]} n)\ \gg\!= f = (m \gg\!= f) \mathbin{[\!]} (n \gg\!= f) \\
\mathsf{Get}\ k \qquad \gg\!= f = \mathsf{Get}\ (\lambda s \to k\ s \gg\!= f) \\
\mathsf{Put}\ s\ m\ \gg\!= f = \mathsf{Put}\ s\ (m \gg\!= k) \quad .
\end{array}
$$

One can see that $(\gg\!=)$ is defined as a purely syntactical manipulation, and its definition has laws (9) and (10) built-in.

The meaning of such a monadic program is determined by a semantic domain of our choosing, which we denote with Dom, and its corresponding domain operators $\langle ret \rangle$, $\langle \varnothing \rangle$, $\langle get \rangle$, $\langle put \rangle$ and $(\langle [\!] \rangle)$ (see figure 4(d)). The $run :: \mathsf{Prog}\ a \to \mathsf{Dom}\ a$ function "runs" a program $\mathsf{Prog}\ a$ into a value in the semantic domain $\mathsf{Dom}\ a$:

$$
\begin{array}{ll}
run\ (\mathsf{Return}\ x) & = \langle ret \rangle\ x \\
run\ \emptyset & = \langle \varnothing \rangle \\
run\ (m_1 \mathbin{[\!]} m_2) & = run\ m_1 \mathbin{\langle [\!] \rangle} run\ m_2 \\
run\ (\mathsf{Get}\ k) & = \langle get \rangle\ (\lambda s \to run\ (k\ s)) \\
run\ (\mathsf{Put}\ s\ m) & = \langle put \rangle\ s\ (run\ m) \quad .
\end{array}
$$

Note that no $\langle \gg\!= \rangle$ operator is required to define run; in other words, Dom need not be a monad. In fact, as we will see later, we will choose our implementation in such a way that there does not exist a bind operator for run.

7.2 Modeling Global State Semantics

We impose the laws upon Dom and the domain operators to ensure the semantics of a non-backtracking (global-state), nondeterministic, stateful computation for our programs. Naturally, we need laws analogous to the state laws and nondeterminism laws to hold for our semantic domain. As it is not required that a bind operator $((\gg\!=) :: \mathsf{Dom}\ a \to (a \to \mathsf{Dom}\ b) \to \mathsf{Dom}\ b)$ can be defined for the semantic domain (and we will later argue that it *cannot* be defined for the domain, given the laws we impose on it), the state laws ((11) through (14)) must be reformulated to fit the continuation-passing style of the semantic domain operators.

$$\langle put \rangle\ s\ (\langle put \rangle\ t\ p) = \langle put \rangle\ t\ p \quad , \tag{19}$$

$$\langle put \rangle\ s\ (\langle get \rangle\ k) = \langle put \rangle\ s\ (k\ s) \quad , \tag{20}$$

$$\langle get \rangle\ (\lambda s \to \langle put \rangle\ s\ m) = m \quad , \tag{21}$$

$$\langle get \rangle\ (\lambda s \to \langle get \rangle\ (\lambda t \to k\ s\ t)) = \langle get \rangle\ (\lambda s \to k\ s\ s) \quad . \tag{22}$$

As for the nondeterminism laws ((7), (8), (9), (10)), we can simply omit the ones that mention at the semantic level $(\gg\!=)$ as these are proven at the syntactic level: their proof follows immediately from Prog's definition of $(\gg\!=)$.

$$(m \mathbin{\langle [\!] \rangle} n) \mathbin{\langle [\!] \rangle} p = m \mathbin{\langle [\!] \rangle} (n \mathbin{\langle [\!] \rangle} p) \quad , \tag{23}$$

$$\langle\varnothing\rangle \mathbin{\langle[\!]\rangle} m = m \mathbin{\langle[\!]\rangle} \langle\varnothing\rangle = m \quad . \tag{24}$$

We also reformulate the global-state law (18):

$$\langle put\rangle\ s\ p \mathbin{\langle[\!]\rangle} q = \langle put\rangle\ s\ (p \mathbin{\langle[\!]\rangle} q) \quad . \tag{25}$$

It turns out that, apart from the **put-or** law, our proofs require certain additional properties regarding commutativity and distributivity which we introduce here:

$$\begin{aligned}&\langle get\rangle\ (\lambda s \to \langle put\rangle\ (t\ s)\ p \mathbin{\langle[\!]\rangle} \langle put\rangle\ (u\ s)\ q \mathbin{\langle[\!]\rangle} \langle put\rangle\ s\ \langle\varnothing\rangle) = \\ &\langle get\rangle\ (\lambda s \to \langle put\rangle\ (u\ s)\ q \mathbin{\langle[\!]\rangle} \langle put\rangle\ (t\ s)\ p \mathbin{\langle[\!]\rangle} \langle put\rangle\ s\ \langle\varnothing\rangle) \quad ,\end{aligned} \tag{26}$$

$$\langle put\rangle\ s\ (\langle ret\rangle\ x \mathbin{\langle[\!]\rangle} p) = \langle put\rangle\ s\ (\langle ret\rangle\ x) \mathbin{\langle[\!]\rangle} \langle put\rangle\ s\ p \quad . \tag{27}$$

These laws are not considered general "global state" laws, because it is possible to define reasonable implementations of global state semantics that violate these laws, and because they are not exclusive to global state semantics.

The $\langle[\!]\rangle$ operator is not, in general, commutative in a global state setting. However, we will require that the order in which results are computed does not matter. Furthermore we require that the implementation is agnostic with repect to the order of $\langle put\rangle$ operations as long as the exact same results are computed with the exact same state at the time of their computation. These properties are enforced by law (26). Implementations that present the results as an ordered list violate this law, as are implementations that record the order of proper state changes.

In global-state semantics, $\langle put\rangle$ operations cannot, in general, distribute over $\langle[\!]\rangle$. However, an implementation may permit distributivity if certain conditions are met. Law (27) states that a $\langle put\rangle$ operation distributes over a nondeterministic choice if the left branch of that choice simply returns a value. An example of an implementation that violates this law would be one that applies a given function $f :: s \to s$ to the state after each return. Such an implementation would conform to an alternative law $\langle put\rangle\ x\ (\langle ret\rangle\ w \langle[\!]\rangle q) = \langle put\rangle\ x\ (\langle ret\rangle\ w) \langle[\!]\rangle \langle put\rangle\ (f\ x)\ q$. Law (27) only holds if the implementation of Dom does not permit the definition of a bind operator $(\langle\gg\!=\rangle) :: \mathsf{Dom}\ a \to (a \to \mathsf{Dom}\ b) \to \mathsf{Dom}\ b$. Consider for instance the following program:

$$\langle put\rangle\ x\ (\langle ret\rangle\ w \mathbin{\langle[\!]\rangle} \langle get\rangle\ \langle ret\rangle) \mathbin{\langle\gg\!=\rangle} \lambda z \to \langle put\rangle\ y\ (\langle ret\rangle\ z) \quad .$$

If (27) holds, this program should be equal to

$$(\langle put\rangle\ x\ (\langle ret\rangle\ w) \mathbin{\langle[\!]\rangle} \langle put\rangle\ x\ (\langle get\rangle\ \langle ret\rangle)) \mathbin{\langle\gg\!=\rangle} \lambda z \to \langle put\rangle\ y\ (\langle ret\rangle\ z) \quad .$$

However, Figure 5 proves that the first program can be reduced to $\langle put\rangle\ y\ (\langle ret\rangle\ w \langle[\!]\rangle \langle ret\rangle\ y)$, whereas the second program is equal to $\langle put\rangle\ y\ (\langle ret\rangle\ w \langle[\!]\rangle \langle ret\rangle\ x)$, which clearly does not always have the same result.

It is worth remarking that, even if we don't impose law (26), this requirement disqualifies the most straightforward candidate for the semantic domain, $\mathsf{ListT}\ (\mathsf{State}\ s)$, as a bind operator can be defined for it.

In Appendix A we present an implementation of Dom and its operators that satisfies all the laws in this section, for which we provide *machine-verified proofs*, and which does not permit the implementation of a sensible bind operator.

$$\begin{aligned}
&\langle put\rangle\ x\ (\langle ret\rangle\ w\ \langle [] \rangle\ \langle get\rangle\ \langle ret\rangle)\ \langle \gg\!= \rangle\ \lambda z \to \langle put\rangle\ y\ (\langle ret\rangle\ z)\\
=&\quad \{\text{ definition of } (\langle \gg\!= \rangle)\ \}\\
&\langle put\rangle\ x\ (\langle put\rangle\ y\ (\langle ret\rangle\ w)\ \langle [] \rangle\ \langle get\rangle\ (\lambda s \to \langle put\rangle\ y\ (\langle ret\rangle\ s)))\\
=&\quad \{\text{ by (25) and (19) }\}\\
&\langle put\rangle\ y\ (\langle ret\rangle\ w\ \langle [] \rangle\ \langle get\rangle\ (\lambda s \to \langle put\rangle\ y\ (\langle ret\rangle\ s)))\\
=&\quad \{\text{ by (27) }\}\\
&\langle put\rangle\ y\ (\langle ret\rangle\ w)\ \langle [] \rangle\ \langle put\rangle\ y\ (\langle get\rangle\ (\lambda s \to \langle put\rangle\ y\ (\langle ret\rangle\ s)))\\
=&\quad \{\text{ by (20) and (19) }\}\\
&\langle put\rangle\ y\ (\langle ret\rangle\ w)\ \langle [] \rangle\ \langle put\rangle\ y\ (\langle ret\rangle\ y)\\
=&\quad \{\text{ by (27) }\}\\
&\langle put\rangle\ y\ (\langle ret\rangle\ w\ \langle [] \rangle\ \langle ret\rangle\ y)
\end{aligned}$$

(a)

$$\begin{aligned}
&(\langle put\rangle\ x\ (\langle ret\rangle\ w)\ \langle [] \rangle\ \langle put\rangle\ x\ (\langle get\rangle\ \langle ret\rangle))\\
&\quad \langle \gg\!= \rangle\ \lambda z \to \langle put\rangle\ y\ (\langle ret\rangle\ z)\\
=&\quad \{\text{ definition of } (\langle \gg\!= \rangle)\ \}\\
&\langle put\rangle\ x\ (\langle put\rangle\ y\ (\langle ret\rangle\ w))\ \langle [] \rangle\ \langle put\rangle\ x\ (\langle get\rangle\ (\lambda s \to \langle put\rangle\ y\ (\langle ret\rangle\ s)))\\
=&\quad \{\text{ by (19) and (20) }\}\\
&\langle put\rangle\ y\ (\langle ret\rangle\ w)\ \langle [] \rangle\ \langle put\rangle\ x\ (\langle put\rangle\ y\ (\langle ret\rangle\ x))\\
=&\quad \{\text{ by (19) }\}\\
&\langle put\rangle\ y\ (\langle ret\rangle\ w)\ \langle [] \rangle\ \langle put\rangle\ y\ (\langle ret\rangle\ x)\\
=&\quad \{\text{ by (27) }\}\\
&\langle put\rangle\ y\ (\langle ret\rangle\ w\ \langle [] \rangle\ \langle ret\rangle\ x)
\end{aligned}$$

(b)

Fig. 5: Proof that law (27) implies that a bind operator cannot exist for the semantic domain.

7.3 Contextual Equivalence

With our semantic domain sufficiently specified, we can prove analogous properties for programs interpreted through this domain. We must take care in how we reformulate these properties however. It is certainly not sufficient to merely copy the laws as formulated for the semantic domain, substituting Prog data constructors for semantic domain operators as needed; we must keep in mind that a term in Prog a describes a syntactical structure without ascribing meaning to it. For example, one cannot simply assert that Put x (Put y p) is *equal* to Put y p, because although these two programs have the same semantics, they are not structurally identical. It is clear that we must define a notion of "semantical equivalence" between programs. We can map the syntactical structures in Prog a onto the semantic domain Dom a using *run* to achieve that. Yet wrapping both sides of an equation in *run* applications is not enough as such statements only apply at the top-level of a program. For instance, while *run* (Put x (Put y p)) = *run* (Put y p) is a correct statement, we cannot prove *run* (Return w [] Put x (Put y p)) = *run* (Return w [] Put y p) from such a law.

So the concept of semantical equivalence in itself is not sufficient; we require a notion of "contextual semantic equivalence" of programs which allows us to formulate properties about semantical equivalence which hold in any surrounding context. Figure 4(c) provides the definition for single-hole contexts Ctx. A context C of type Ctx e_1 a e_2 b can be interpreted as a function that, given a program that returns a value of type a under environment e_1 (in other words: the type and environment of the hole), produces a program that returns a value of type b under environment e_2 (the type and environment of the whole program). Filling the hole with p is denoted by C[p]. The type of environments, Env is defined using heterogeneous lists (Figure 4(b)). When we consider the notion of programs in contexts, we must take into account that these contexts may introduce variables which are referenced by the program. The Prog datatype however represents only closed programs. Figure 4(b) introduces the OProg type to represent "open" programs, and the Env type to represent environments. OProg e a is defined as the type of functions that construct a *closed* program of type $\mathsf{Prog}\ a$, given an environment of type $\mathsf{Env}\ e$. Environments, in turn, are defined as heterogeneous lists. We also define a function for mapping open programs onto the semantic domain.

$$\begin{array}{l} \mathit{orun} :: \mathsf{OProg}\ e\ a \to \mathsf{Env}\ e \to \mathsf{Dom}\ a \\ \mathit{orun}\ p\ \mathit{env} = \mathit{run}\ (p\ \mathit{env}) \quad . \end{array}$$

We can then assert that two programs are contextually equivalent if, for *any* context, running both programs wrapped in that context will yield the same result:

$$m_1 =_{\mathsf{GS}} m_2 \triangleq \forall C . \mathit{orun}\ (\mathsf{C}[m_1]) = \mathit{orun}\ (\mathsf{C}[m_2]) \quad .$$

We can then straightforwardly formulate variants of the state laws, the nondeterminism laws and the *put-or* law for this global state monad as lemmas. For example, we reformulate law (19) as

$$\mathsf{Put}\ s\ (\mathsf{Put}\ t\ p) =_{\mathsf{GS}} \mathsf{Put}\ t\ p \quad .$$

Proofs for the state laws, the nondeterminism laws and the *put-or* law then easily follow from the analogous semantic domain laws. The formulation of a *put-ret-or* law-like property (27) requires somewhat more care: because there exists a bind operator for Prog, we must stipulate that it does not hold in arbitrary contexts:

$$\mathit{run}\ (\mathsf{Put}\ s\ (\mathsf{Return}\ x \mathbin{[\!]} p)) = \mathit{run}\ (\mathsf{Put}\ s\ (\mathsf{Return}\ x) \mathbin{[\!]} \mathsf{Put}\ s\ p) \quad .\checkmark \tag{28}$$

The proof of this statement has been machine-verified in Coq. We annotate theorems which have been verified in Coq with a ✓.

7.4 Simulating Local-State Semantics

We simulate local-state semantics by replacing each occurrence of Put by a variant that restores the state, as described in Section 6.3. This transformation is implemented by the function *trans* for closed programs, and *otrans* for open programs:

$$\begin{array}{lll} \mathit{trans} & :: \mathsf{Prog}\ a \to \mathsf{Prog}\ a & \\ \mathit{trans}\ (\mathsf{Return}\ x) & = \mathsf{Return}\ x & \\ \mathit{trans}\ (p \mathbin{[\!]} q) & = \mathit{trans}\ p \mathbin{[\!]} \mathit{trans}\ q & \\ \mathit{trans}\ \emptyset & = \emptyset & \\ \mathit{trans}\ (\mathsf{Get}\ p) & = \mathsf{Get}\ (\lambda s \to \mathit{trans}\ (p\ s)) & \\ \mathit{trans}\ (\mathsf{Put}\ s\ p) & = \mathsf{Get}\ (\lambda s' \to \mathsf{Put}\ s\ (\mathit{trans}\ p) \mathbin{[\!]} \mathsf{Put}\ s'\ \emptyset) & , \\ \mathit{otrans} & :: \mathsf{OProg}\ e\ a \to \mathsf{OProg}\ e\ a & \\ \mathit{otrans}\ p & = \lambda \mathit{env} \to \mathit{trans}\ (p\ \mathit{env}) & . \end{array}$$

We then define the function *eval*, which runs a transformed program (in other words, it runs a program with local-state semantics).

$$\begin{aligned}&\mathit{eval} :: \mathsf{Prog}\ a \rightarrow \mathsf{Dom}\ a\\&\mathit{eval} = \mathit{run} \cdot \mathit{trans}\quad.\end{aligned}$$

We show that the transformation works by proving that our free monad equipped with *eval* is a correct implementation for a nondeterministic, stateful monad with local-state semantics. We introduce notation for "contextual equivalence under simulated backtracking semantics":

$$m_1 =_{\mathsf{LS}} m_2 \triangleq \forall C.\mathit{eval}\ (\mathsf{C}[m_1]) = \mathit{eval}\ (\mathsf{C}[m_2])\quad.$$

For example, we formulate the statement that the *put-put* law (19) holds for our monad as interpreted by *eval* as

$$\mathsf{Put}\ s\ (\mathsf{Put}\ t\ p) =_{\mathsf{LS}} \mathsf{Put}\ t\ p\quad.\checkmark$$

Proofs for the nondeterminism laws follow trivially from the nondeterminism laws for global state. The state laws are proven by promoting *trans* inside, then applying global-state laws. For the proof of the *get-put* law, we require the property that in global-state semantics, Put distributes over $(\|)$ if the left branch has been transformed (in which case the left branch leaves the state unmodified). This property only holds at the top-level.

$$\mathit{run}\ (\mathsf{Put}\ x\ (\mathit{trans}\ m_1 \mathbin{\|} m_2)) = \mathit{run}\ (\mathsf{Put}\ x\ (\mathit{trans}\ m_1) \mathbin{\|} \mathsf{Put}\ x\ m_2)\quad.\checkmark \tag{29}$$

Proof of this lemma depends on law (27).

Finally, we arrive at the core of our proof: to show that the interaction of state and nondeterminism in this implementation produces backtracking semantics. To this end we prove laws analogous to the local state laws (15) and (16)

$$m \gg \emptyset =_{\mathsf{LS}} \emptyset\quad,\checkmark \tag{30}$$

$$m \gg\!= (\lambda x \rightarrow f_1\ x \mathbin{\|} f_2\ x) =_{\mathsf{LS}} (m \gg\!= f_1) \mathbin{\|} (m \gg\!= f_2)\quad.\checkmark \tag{31}$$

We provide machine-verified proofs for these theorems.

The proof for (30) follows by straightforward induction.

The inductive proof (with induction on m) of law (31) requires some additional lemmas.

For the case $m = m_1 \mathbin{\|} m_2$, we require the property that, at the top-level of a global-state program, $(\|)$ is commutative if both its operands are state-restoring. Formally:

$$\mathit{run}\ (\mathit{trans}\ p \mathbin{\|} \mathit{trans}\ q) = \mathit{run}\ (\mathit{trans}\ q \mathbin{\|} \mathit{trans}\ p)\quad.\checkmark \tag{32}$$

The proof of this property motivated the introduction of law (26).

The proof for both the $m = \mathsf{Get}\ k$ and $m = \mathsf{Put}\ s\ m'$ cases requires that Get distributes over $(\|)$ at the top-level of a global-state program if the left branch is state restoring.

$$\mathit{run}\ (\mathsf{Get}\ (\lambda s \rightarrow \mathit{trans}\ (m_1\ s) \mathbin{\|} (m_2\ s))) = \mathit{run}\ (\mathsf{Get}\ (\lambda s \rightarrow \mathit{trans}\ (m_1\ s)) \mathbin{\|} \mathsf{Get}\ m_2)\quad.\checkmark \tag{33}$$

And finally, we require that the *trans* function is, semantically speaking, idempotent, to prove the case $m = \mathsf{Put}\ s\ m'$.

$$run\ (trans\ (trans\ p)) = run\ (trans\ p)\quad .✓ \tag{34}$$

7.5 Backtracking with a Global State Monad

There is still one technical detail to to deal with before we deliver a backtracking algorithm that uses a global state. As mentioned in Section 6.2, rather than using *put*, some algorithms typically use a pair of commands *modify next* and *modify prev*, with $prev \cdot next = id$, to respectively update and roll back the state. This is especially true when the state is implemented using an array or other data structure that is usually not overwritten in its entirety. Following a style similar to put_{R}, this can be modelled by:

$$modify_{\mathsf{R}} :: \{\mathsf{N}, \mathsf{S}\ s\} \subseteq \epsilon \to (s \to s) \to (s \to s) \to \mathsf{M}_\epsilon\ ()$$
$$modify_{\mathsf{R}}\ next\ prev = modify\ next\ [\!]\ side\ (modify\ prev)\quad .$$

Is it safe to use an alternative translation, where the pattern $get \gg\!= (\lambda s \to put\ (next\ s) \gg m)$ is not translated into $get \gg\!= (\lambda s \to put_{\mathsf{R}}\ (next\ s) \gg trans\ m)$, but rather into $modify\_R\ next\ prev \gg trans\ m$? We explore this question by extending our Prog syntax with an additional Modify_R construct, thus obtaining a new Prog_m syntax:

```
data Prog_m a where
 ...
   Modify_R :: (S → S) → (S → S) → Prog_m a → Prog_m a
```

We assume that $prev \cdot next = id$ for every Modify_R *next prev p* in a Prog_m *a* program.

We then define two translation functions from Prog_m *a* to Prog *a*, which both replace Puts with put_{R}s along the way, like the regular *trans* function. The first replaces each Modify_R in the program by a direct analogue of the definition given above, while the second replaces it by Get $(\lambda s \to \mathsf{Put}\ (next\ s)\ (trans\_2\ p))$:

```
trans_1 :: Prog_m a → Prog a
 ...
trans_1 (Modify_R next prev p) = Get (λs → Put (next s) (trans_1 p) [] Get (λt → Put (prev t) ∅))
trans_2 :: Prog_m a → Prog a
 ...
trans_2 (Modify_R next prev p) = Get (λs → put_R (next s) (trans_2 p))
   where put_R s p = Get (λt → Put s p [] Put t ∅)
```

It is clear that *trans_2 p* is the exact same program as *trans p'*, where p' is p but with each ModifyR *next prev p* replaced by Get $(\lambda s \to \mathsf{Put}\ (next\ s)\ p)$.

We then prove that these two transformations lead to semantically identical instances of Prog *a*.

Lemma 2. *run (trans_1 p) = run (trans_2 p).* ✓

Backtracking using a global-state monad Recall that, in Section 5.2, we showed that a problem formulated by $\mathit{unfoldM}\ p\ f\ z \mathbin{>\!\!>\!\!=} \mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ \mathit{st})$ can be solved by a hylomorphism $\mathit{solve}\ p\ f\ \mathit{ok}\ (\oplus)\ \mathit{st}\ z$, run in a non-deterministic local-state monad. Putting all the information in this section together, we conclude that solutions of the same problem can be computed, in a non-deterministic global-state monad, by $\mathit{run}\ (\mathit{solve}_{\mathsf{R}}\ p\ f\ \mathit{ok}\ (\oplus)\ (\ominus)\ \mathit{st}\ z)$, where $(\ominus x) \cdot (\oplus x) = \mathit{id}$ for all x, and $\mathit{solve}_{\mathsf{R}}$ is defined by:

$$\begin{aligned}
&\mathit{solve}_{\mathsf{R}} :: \{\mathsf{N}, \mathsf{S}\ s\} \subseteq \epsilon \Rightarrow (b \to \mathsf{Bool}) \to (b \to \mathsf{M}_{\epsilon}\ (a, b)) \to \\
&\qquad\qquad (s \to \mathsf{Bool}) \to (s \to a \to s) \to (s \to a \to s) \to s \to b \to \mathsf{M}_{\epsilon}\ [a] \\
&\mathit{solve}_{\mathsf{R}}\ p\ f\ \mathit{ok}\ (\oplus)\ (\ominus)\ \mathit{st}\ z = \mathit{put}_{\mathsf{R}}\ \mathit{st} \gg \mathit{hyloM}\ (\odot)\ (\mathit{return}\ [\,])\ p\ f\ z \\
&\quad \textbf{where}\ x \odot m = (\mathit{get} \mathbin{>\!\!>\!\!=} \mathit{guard} \cdot \mathit{ok} \cdot (\oplus x)) \gg \\
&\qquad\qquad\qquad \mathit{modify}_{\mathsf{R}}\ (\oplus x)\ (\ominus x) \gg ((x{:})\ \langle\$\rangle\ m) \quad .
\end{aligned}$$

Note that the use of *run* enforces that the program is run as a whole, that is, it cannot be further composed with other monadic programs.

n-Queens using a global state To wrap up, we revisit the n-queens puzzle. Recall that, for the puzzle, $(i, \mathit{us}, \mathit{ds}) \oplus x = (1 + i, (i + x) : \mathit{us}, (i - x) : \mathit{ds})$. By defining $(i, \mathit{us}, \mathit{ds}) \ominus x = (i - 1, \mathit{tail}\ \mathit{us}, \mathit{tail}\ \mathit{ds})$, we have $(\ominus x) \cdot (\oplus x) = \mathit{id}$. One may thus compute all solutions to the puzzle, in a scenario with a shared global state, by $\mathit{run}\ (\mathit{queens}_{\mathsf{R}}\ n)$, where *queens* expands to:

$$\begin{aligned}
&\mathit{queens}_{\mathsf{R}}\ n = \mathit{put}\ (0, [\,], [\,]) \gg \mathit{queensBody}\ [0\,..\,n-1] \quad , \\
&\mathit{queensBody}\ [\,]\ = \mathit{return}\ [\,] \\
&\mathit{queensBody}\ \mathit{xs} = \mathit{select}\ \mathit{xs} \mathbin{>\!\!>\!\!=} \lambda(x, \mathit{ys}) \to \\
&\qquad\qquad (\mathit{get} \mathbin{>\!\!>\!\!=} (\mathit{guard} \cdot \mathit{ok} \cdot (\oplus x))) \gg \\
&\qquad\qquad \mathit{modify}_{\mathsf{R}}\ (\oplus x)\ (\ominus x) \gg ((x{:})\ \langle\$\rangle\ \mathit{queensBody}\ \mathit{ys}) \quad , \\
&\quad \textbf{where}\ (i, \mathit{us}, \mathit{ds}) \oplus x = (1 + i, (i + x) : \mathit{us}, (i - x) : \mathit{ds}) \\
&\qquad\quad (i, \mathit{us}, \mathit{ds}) \ominus x = (i - 1, \mathit{tail}\ \mathit{us}, \quad \mathit{tail}\ \mathit{ds}) \\
&\qquad\quad \mathit{ok}\ (\_, u : \mathit{us}, d : \mathit{ds}) = (u \notin \mathit{us}) \wedge (d \notin \mathit{ds}) \quad .
\end{aligned}$$

8 Conclusions and Related Work

This paper started as a case study of reasoning and derivation of monadic programs. To study the interaction between non-determinism and state, we construct backtracking algorithms solving problems that can be specified in the form $\mathit{unfoldM}\ f\ p \mathbin{>\!\!>\!\!>\!\!=} \mathit{assert}\ (\mathit{all}\ \mathit{ok} \cdot \mathit{scanl}_{+}\ (\oplus)\ \mathit{st})$, for two scenarios. In the first scenario, we assume that right-distributivity and right-zero laws hold, which imply that each non-deterministic branch has its own state. The derivation of the backtracking algorithm works by fusing the two phases into a monadic hylomorphism.

In the second scenario we consider the case when the state is global. We find that we may use $(\talloblong)$ to simulate sequencing, and that the idea can be elegantly packaged into commands like $\mathit{put}_{\mathsf{R}}$ and $\mathit{modify}_{\mathsf{R}}$. The interaction between global state and non-determinism turns out to be rather tricky. For a more rigorous treatment, we enforce a more precise separation between syntax and semantics and, as a side contribution of this paper, propose a collection of *global state laws* which the semantics should satisfy, and verified in Coq that there is an implementation satisfying these laws. With the setting up, we show that a program written for local state works for the global state scenario if we replace all occurrences of *put* by put_{R}.

It turns out that in derivations of programs using non-determinism and state, commutativity plays an important role. When the state is local, we have nicer properties at hand, and commutativity holds more generally. With a shared global state, commutativity holds in limited cases. In particular, put_{R} still commutes with non-determinism.

8.1 Related Work

Prolog Four-Port Box Model [**?**] applied an idea, similar to put_{R}, to implement debugging for the *4-port box model* of Prolog. In this model, upon the first entrance of a Prolog procedure it is *called*; it may yield a result and *exits*; when the subsequent procedure fails and backtracks, it is asked to *redo* its computation, possibly yielding the next result; finally it may *fail*. Given a Prolog procedure p implemented in Haskell, the following program prints debugging messages when each of the four ports are used:

$$\begin{array}{l}(putStr\ \texttt{"call"}\ [\!]\ side\ (putStr\ \texttt{"fail"})) \gg \\ p \gg\!= \lambda x \rightarrow \\ (putStr\ \texttt{"exit"}\ [\!]\ side\ (putStr\ \texttt{"redo"})) \gg return\ x \quad .\end{array}$$

Local Algebraic Effect Theories In this paper, we have used two different techniques to distinguish between effect operators from their implementations: type classes and free monads. In both cases, the meaning of the effect operators is given by a set of externally applied axioms. [**?**] explore another approach using algebraic effects and handlers. In their approach, axioms (or "effect theories") are encoded in the type system: the type of an effectful function declares the operators used in the function, as well as the equalities that handlers for these operators should comply with. The type of a handler indicates which operators it handles and which equations it complies with.

Acknowledgements to be added.

A An Implementation of the Semantical Domain

We present an implementation of Dom that satisfies the axioms demanded by Section 7. We have proven its compliance to the axioms laid out in Section 7.2 by writing a machine-verified proof (✓). We let Dom be the union of $\mathsf{M}\ s\ a$ for all a and for a given s.

The implementation is based on a multiset or Bag data structure.

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Bag}\ a\\
&\mathit{singleton}\ :: a \to \mathsf{Bag}\ a\\
&\mathit{emptyBag} :: \mathsf{Bag}\ a\\
&\mathit{sum}\ \ \ \ \ \ \ \ :: \mathsf{Bag}\ a \to \mathsf{Bag}\ a \to \mathsf{Bag}\ a
\end{aligned}
$$

We model a stateful, nondeterministic computation with global state semantics as a function that maps an initial state onto a bag of results, and a final state. Each result is a pair of the value returned, as well as the state at that point in the computation. As we mentioned in Section 7.2, a bind operator cannot be defined for this implementation (and this is by design), because we retain only the final result of the branch without any information on how to continue the branch.

$$\textbf{type}\ \mathsf{M}\ s\ a = s \to (\mathsf{Bag}\ (a, s), s)$$

$\langle\varnothing\rangle$ does not modify the state and produces no results. $\langle \mathit{ret}\rangle$ does not modify the state and produces a single result.

$$
\begin{aligned}
&\langle\varnothing\rangle :: \mathsf{M}\ s\ a\\
&\langle\varnothing\rangle = \lambda s \to (\mathit{emptyBag}, s)\\
&\langle \mathit{ret}\rangle :: a \to \mathsf{M}\ s\ a\\
&\langle \mathit{ret}\rangle\ x = \lambda s \to (\mathit{singleton}\ (x, s), s)
\end{aligned}
$$

$\langle \mathit{get}\rangle$ simply passes along the initial state to its continuation. $\langle \mathit{put}\rangle$ ignores the initial state and calls its continuation with the given parameter instead.

$$
\begin{aligned}
&\langle \mathit{get}\rangle :: (s \to \mathsf{M}\ s\ a) \to \mathsf{M}\ s\ a\\
&\langle \mathit{get}\rangle\ k = \lambda s \to k\ s\ s\\
&\langle \mathit{put}\rangle :: s \to \mathsf{M}\ s\ a \to \mathsf{M}\ s\ a\\
&\langle \mathit{put}\rangle\ s\ k = \lambda\_ \to k\ s
\end{aligned}
$$

The $\langle [\!] \rangle$ operator runs the left computation with the initial state, then runs the right computation with the final state of the left computation, and obtains the final result by merging the two bags of results.

$$
\begin{array}{l}
(\langle\|\rangle) :: \mathsf{M}\ s\ a \rightarrow \mathsf{M}\ s\ a \rightarrow \mathsf{M}\ s\ a \\
(xs\ \langle\|\rangle\ ys)\ s = \mathbf{let}\ (ansx, s')\ \ = xs\ s \\
\qquad\qquad\qquad\qquad\quad (ansy, s'') = ys\ s' \\
\qquad\qquad\qquad\quad \mathbf{in}\ (sum\ ansx\ ansy, s'')
\end{array}
$$