Handling Local State with Global State

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

[1] Department of Computer Science, KU Leuven, Belgium,
first.last@cs.kuleuven.be
[2] Institute of Information Science, Academia Sinica, Taiwan,
scm@iis.sinica.edu.tw

Abstract. Equational reasoning is one of the most important tools of functional programming. To facilitate its application to monadic programs, Gibbons and Hinze have proposed a simple axiomatic approach using laws that characterise the computational effects without exposing their implementation details. At the same time Plotkin and Pretnar have proposed algebraic effects and handlers, a mechanism of layered abstractions by which effects can be implemented in terms of other effects.
This paper performs a case study that connects these two strands of research. We consider two ways in which the nondeterminism and state effects can interact: the high-level semantics where every nondeterministic branch has a local copy of the state, and the low-level semantics where a single sequentially threaded state is global to all branches.
We give a monadic account of the folklore technique of handling local state in terms of global state, provide a novel axiomatic characterisation of global state and prove that the handler satisfies Gibbons and Hinze's local state axioms by means of a novel combination of free monads and contextual equivalence. We also provide a model for global state that is necessarily non-monadic.

Keywords: monads · effect handlers · equational reasoning · nondeterminism · state · contextual equivalence

1 Introduction

Monads have been introduced to functional programming to support side effects in a rigorous, mathematically manageable manner [11,14]. Over time they have become the main framework in which effects are modelled. Various monads were developed for different effects, from general ones such as IO, state, nondeterminism, exception, continuation, environment passing, to specific purposes such as parsing. Much research was also devoted to producing practical monadic programs.

Equational reasoning about pure functional programs is particularly simple and powerful. Yet, Hutton and Fulger [7] noted that a lot less attention has been paid to reasoning about monadic programs in that style. Gibbons and Hinze [4] argue that equational reasoning about monadic programs becomes particularly

convenient and elegant when one respects the abstraction boundaries of the monad. This is possible by reasoning in terms of axioms or laws that characterise the monad's behavior without fixing its implementation.

This paper is a case study of equational reasoning with monadic programs. Following the approach of algebraic effects and handlers [12], we consider how one monad can be implemented in terms of another—or, in other words, how one can be simulated in by another using a careful discipline. Our core contribution is a novel approach for proving the correctness of such a simulation. The proof approach is a convenient hybrid between equational reasoning based on axioms and inductive reasoning on the structure of programs. To capture the simulation we apply the algebraic effects technique of *handling* a free monad representation [15]. The latter provides a syntax tree on which to perform induction. To capture the careful discipline of the simulation we use contextual equivalence and perform inductive reasoning about program contexts. This allows us to deal with a heterogeneous axiom set where different axioms may make use of different notions of equality for programs.

We apply this proof technique to a situation where each "monad" (both the simulating monad and the simulated monad) is in fact a combination of two monads, with differing laws on how these effects interact: non-determinism and state.

In the monad we want to implement, each non-deterministic branch has its own 'local' copy of the state. This is a convenient effect interaction which is provided by many systems that solve search problems, including Prolog. A characterisation of this 'local state' monad was given by Gibbons and Hinze [4].

We realise this local state semantics in terms of a more primitive monad where a single state is sequentially threaded through the non-deterministic branches. Because this state is shared among the branches, we call this the 'global state' semantics. The appearance of local state is obtained by following a discipline of undoing changes to the state when backtracking to the next branch. This folklore backtracking technique is implemented by most sequential search systems because of its relative efficiency: remembering what to undo often requires less memory than creating multiple copies of the state, and undoing changes often takes less time than recomputing the state from scratch. To the best of our knowledge, our axiomatic characterisation of the global state monad is novel.

In brief, our contributions can be summarized as follows:

- We provide an axiomatic characterisation for the interaction between the monadic effects of non-determinism and state where the state is persistent (i.e., does not backtrack), together with a model that satisfies this characterisation.
- We prove that—with a careful discipline—our characterisation of persistent state can correctly simulate Gibbons and Hinze's monadic characterisation of backtrackable state [4]. We use our novel proof approach (the core contribution of this paper) to do so.
- Our proof also comes with a mechanization in Coq.[3]

[3] The proof can be found at `https://github.com/KoenP/LocalAsGlobal`.

The rest of the paper is structured as follows. First, Section 2 gives an overview of the main concepts used in the paper and defines our terminology. Then, Section 3 informally explores the differences between local and global state semantics. Next, Section 4 explains how to handle local state in terms of global state. Section 5 formalizes this approach and proves it correct. Finally, Sections 6 and 7 respectively discuss related work and conclude.

2 Background

This section briefly summarises the main concepts we need for equational reasoning about effects. For a more extensive treatment we refer to the work of Gibbons and Hinze [4].

2.1 Monads, Nondeterminism and State

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

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

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

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

Nondeterminism The first effect we introduce is nondeterminism. Following the trail of Hutton and Fulger [7] and Gibbons and Hinze, we introduce effects based on an axiomatic characterisation rather than a specific implementation. We define a type class to capture this interface as follows:

$$\begin{array}{l}\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MNondet}\ m\ \textbf{where} \\ \quad \emptyset \quad :: m\ a \\ \quad (\mathbin{[\!]}) :: m\ a \to m\ a \to m\ a \quad .\end{array}$$

In this interface, $\emptyset$ denotes failure, while $m \mathbin{[\!]} n$ denotes that the computation may yield either m or n. Precisely what laws these operators should satisfy, however, can be a tricky issue. As discussed by Kiselyov [8], it eventually comes down to what we use the monad for.

It is usually expected that $(\mathbin{[\!]})$ and $\emptyset$ form a monoid. That is, $(\mathbin{[\!]})$ is associative, with $\emptyset$ as its zero:

$$(m \mathbin{[\!]} n) \mathbin{[\!]} k \ = \ m \mathbin{[\!]} (n \mathbin{[\!]} k) \quad , \tag{4}$$

$$\emptyset \mathbin{[\!]} m \ = \ m \ = \ m \mathbin{[\!]} \emptyset \quad . \tag{5}$$

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

$$\textbf{left-distributivity}: \quad (m_1 \mathbin{[\!]} m_2) \gg\!= f \ = \ (m_1 \gg\!= f) \mathbin{[\!]} (m_2 \gg\!= f) \quad , \tag{6}$$

$$\textbf{left-zero}: \qquad \emptyset \gg\!= f \;=\; \emptyset \quad . \tag{7}$$

We will refer to the laws (4), (5), (6), (7) collectively as the *nondeterminism laws*.

One might might intuitively expect some additional laws from a set of nondeterminism operators, such as idempotence ($p \mathbin{[\!]} p = p$) or commutativity ($p \mathbin{[\!]} q = q \mathbin{[\!]} p$). However, our primary interest lies in the study of combinations of effects and – as we shall see very soon – in particular the combination of nondeterminism with *state*. One of our characterisations of this interaction would be incompatible with both idempotence and commutativity, at least if they are stated as strongly as we have done here. We will eventually introduce a weaker notion of commutativity, but it would not be instructive to do so here (as its properties would be difficult to motivate at this point).

State The state effect provides two operators:

$$\begin{aligned}&\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MState}\ s\ m \mid m \to s\ \textbf{where}\\ &\quad \mathit{get} :: m\ s\\ &\quad \mathit{put} :: s \to m\ ()\quad .\end{aligned}$$

The *get* operator retrieves the state, while *put* overwrites the state by the given value. They satisfy the *state laws*:

$$\textbf{put-put}: \qquad \mathit{put}\ st \gg \mathit{put}\ st' = \mathit{put}\ st' \quad , \tag{8}$$

$$\textbf{put-get}: \qquad \mathit{put}\ st \gg \mathit{get} = \mathit{put}\ st \gg \mathit{return}\ st \quad , \tag{9}$$

$$\textbf{get-put}: \qquad \mathit{get} \gg\!= \mathit{put} = \mathit{return}\ () \quad , \tag{10}$$

$$\textbf{get-get}: \qquad \mathit{get} \gg\!= (\lambda st \to \mathit{get} \gg\!= k\ st) = \mathit{get} \gg\!= (\lambda st \to k\ st\ st) \quad , \tag{11}$$

where $m_1 \gg m_2 = m_1 \gg\!= \lambda\_ \to m_2$, which has type $(\gg) :: m\ a \to m\ b \to m\ b$.

2.2 Combining Effects

As Gibbons and Hinze already noted, an advantage of defining our effects axiomatically, rather than by providing some concrete implementation, is that it is straightforward to reason about combinations of effects. In this paper, we are interested in the interaction between nondeterminism and state, specifically.

$$\textbf{class}\ (\mathsf{MState}\ s\ m, \mathsf{MNondet}\ m) \Rightarrow \mathsf{MStateNondet}\ s\ m \mid m \to s \quad .$$

The type class $\mathsf{MStateNondet}\ s\ m$ simply inherits the operators of its superclasses $\mathsf{MState}\ s\ m$ and $\mathsf{MNondet}\ m$ without adding new operators, and implementations of this class should comply with all laws of both superclasses.

However, this is not the entire story. Without additional 'interaction laws', the design space for implementations of this type class is left wide-open with respect to questions about how these effects interact. In particular, it seems hard to imagine that one could write nontrivial programs which are agnostic towards the question of what happens to the state of the program when the program backtracks. We discuss two possible approaches.

Local State Semantics One is what Gibbons and Hinze call "backtrackable state", that is, when a branch of the nondeterministic computation runs into a dead end and the continuation of the computation is picked up at the most recent branching point, any alterations made to the state by our terminated branch are invisible to the continuation. Because in this scheme state is local to a branch, we will refer to these semantics as *local state semantics*. We characterise local state semantics with the following laws:

$$\textbf{right-zero}: \qquad m \gg \emptyset \;=\; \emptyset \quad , \tag{12}$$

$$\textbf{right-distributivity}: \quad m \gg\!= (\lambda x \to f_1\ x \,[]\, f_2\ x) = (m \gg\!= f_1) \,[]\, (m \gg\!= f_2). \tag{13}$$

With some implementations of the monad, it is likely that in the lefthand side of (13), the effect of m happens once, while in the righthand side it happens twice. In (12), the m on the lefthand side may incur some effects that do not happen in the righthand side.

Having (12) and (13) leads to profound consequences on the semantics and implementation of monadic programs. To begin with, (13) implies that for $([])$ we have some limited notion of commutativity. For instance, both the left and right distributivity rules can be applied to the term $(\mathit{return}\ x \,[]\, \mathit{return}\ y) \gg\!= \lambda z \to \mathit{return}\ z \,[]\, \mathit{return}\ z$. It is then easy to show that this term must be equal to both $\mathit{return}\ x \,[]\, \mathit{return}\ x \,[]\, \mathit{return}\ y \,[]\, \mathit{return}\ y$ and $\mathit{return}\ x \,[]\, \mathit{return}\ y \,[]\, \mathit{return}\ x \,[]\, \mathit{return}\ y$.[4]

In fact, having (12) and (13) gives us very strong and useful commutative properties. To be clear what we mean, we give a formal definition:

Definition 1. *Let m and n be two monadic programs such that x does not occur free in m, and y does not occur free in n. We say m and n commute if*

$$\begin{aligned} m \gg\!= \lambda x \to n \gg\!= \lambda y \to f\ x\ y \;\; &= \\ n \gg\!= \lambda y \to m \gg\!= \lambda x \to f\ x\ y \;\; &. \end{aligned} \tag{14}$$

We say that effects ϵ and δ commute if any m and n commute as long as their only effects are respectively ϵ and δ.

One important result is that, in local state semantics, non-determinism commutes with any effect :

Theorem 1. *If right-zero* (12) *and right-distributivity* (13) *hold in addition to the other laws, non-determinism commutes with any effect.*

Implementation-wise, (12) and (13) imply that each nondeterministic branch has its own copy of the state. To see that, let $m = \mathit{put}\ 1$, $f_1\ () = \mathit{put}\ 2$, and $f_2\ () = \mathit{get}$ in (13) — the state we *get* in the second branch does not change, despite the *put* 2 in the first branch. One implementation satisfying the laws is

[4] Gibbons and Hinze [4] were mistaken in their claim that the type $s \to [(a, s)]$ constitutes a model of their backtrackable state laws; it is not a model because its $([])$ does not commute with itself. One could consider a relaxed semantics that admits $s \to [(a, s)]$, but that is not the focus of this paper.

$\mathsf{M}\ s\ a = s \to \mathsf{Bag}\ (a, s)$, where $\mathsf{Bag}\ a$ is an implementation of a multiset or "bag" data structure. If we ignore the unordered nature of the Bag type, this implementation is similar to $\mathsf{StateT}\ s\ (\mathsf{ListT}\ \mathsf{Identity})$ in the Monad Transformer Library [5]. With effect handling [15,9], the monad behaves similarly (except for the limited commutativity implied by law (13)) if we run the handler for state before that for list.

Global State Semantics Alternatively, we can choose a semantics where state reigns over nondeterminism. In this case of non-backtrackable state, alterations to the state persist over backtracks. Because only a single state is shared over all the branches of the nondeterministic computation, we call this semantics *global state semantics*. We will return later to the question of how to define laws that capture our intuition for this kind of semantics, because (to the best of our knowledge) this constitutes a novel contribution.

Even just figuring out an implementation of a global state monad that matches our intuition is already tricky. One might believe that $\mathsf{M}\ s\ a = s \to ([a], s)$ is a natural implementation of such a monad. The usual, naive implementation of $(\gg\!=)$ using this representation, however, does not satisfy left-distributivity (6), violates monad laws, and is therefore not even a monad. The type $\mathsf{ListT}\ (\mathsf{State}\ s)$ generated using the Monad Transformer Library [5] expands to essentially the same implementation, and is flawed in the same way. More careful implementations of ListT, which do satisfy (6) and the monad laws, have been proposed [3,13]. Effect handlers (e.g. Wu [15] and Kiselyov and Ishii [9]) do produce implementations which match our intuition of a non-backtracking computation if we run the handler for non-determinism before that of state.

We provide a direct implementation to aid the intuition of the reader. Essentially the same implementation is obtained by using the type $\mathsf{ListT}\ (\mathsf{State}\ s)$ where ListT is implemented as a correct monad transformer. This implementation has a non-commutative $(\llbracket)$.

$$\mathsf{M}\ s\ a = s \to (\mathsf{Maybe}\ (a, \mathsf{M}\ s\ a), s) \quad .$$

The Maybe in this type indicates that a computation might fail to produce a result. But note that the s is outside of the Maybe: even if the computation fails to produce any result, a modified state may be returned (this is different from local state semantics). $\emptyset$, of course, returns an empty continuation ($\mathsf{Nothing}$) and an unmodified state. $(\llbracket)$ first exhausts the left branch (always collecting any state modifications it performs), before switching to the right branch.

$$\begin{aligned}
\emptyset &= \lambda s \to (\mathsf{Nothing}, s) \quad , \\
p \llbracket q &= \lambda s \to \mathbf{case}\ p\ s\ \mathbf{of}\ (\mathsf{Nothing}, t) \to q\ t \\
&\qquad\qquad\qquad\qquad (\mathsf{Just}\ (x, r), t) \to (\mathsf{Just}\ (x, r \llbracket q), t) \quad .
\end{aligned}$$

The state operators are implemented in a straightforward manner.

$$\begin{aligned}
get &= \lambda s \to (\mathsf{Just}\ (s\ , \emptyset), s) \quad , \\
put\ s &= \lambda t \to (\mathsf{Just}\ ((), \emptyset), s) \quad .
\end{aligned}$$

And this implementation is also a monad. The implementation of $p \gg\!= k$ extends every branch within p with k, threading the state through this entire process.

$$
\begin{aligned}
&\mathit{return}\ x = \lambda s \to (\mathsf{Just}\ (x, \emptyset), s) \quad , \\
&p \gg\!= k \quad = \lambda s \to \mathbf{case}\ p\ s\ \mathbf{of}\ (\mathsf{Nothing}, t) \quad \to (\mathsf{Nothing}, t) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{Just}\ (x, q), t) \to (k\ x\ [\!]\ (q \gg\!= k))\ t \quad .
\end{aligned}
$$

3 Motivation

In the previous section we discussed two possible semantics for the interaction of state and nondeterminism: global and local state semantics. In this section, we will further explore the differences between these two interpretations. Using the classic n-queens puzzle as an example, we show that sometimes we end up in a situation where we want to write our program according to local state semantics (which is generally speaking easier to reason about), but desire the space usage characteristics of global state semantics.

3.1 Example: The n-Queens Problem

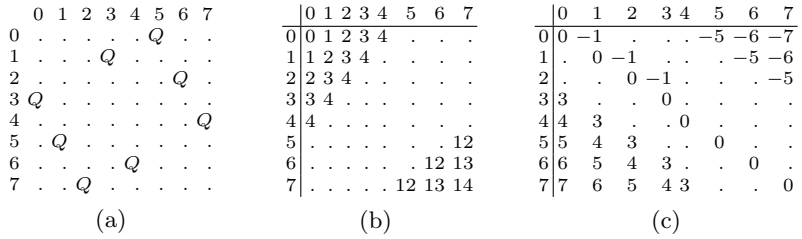

Fig. 1: (a) This placement can be represented by $[3, 5, 7, 1, 6, 0, 2, 4]$. (b) Up diagonals. (c) Down diagonals.

The n-queens puzzle presented in this section is adapted and simplified from that of Gibbons and Hinze [4]. The aim of the puzzle is to place n queens on a n by n chess board such that no two queens can attack each other. Given n, we number the rows and columns by $[0\,..\,n-1]$. Since all queens should be placed on distinct rows and distinct columns, a potential solution can be represented by a permutation xs of the list $[0\,..\,n-1]$, such that $xs\ !!\ i = j$ denotes that the queen on the ith column is placed on the jth row (see Figure 1(a)). The specification can be written as a non-deterministic program:

$$
\begin{aligned}
&\mathit{queens} :: \mathsf{MNondet}\ m \Rightarrow \mathsf{Int} \to m\ [\mathsf{Int}] \\
&\mathit{queens}\ n = \mathit{perm}\ [0\,..\,n-1] \gg\!= \mathit{filt}\ \mathit{safe} \quad ,
\end{aligned}
$$

where *perm* non-deterministically computes a permutation of its input, and the pure function $safe :: [\mathsf{Int}] \to \mathsf{Bool}$, to be defined later, determines whether a solution is valid. The function *filt* $p\ x$ returns x if $p\ x$ holds, and fails otherwise. It can be defined in terms of a standard monadic function *guard*:

$$
\begin{aligned}
&\mathit{filt} :: \mathsf{MNondet}\ m \Rightarrow (a \to \mathsf{Bool}) \to a \to m\ a \\
&\mathit{filt}\ p\ x = \mathit{guard}\ (p\ x) \gg \mathit{return}\ x \quad , \\
&\mathit{guard} :: \mathsf{MNondet}\ m \Rightarrow \mathsf{Bool} \to m\ () \\
&\mathit{guard}\ b = \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\ ()\ \mathbf{else}\ \emptyset \quad .
\end{aligned}
$$

The function *perm* can be written either as a fold or an unfold. For this problem we choose the latter, using a function *select*, which non-deterministically splits a list into a pair containing one chosen element and the rest. For example, *select* $[1,2,3]$ yields one of $(1,[2,3])$, $(2,[1,3])$ and $(3,[1,2])$.

$$
\begin{aligned}
&\mathit{select} :: \mathsf{MNondet}\ m \Rightarrow [a] \to m\ (a,[a]) \\
&\mathit{select}\ [\,] \qquad = \emptyset \\
&\mathit{select}\ (x : xs) = \mathit{return}\ (x, xs)\ [\!]\ ((id \times (x:)) \mathbin{\$\!\rangle} \mathit{select}\ xs) \quad , \\
&\mathit{perm} :: \mathsf{MNondet}\ m \Rightarrow [a] \to m\ [a] \\
&\mathit{perm}\ [\,] = \mathit{return}\ [\,] \\
&\mathit{perm}\ xs = \mathit{select}\ xs \gg\!= \lambda(x, ys) \to (x:) \mathbin{\$\!\rangle} \mathit{perm}\ ys \quad ,
\end{aligned}
$$

where $f \mathbin{\$\!\rangle} m = m \gg (\mathit{return} \cdot f)$ which applies a pure function to a monadic value, and $(f \times g)\ (x, y) = (f\ x, g\ y)$.

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, which allows branches generates a non-safe placement to be pruned earlier, and thus produce a faster implementation.

A Backtracking Algorithm In our representation, queens cannot be put on the same row or column. Therefore, *safe* only needs to make sure that no 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 1(b)). Similarly, there are 15 *down diagonals* (running between top-left and bottom right, indexed by $[-7\,..\,7]$ in Figure 1(c)). We can show, by routine program calculation, that whether a placement is safe can be checked in one left-to-right traversal — define $\mathit{safe}\ xs = \mathit{safeAcc}\ (0, [\,], [\,])\ xs$, where

$$
\begin{aligned}
&\mathit{safeAcc} :: (\mathsf{Int}, [\mathsf{Int}], [\mathsf{Int}]) \to [\mathsf{Int}] \to \mathsf{Bool} \\
&\mathit{safeAcc}\ (i, us, ds)\ [\,] \qquad = \mathsf{True} \\
&\mathit{safeAcc}\ (i, us, ds)\ (x : xs) = \mathit{ok}\ (i', us', ds') \wedge \mathit{safeAcc}\ (i', us', ds')\ xs \quad , \\
&\quad \mathbf{where}\ (i', us', ds') = (i+1, (i + x : us), (i - x : ds)) \quad , \\
&\mathit{ok}\ (i, (x : us), (y : ds)) = x \notin us \wedge y \notin ds \quad .
\end{aligned}
$$

Operationally, (i, us, ds) is a "state" kept by *safeAcc*, where i is the current column, while us and ds are respectively the up and down diagonals encountered

so far. Function *safeAcc* behaves like a fold-left. Indeed, it can be defined using *scanl* and *all* (where $all\ p = foldr\ (\wedge)\ \mathsf{True} \cdot map\ p$):

$$\begin{aligned} &safeAcc\ (i, us, ds) = all\ ok \cdot tail \cdot scanl\ (\oplus)\ (i, us, ds)\ \ , \\ &\quad \textbf{where}\ (i, us, ds) \oplus x = (i+1, (i+x : us), (i-x : ds))\ \ . \end{aligned}$$

One might wonder whether the "state" can be implemented using an actual state monad. Indeed, the following is among the theorems we have proved:

Theorem 2. *If state and non-determinism commute, we have that for all xs, st,* $(\oplus)$*, and ok,*

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

The function $protect\ m = get \gg\!= \lambda ini \to m \gg\!= \lambda x \to put\ ini \gg return\ x$ saves the initial state and restores it after the computation. As for $(\odot)$, it assumes that the "state" passed around by *scanl* is stored in a monadic state, checks whether the new state $st \oplus x$ satisfies *ok*, and updates the state with the new value.

For Theorem 2 to hold, however, we need state and non-determinism to commute. It is so in the local state semantics, which can be proved using the non-determinism laws, (12), and (13).

Now that the safety check can be performed in a *foldr*, recalling that *perm* is an unfold, it is natural to try to fuse them into one. Indeed, it can be proved that, with $(\oplus)$, *ok*, and $(\odot)$ defined above, we have $perm\ xs \gg\!= foldr\ (\odot)\ (return\ [\,]) = qBody\ xs$, where

$$\begin{aligned} &qBody :: \mathsf{MStateNondet}\ (\mathsf{Int}, [\mathsf{Int}], [\mathsf{Int}])\ m \Rightarrow [\mathsf{Int}] \to m\ [\mathsf{Int}] \\ &qBody\ [\,] = return\ [\,] \\ &qBody\ xs = select\ xs \gg\!= \lambda(x, ys) \to \\ &\qquad\qquad get \gg\!= \lambda st \to guard\ (ok\ (st \oplus x)) \gg \\ &\qquad\qquad put\ (st \oplus x) \gg ((x:) \mathbin{\langle\$\rangle} qBody\ ys)\ \ . \end{aligned}$$

The proof also heavily relies on the commutativity between non-determinism and state.

To wrap up, having fused *perm* and safety checking into one phase, we may compute *queens* by:

$$\begin{aligned} &queens :: \mathsf{MStateNondet}\ (\mathsf{Int}, [\mathsf{Int}], [\mathsf{Int}])\ m \Rightarrow \mathsf{Int} \to m\ [\mathsf{Int}] \\ &queens\ n = protect\ (put\ (0, [\,], [\,]) \gg qBody\ [0\,..\,n-1])\ \ . \end{aligned}$$

This is a backtracking algorithm that attempts to place queens column-by-column, proceeds to the next column if *ok* holds, and backtracks otherwise. The derivation from the specification to this program relies on a number of properties that hold in the local state semantics.

3.2 Transforming between Local State and 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: When each new state is only slightly different from the previous (say, the array is updated in one place each time), we have a wasteful duplication of information. Although this is not expected to be an issue for realistic sizes of the n-queens problem due to the relatively small state, one can imagine that for some problems where the state is very large, this can be a problem.

Global state semantics, on the other hand, has a more "low-level" feel to it. Because a single state is threaded through the entire computation without making any implicit copies, it is easier to reason about resource usage in this setting. So we might write our programs directly in the global state style. However, if we do this to a program that would be more naturally expressed in the local state style (such as our n-queens example), this will come at a great loss of clarity. Furthermore, as we shall see, although it is easier to reason about resource usage of programs in the global state setting, it is significantly more difficult to reason about their semantics. We could also write our program first in a local state style and then translate it to global state style. Doing this manually is a tedious and error-prone process that leaves us with code that is hard to maintain. A more attractive proposition is to design a systematic program transformation that takes a program written for local state semantics as input, and outputs a program that, when interpreted under global state semantics, behaves exactly the same as the original program interpreted under local state semantics.

In the remainder of the paper we define this program transformation and prove it correct. We believe that, in particular, the proof *technique* is of interest.

4 Non-Determinism with Global State

So far, we have evaded giving a precise axiomatic characterisation of global state semantics: although in Section 2 we provided an example implementation that matches our intuition of global state semantics, we haven't provided a clear formulation of that intuition. We begin this section by finally stating the "global state law", which characterises exactly the property that sets apart non-backtrackable state from backtrackable state.

In the rest of the section, we appeal to intuition and see what happens when we work with a global state monad: what pitfalls we may encounter, and what programming pattern we may use, to motivate the more formal treatment in Section 5.

4.1 The Global State Law

We have already discussed general laws for nondeterministic monads (laws (4) through (7)), as well as laws which govern the interaction between state and

nondeterminism in a local state setting (laws (13) and (12)). For global state semantics, an alternative law is required to govern the interactions between nondeterminism and state. We call this the *global state law*.

$$\textbf{put-or}: \quad (put\ s \gg m) \,[\!]\, n = put\ s \gg (m \,[\!]\, n) \quad , \tag{15}$$

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 (12) and (5), the left-hand side of the equation is equal to n, whereas by (5), the right-hand side of the equation is equal to $put\ s \gg n$.

By itself, this law leaves us free to choose from a large space of implementations with different properties. For example, in any given implementation, the programs $return\ x \,[\!]\, return\ y$ and $return\ y \,[\!]\, return\ x$ may be considered semantically identical, or they may be considered semantically distinct. The same goes for the programs $return\ x \,[\!]\, return\ x$ and $return\ x$, or the programs $(put\ s \gg return\ x) \,[\!]\, m$ and $(put\ s \gg return\ x) \,[\!]\, (put\ s \gg m)$. Additional axioms will be introduced as needed to cover these properties in Section 5.2.

4.2 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 (regardless of whether a solution is found). To implement such pattern as a monadic program, one might come up with something like the code below:

$$modify\ next \gg search \gg\!= modReturn\ prev \quad ,$$

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

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

Let the initial state be *st* and assume that *search* found three choices $m_1 \,[\!]\, m_2 \,[\!]\, m_3$. We wish that m_1, m_2, and m_3 all start running with state $next\ st$, and the state is restored to $prev\ (next\ st) = st$ afterwards. Due to (6), however, it expands to

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

With a global state, it means that m_2 starts with state *st*, after which the state is rolled back further 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 and reduces to $\emptyset$, *modReturn prev* is not run at all, due to (7).

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

$$\begin{aligned}&\mathit{side} :: \mathsf{MNondet}\ m \Rightarrow m\ a \to m\ b\\ &\mathit{side}\ m = m \gg \emptyset \quad .\end{aligned}$$

Since non-deterministic branches are executed sequentially, the program

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

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

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 (6) 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.

4.3 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 variation of *put*, which restores the original state when it is backtracked over:

$$\begin{aligned}&\mathit{put}_{\mathsf{R}} :: \mathsf{MStateNondet}\ s\ m \Rightarrow s \to m\ ()\\ &\mathit{put}_{\mathsf{R}}\ s = \mathit{get} \gg\!= \lambda s_0 \to \mathit{put}\ s \mathbin{[\!]} \mathit{side}\ (\mathit{put}\ s_0) \quad .\end{aligned}$$

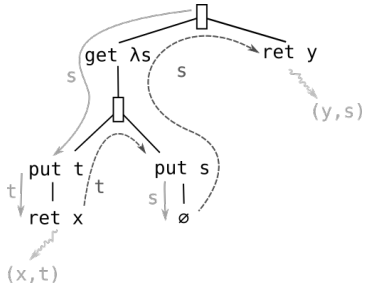

Fig. 2: Illustration of state-restoring put

To help build understanding for put_{R}, Figure 2 shows the flow of execution for the expression $(put_{\mathsf{R}}\ t \gg ret\ x) \,[]\, ret\ y$. 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, because we found a result, we backtrack (since we're using global-state semantics, the state modification caused by $put\ t$ is not reversed), arriving in the *side* operation branch. The $put\ s$ operation is executed, which resets the state to s, and then the branch immediately fails, so we backtrack to the right branch of the topmost ($[]$). There the value y is output with working state s.

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

$$
\begin{aligned}
& put_{\mathsf{R}}\ s \gg comp \\
= {} & (get \gg\!= \lambda s_0 \to put\ s \,[]\, side\ (put\ s_0)) \gg comp \\
= {} & \quad \{\ \text{monad law, left-distributivity (6)}\ \} \\
& get \gg\!= \lambda s_0 \to (put\ s \gg comp) \,[]\, (side\ (put\ s_0) \gg comp) \\
= {} & \quad \{\ \text{by (7) } \emptyset \gg comp = \emptyset\text{, monad laws}\ \} \\
& get \gg\!= \lambda s_0 \to (put\ s \gg comp) \,[]\, side\ (put\ s_0) \quad .
\end{aligned}
$$

Thanks to left-distributivity (6), $(\gg comp)$ is promoted into ($[]$). Furthermore, the $(\gg comp)$ after $side\ (put\ s_0)$ is discarded by (7). 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} is rather tricky. It is instructive comparing

(a) $return\ x$,
(b) $put\ s \gg return\ x$,
(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 (8) through (13) — 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** (8) and **put-get** (10), but **get-put** (9) fails — $get \gg\!= put_{\mathsf{R}}$ and $return\ ()$ can be differentiated by some contexts, for example $(\gg put\ t)$. To see that, we calculate:

$$
\begin{aligned}
& (get \gg\!= put_{\mathsf{R}}) \gg put\ t \\
= {} & (get \gg\!= \lambda s \to get \gg\!= \lambda s_0 \to put\ s \,[]\, side\ (put\ s_0)) \gg put\ t \\
= {} & \quad \{\ \textbf{get-get}\ \} \\
& (get \gg\!= \lambda s \to put\ s \,[]\, side\ (put\ s)) \gg put\ t \\
= {} & \quad \{\ \text{monad laws, left-distributivity}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{get} \gg\!= \lambda s \to (\mathit{put}\ s \gg \mathit{put}\ t) \,[]\, \mathit{side}\ (\mathit{put}\ s) \\
= \quad & \{\ \textbf{put-put}\ \} \\
& \mathit{get} \gg\!= \lambda s \to \mathit{put}\ t \,[]\, \mathit{side}\ (\mathit{put}\ s) \quad .
\end{aligned}
$$

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

In a global-state setting, the left-distributivity law (6) makes it tricky to reason about combinations of $([])$ and $(\gg\!=)$ operators. Suppose we have a program $(m \,[]\, n)$, and we construct an extended program by binding a continuation f to it such that we get $(m \,[]\, n) \gg\!= f$ (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 (6), 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 $m \gg\!= f$) compared to 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 put_R—problematic contexts such as $\mathit{put}\ t$ above are thus ruled out. However, that "all *put* are replaced by put_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 5. Notice though, that simulation of local state semantics by judicious use of put_R does not avoid the unnecessary copying mentioned in Section 3.2, it merely makes it explicit in the program. We will address this shortcoming in Section 5.6.

5 Laws and Translation for Global State Monad

In this section we give a more formal treatment of the non-deterministic global state monad. Not every implementation of the global state law allows us to accurately simulate local state semantics though, so we propose additional laws that the implementation must respect. These laws turn out to be rather intricate. To make sure that there exists a model, an implementation is proposed, 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 put_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.

5.1 Programs and Contexts

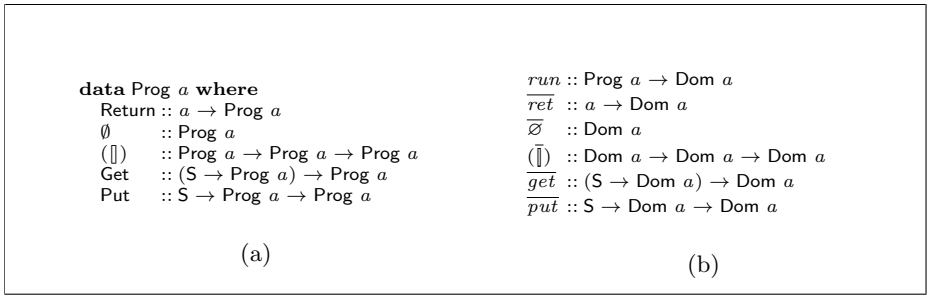

Fig. 3: (a) Syntax for programs. (b) Semantic domain.

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 3(a) defines a syntax for nondeterministic, stateful, closed programs Prog, where the Get and Put constructors take continuations as arguments, and the $(\gg\!=)$ operator is defined as follows:

$$
\begin{aligned}
&(\gg\!=) :: \mathsf{Prog}\ a \to (a \to \mathsf{Prog}\ b) \to \mathsf{Prog}\ b\\
&\mathsf{Return}\ x \gg\!= f = f\ x\\
&\emptyset \gg\!= f = \emptyset\\
&(m \,[]\, n) \gg\!= f = (m \gg\!= f) \,[]\, (n \gg\!= f)\\
&\mathsf{Get}\ k \gg\!= f = \mathsf{Get}\ (\lambda s \to k\ s \gg\!= f)\\
&\mathsf{Put}\ s\ m \gg\!= f = \mathsf{Put}\ s\ (m \gg\!= k)\ .
\end{aligned}
$$

One can see that $(\gg\!=)$ is defined as a purely syntactical manipulation, and its definition has laws (6) and (7) 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 $\overline{ret}$, $\overline{\varnothing}$, $\overline{get}$, $\overline{put}$ and $(\bar{[]})$ (see figure 3(b)). The $run :: \mathsf{Prog}\ a \to \mathsf{Dom}\ a$ function "runs" a program Prog a into a value in the semantic domain Dom a:

$$
\begin{aligned}
&run\ (\mathsf{Return}\ x) = \overline{ret}\ x\\
&run\ \emptyset = \overline{\varnothing}\\
&run\ (m_1 \,[]\, m_2) = run\ m_1\ \bar{[]}\ run\ m_2\\
&run\ (\mathsf{Get}\ k) = \overline{get}\ (\lambda s \to run\ (k\ s))\\
&run\ (\mathsf{Put}\ s\ m) = \overline{put}\ s\ (run\ m)\ .
\end{aligned}
$$

Note that no $\overline{\gg\!=}$ 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*.

5.2 Laws for 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 $((\gg\!=) :: \mathsf{Dom}\ a \to (a \to \mathsf{Dom}\ b) \to \mathsf{Dom}\ b)$ 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 ((8) through (11)) must be reformulated to fit the continuation-passing style of the semantic domain operators.

$$\overline{put}\ s\ (\overline{put}\ t\ p) = \overline{put}\ t\ p \quad , \tag{16}$$

$$\overline{put}\ s\ (\overline{get}\ k) = \overline{put}\ s\ (k\ s) \quad , \tag{17}$$

$$\overline{get}\ (\lambda s \to \overline{put}\ s\ m) = m \quad , \tag{18}$$

$$\overline{get}\ (\lambda s \to \overline{get}\ (\lambda t \to k\ s\ t)) = \overline{get}\ (\lambda s \to k\ s\ s) \quad . \tag{19}$$

Two of the nondeterminism laws—(6) and (7)—also mention the bind operator. As we have seen earlier, they are trivially implied by the definition of $(\gg\!=)$ for Prog. Therefore, we need not impose equivalent laws for the semantic domain (and in fact, we cannot formulate them given the representation we have chosen). Only the two remaining nondeterminism laws—(4) and (5)—need to be stated:

$$(m \mathbin{\overline{[\!]}} n) \mathbin{\overline{[\!]}} p = m \mathbin{\overline{[\!]}} (n \mathbin{\overline{[\!]}} p) \quad , \tag{20}$$

$$\overline{\varnothing} \mathbin{\overline{[\!]}} m = m \mathbin{\overline{[\!]}} \overline{\varnothing} = m \quad . \tag{21}$$

We also reformulate the global-state law (15):

$$\overline{put}\ s\ p \mathbin{\overline{[\!]}} q = \overline{put}\ s\ (p \mathbin{\overline{[\!]}} q) \quad . \tag{22}$$

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

$$\begin{aligned} &\overline{get}\ (\lambda s \to \overline{put}\ (t\ s)\ p \mathbin{\overline{[\!]}} \overline{put}\ (u\ s)\ q \mathbin{\overline{[\!]}} \overline{put}\ s\ \overline{\varnothing}) = \\ &\overline{get}\ (\lambda s \to \overline{put}\ (u\ s)\ q \mathbin{\overline{[\!]}} \overline{put}\ (t\ s)\ p \mathbin{\overline{[\!]}} \overline{put}\ s\ \overline{\varnothing}) \quad , \end{aligned} \tag{23}$$

$$\overline{put}\ s\ (\overline{ret}\ x \mathbin{\overline{[\!]}} p) = \overline{put}\ s\ (\overline{ret}\ x) \mathbin{\overline{[\!]}} \overline{put}\ s\ p \quad . \tag{24}$$

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 $(\overline{[\!]})$ 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. This might seem contradictory at first glance. To be more precise, we do not require the property $p \mathbin{\overline{[\!]}} q = q \mathbin{\overline{[\!]}} p$ because the subprograms p and q might perform non-commuting edits to the global state. But we do expect that programs without side-effects commute freely; for instance, we expect $return\ x \mathbin{\overline{[\!]}} return\ y = return\ y \mathbin{\overline{[\!]}} return\ x$. In other words, collecting all the

results of a nondeterministic computation is done with a set-based semantics in mind rather than a list-based semantics, but this does not imply that the order of state effects does not matter.

In fact, the notion of commutativity we wish to impose is still somewhat stronger than just the fact that results commute: we want the $(\bar{[\![}])$ operator to commute with respect to any pair of subprograms whose modifications of the state are ignored—that is, immediately overwritten—by the surrounding program. This property is expressed by law (23). An example of an implementation which does not respect this law is one that records the history of state changes.

In global-state semantics, $\overline{put}$ operations cannot, in general, distribute over $(\bar{[\![}])$. However, an implementation may permit distributivity if certain conditions are met. Law (24) states that a $\overline{put}$ operation distributes over a nondeterministic choice if the left branch of that choice simply returns a value. This law has a particularly striking implication: it disqualifies any implementation for which a bind operator $(\mathbin{\overline{\gg=}})::\mathsf{Dom}\ a \to (a \to \mathsf{Dom}\ b) \to \mathsf{Dom}\ b$ can be defined! Consider for instance the following program:

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

If (24) holds, this program should be equal to

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

However, it is proved in Figure 4 that the first program can be reduced to $\overline{put}\ y\ (\overline{ret}\ w\ \bar{[\![}\ \overline{ret}\ y)$, whereas the second program is equal to $\overline{put}\ y\ (\overline{ret}\ w\ \bar{[\![}\ \overline{ret}\ x)$, which clearly does not always have the same result.

To gain some intuition about why law (24) prohibits a bind operator, consider that the presence or absence of a bind operator influences what equality of programs means. Our first intuition might be that we consider two programs equal if they produce the same outputs given the same inputs. But this is too narrow a view: for two programs to be considered equal, they must also behave the same under *composition*; that is, we must be able to replace one for the other within a larger whole, without changing the meaning of the whole. The bind operator allows us to compose programs *sequentially*, and therefore its existence implies that, for two programs to be considered equal, they must also behave identically under sequential composition. Under local-state semantics, this additional requirement coincides with other notions of equality: we can't come up with a pair of programs which both produce the same outputs given the same inputs, but behave differently under sequential composition. But under global-state semantics, we can come up with such counterexamples: consider the subprograms of our previous example $\overline{put}\ x\ (\overline{ret}\ w\ \bar{[\![}\ \overline{get}\ \overline{ret})$ and $(\overline{put}\ x\ (\overline{ret}\ w)\ \bar{[\![}\ \overline{put}\ x\ (\overline{get}\ \overline{ret}))$. Clearly we expect these two programs to produce the exact same results in isolation, yet when they are sequentially composed with the program $\lambda z \to \overline{put}\ y\ (\overline{ret}\ z)$, their different nature is revealed (by (6)).

It is worth remarking that introducing either one of these additional laws disqualify the example implementation given in Section 2.2 (even if it is adapted

$$
\begin{array}{l}
\overline{put}\ x\ (\overline{ret}\ w \mathbin{\overline{[]}} \overline{get}\ \overline{ret}) \gg\!= \lambda z \to \overline{put}\ y\ (\overline{ret}\ z) \\
\quad = \quad \{\ \text{definition of } (\ \gg\!=\ )\ \} \\
\overline{put}\ x\ (\overline{put}\ y\ (\overline{ret}\ w) \mathbin{\overline{[]}} \overline{get}\ (\lambda s \to \overline{put}\ y\ (\overline{ret}\ s))) \\
\quad = \quad \{\ \text{by (22) and (16)}\ \} \\
\overline{put}\ y\ (\overline{ret}\ w \mathbin{\overline{[]}} \overline{get}\ (\lambda s \to \overline{put}\ y\ (\overline{ret}\ s))) \\
\quad = \quad \{\ \text{by (24)}\ \} \\
\overline{put}\ y\ (\overline{ret}\ w) \mathbin{\overline{[]}} \overline{put}\ y\ (\overline{get}\ (\lambda s \to \overline{put}\ y\ (\overline{ret}\ s))) \\
\quad = \quad \{\ \text{by (17) and (16)}\ \} \\
\overline{put}\ y\ (\overline{ret}\ w) \mathbin{\overline{[]}} \overline{put}\ y\ (\overline{ret}\ y) \\
\quad = \quad \{\ \text{by (24)}\ \} \\
\overline{put}\ y\ (\overline{ret}\ w \mathbin{\overline{[]}} \overline{ret}\ y)
\end{array}
$$

(a)

$$
\begin{array}{l}
(\overline{put}\ x\ (\overline{ret}\ w) \mathbin{\overline{[]}} \overline{put}\ x\ (\overline{get}\ \overline{ret})) \\
\quad \gg\!= \lambda z \to \overline{put}\ y\ (\overline{ret}\ z) \\
\quad = \quad \{\ \text{definition of } (\ \gg\!=\ )\ \} \\
\overline{put}\ x\ (\overline{put}\ y\ (\overline{ret}\ w)) \\
\quad \mathbin{\overline{[]}} \overline{put}\ x\ (\overline{get}\ (\lambda s \to \overline{put}\ y\ (\overline{ret}\ s))) \\
\quad = \quad \{\ \text{by (16) and (17)}\ \} \\
\overline{put}\ y\ (\overline{ret}\ w) \mathbin{\overline{[]}} \overline{put}\ x\ (\overline{put}\ y\ (\overline{ret}\ x)) \\
\quad = \quad \{\ \text{by (16)}\ \} \\
\overline{put}\ y\ (\overline{ret}\ w) \mathbin{\overline{[]}} \overline{put}\ y\ (\overline{ret}\ x) \\
\quad = \quad \{\ \text{by (24)}\ \} \\
\overline{put}\ y\ (\overline{ret}\ w \mathbin{\overline{[]}} \overline{ret}\ x)
\end{array}
$$

(b)

Fig. 4: Proof that law (24) implies that a bind operator cannot exist for the semantic domain.

for the continuation-passing style of these laws). As the given implementation records the order in which results are yielded by the computation, law (23) cannot be satisfied. And the example implementation also forms a monad, which means it is incompatible with law (24).

Machine-Verified Proofs From this point forward, we provide proofs mechanized in Coq for many theorems. When we do, we mark the proven statement with a check mark (✓).

5.3 An Implementation of the Semantic Domain

We present an implementation of Dom that satisfies the laws of section 5.2, and we provide machine-verified proofs to that effect. In the following implementation, we let Dom be the union of M s a for all a and for a given s.

The implementation is based on a multiset or Bag data structure. In the mechanization, we implement Bag a as a function $a \to$ Nat.

type Bag a
singleton :: $a \to$ Bag a
emptyBag :: Bag a
sum :: Bag $a \to$ Bag $a \to$ Bag a

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. The use of an unordered data structure to return the results of the computation is needed to comply with law (23).

In Section 5.2 we mentioned that, as a consequence of law (24) we must design the implementation of our semantic domain in such a way that it is impossible to define a bind operator $\overline{\gg=} :: \mathsf{Dom}\ a \to (a \to \mathsf{Dom}\ b) \to \mathsf{Dom}\ b$ for it. This is the case for our implementation: we only retain the final result of the branch without any information on how to continue the branch, which makes it impossible to define the bind operator.

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

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

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

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

$$\begin{aligned}
&\overline{get} :: (s \to \mathsf{M}\ s\ a) \to \mathsf{M}\ s\ a \\
&\overline{get}\ k = \lambda s \to k\ s\ s \\
&\overline{put} :: s \to \mathsf{M}\ s\ a \to \mathsf{M}\ s\ a \\
&\overline{put}\ s\ k = \lambda_ \to k\ s
\end{aligned}$$

The $\bar{[\!]}$ 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.

$$\begin{aligned}
&(\bar{[\!]}) :: \mathsf{M}\ s\ a \to \mathsf{M}\ s\ a \to \mathsf{M}\ s\ a \\
&(xs\ \bar{[\!]}\ ys)\ s = \textbf{let}\ (ansx, s') = xs\ s \\
&\qquad\qquad\qquad\qquad\ \ (ansy, s'') = ys\ s' \\
&\qquad\qquad\qquad\ \ \textbf{in}\ (sum\ ansx\ ansy, s'')
\end{aligned}$$

Lemma 1. *This implementation conforms to every law introduced in Section 5.2.* ✓

5.4 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 $\mathsf{Prog}\ a$ describes a syntactical structure without ascribing meaning to it. For example, one cannot simply assert that $\mathsf{Put}\ x\ (\mathsf{Put}\ y\ p)$ is *equal*

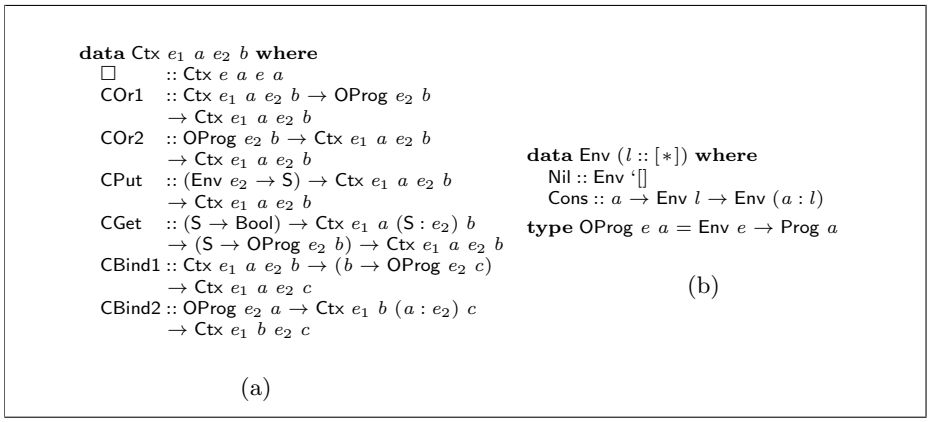

Fig. 5: (a) Environments and open programs. (b) Syntax for contexts.

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 "semantic 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 semantic equivalence in itself is not sufficient; we require a notion of "contextual semantic equivalence" of programs which allows us to formulate properties about semantic equivalence which hold in any surrounding context. Figure 5(a) 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 5(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 5(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 Prog a, given an environment of type Env e. Environments, in turn, are defined as heterogeneous lists. We also define a function for mapping open programs onto the semantic domain.

$$orun :: \mathsf{OProg}\ e\ a \to \mathsf{Env}\ e \to \mathsf{Dom}\ a$$
$$orun\ p\ env = run\ (p\ env)\ .$$

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:

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

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 (16) as

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

Proofs for the state laws, the nondeterminism laws and the *put-or* law then easily follow from the analogous semantic domain laws.

More care is required when we want to adapt law (24) into the Prog setting. At the end of Section 5.2, we saw that this law precludes the existence of a bind operator in the semantic domain. Since a bind operator for Progs exists, it might seem we're out of luck when we want to adapt law (24) to Progs. But because Progs are merely syntax, we have much more fine-grained control over what equality of programs means. When talking about the semantic domain, we only had one notion of equality: two programs are equal only when one can be substituted for the other in any context. So if, in that setting, we want to introduce a law that does not hold under a particular composition (in this case, sequential composition), the only way we can express that is to say that that composition is impossible in the domain. But the fact that Prog is defined purely syntactically opens up the possibility of defining multiple notions of equality which may exist at the same time. In fact, we have already introduced syntactic equality, semantic equality, and contextual equality. It is precisely this choice of granularity that allows us to introduce laws which only hold in programs of a certain form (non-contextual laws), while other laws are much more general (contextual laws). A direct adaptation of law (24) would look something like this:

$$\begin{aligned}\forall \mathsf{C}.\ \mathsf{C}\text{ is bind-free} \Rightarrow &\mathit{orun}\ (\mathsf{C}[\mathsf{Put}\ s\ (\mathsf{Return}\ x \mathbin{[\!]} p)]) \\ &= \mathit{orun}\ (\mathsf{C}[\mathsf{Put}\ s\ (\mathsf{Return}\ x) \mathbin{[\!]} \mathsf{Put}\ s\ p])\quad .\end{aligned}$$

In other words, the two sides of the equation can be substituted for one another, but only in contexts which do not contain any binds. However, in our mechanization, we only prove a more restricted version where the context must be empty (in other words, what we called semantic equivalence), which turns out to be enough for our purposes.

$$\mathit{run}\ (\mathsf{Put}\ s\ (\mathsf{Return}\ x \mathbin{[\!]} p)) = \mathit{run}\ (\mathsf{Put}\ s\ (\mathsf{Return}\ x) \mathbin{[\!]} \mathsf{Put}\ s\ p)\quad .\checkmark \tag{25}$$

5.5 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 4.3. This transformation is implemented by the function *trans* for closed programs, and *otrans* for open programs:

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

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

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

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":

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

For example, we formulate the statement that the *put-put* law (16) holds for our monad as interpreted by *eval* as

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

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.

$$\mathit{run}\ (\mathsf{Put}\ x\ (\mathit{trans}\ m_1 \,[]\, m_2)) = \mathit{run}\ (\mathsf{Put}\ x\ (\mathit{trans}\ m_1) \,[]\, \mathsf{Put}\ x\ m_2) \quad .\checkmark \tag{26}$$

Proof of this lemma depends on law (24).

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 (13) and (12)

$$m \gg \emptyset =_{\mathtt{LS}} \emptyset \quad ,\checkmark \tag{27}$$

$$m \gg\!= (\lambda x \to f_1\ x \,[]\, f_2\ x) =_{\mathtt{LS}} (m \gg\!= f_1) \,[]\, (m \gg\!= f_2) \quad .\checkmark \tag{28}$$

We provide machine-verified proofs for these theorems. The proof for (27) follows by straightforward induction. The inductive proof (with induction on m) of law (28) requires some additional lemmas.

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

$$run\ (trans\ p\ []\ trans\ q) = run\ (trans\ q\ []\ trans\ p)\ \ .\checkmark \tag{29}$$

The proof of this property motivated the introduction of law (23).

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.

$$run\ (\mathsf{Get}\ (\lambda s \to trans\ (m_1\ s)\ []\ (m_2\ s))) = \tag{30}$$

$$run\ (\mathsf{Get}\ (\lambda s \to trans\ (m_1\ s))\ []\ \mathsf{Get}\ m_2)\ \ .\checkmark \tag{31}$$

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

$$run\ (trans\ (trans\ p)) = run\ (trans\ p)\ \ .\checkmark \tag{32}$$

5.6 Backtracking with a Global State Monad

Although we can now interpret a local state program through translation to a global state program, we have not quite yet delivered on our promise to address the space usage issue of local state semantics. From the definition of put_{R} it is clear that we simply make the implicit copying of the local state semantics explicit in the global state semantics. As mentioned in Section 4.2, 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:

$$modify_{\mathsf{R}} :: \mathsf{MStateNondet}\ s\ m \Rightarrow (s \to s) \to (s \to s) \to m\ ()$$
$$modify_{\mathsf{R}}\ next\ prev = modify\ next\ []\ side\ (modify\ prev)\ \ .$$

Unlike put_{R}, $modify_{\mathsf{R}}$ does not keep any copies of the old state alive, as it does not introduce a branching point where the right branch refers to a variable introduced outside the branching point. 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 $modify_{\mathsf{R}}\ next\ prev \gg\! trans\ m$? We explore this question by extending our Prog syntax with an additional $\mathsf{Modify}_{\mathsf{R}}$ construct, thus obtaining a new $\mathsf{Prog}_{\mathsf{M}}$ 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 $\mathsf{Modify}_{\mathsf{R}}\ next\ prev\ p$ in a $\mathsf{Prog}_{\mathsf{M}}\ a$ program.

We then define two translation functions from $\mathsf{Prog}_{\mathsf{M}}\ a$ to $\mathsf{Prog}\ a$, which both replace Puts with put_{R}s along the way, like the regular *trans* function. The first replaces each $\mathsf{Modify}_{\mathsf{R}}$ in the program by a direct analogue of the definition given above, while the second replaces it by $\mathsf{Get}\ (\lambda s \to \mathsf{Put}\ (next\ s)\ (trans_2\ p))$:

$$
\begin{aligned}
&trans_1 :: \mathsf{Prog}_{\mathsf{M}}\ a \to \mathsf{Prog}\ a\\
&\ldots\\
&trans_1\ (\mathsf{Modify}_{\mathsf{R}}\ next\ prev\ p) = \mathsf{Get}\ (\lambda s \to \mathsf{Put}\ (next\ s)\ (trans_1\ p)\\
&\qquad\qquad\qquad\qquad\qquad\qquad [] \ \mathsf{Get}\ (\lambda t \to \mathsf{Put}\ (prev\ t)\ \emptyset))\\
&trans_2 :: \mathsf{Prog}_{\mathsf{M}}\ a \to \mathsf{Prog}\ a\\
&\ldots\\
&trans_2\ (\mathsf{Modify}_{\mathsf{R}}\ next\ prev\ p) = \mathsf{Get}\ (\lambda s \to put_{\mathsf{R}}\ (next\ s)\ (trans_2\ p))\\
&\quad \textbf{where}\ put_{\mathsf{R}}\ s\ p = \mathsf{Get}\ (\lambda t \to \mathsf{Put}\ s\ p\ [] \ \mathsf{Put}\ t\ \emptyset)
\end{aligned}
$$

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

We then prove that these two transformations lead to semantically identical instances of $\mathsf{Prog}\ a$.

Lemma 2. $run\ (trans_1\ p) = run\ (trans_2\ p)$. ✓

This means that, if we make some effort to rewrite parts of our program to use the $\mathsf{Modify}_{\mathsf{R}}$ construct rather than Put, we can use the more efficient translation scheme $trans_1$ to avoid introducing unnecessary copies.

n-Queens using a global state To wrap up, we revisit the n-queens puzzle. Recall that, for the puzzle, the operator that alters the state (to check whether a chess placement is safe) is defined by

$$(i, us, ds) \oplus x = (1 + i, (i + x) : us, (i - x) : ds)\ \ .$$

By defining $(i, us, ds) \ominus x = (i - 1, tail\ us, tail\ ds)$, we have $(\ominus x) \cdot (\oplus x) = id$. One may thus compute all solutions to the puzzle, in a scenario with a shared global state, by $run\ (queens_{\mathsf{R}}\ n)$, where

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

6 Related Work

6.1 Prolog

Prolog is a prominent example of a system that exposes nondeterminism with local state to the user, but is itself implemented in terms of a single global state.

Warren Abstract Machine The folklore idea of undoing modifications upon backtracking is a key feature of many Prolog implementations, in particular those based on the Warren Abstract Machine (WAM) [1]. The WAM's global state is the program heap and Prolog programs modify this heap during unification only in a very specific manner: following the union-find algorithm, they overwrite cells that contain self-references with pointers to other cells. Undoing these modifications only requires knowledge of the modified cell's address, which can be written back in that cell during backtracking. The WAM has a special stack, called the trail stack, for storing theses addresses, and the process of restoring those cells is called *untrailing*.

The 4-Port Box Model While trailing happens under the hood, there is a folklore Prolog programming pattern for observing and intervening at different points in the control flow of a procedure call, known as the *4-port box model*. 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:

$$
\begin{aligned}
&(\mathit{putStr}\ \texttt{"call"} \mathbin{[\!]} \mathit{side}\ (\mathit{putStr}\ \texttt{"fail"})) \gg \\
&p \gg\!= \lambda x \to \\
&(\mathit{putStr}\ \texttt{"exit"} \mathbin{[\!]} \mathit{side}\ (\mathit{putStr}\ \texttt{"redo"})) \gg \mathit{return}\ x \quad .
\end{aligned}
$$

This technique was applied in the monadic setting by Hinze [6], and it has been our inspiration for expressing the state restoration with global state.

6.2 Reasoning About Side Effects

There are many works on reasoning and modelling side effects. Here we cover those that have most directly inspired this paper.

Axiomatic Reasoning Our work was directly inspired by Gibbons and Hinze's proposal to reason axiomatically about programs with side effects, and their axiomatic characterisation of local state in particular [4]. We have extended their work with an axiomatic characterisation of global state and on handling the former in terms of the latter. We also provide models that satisfy the axioms, whereas their paper mistakenly claims that one model satisfies the local state axioms and that another model is monadic.

Algebraic Effects Our formulation of implementing local state with global state is directly inspired by the effect handlers approach of Plotkin and Pretnar [12]. By making the free monad explicit our proofs benefit directly from the induction principle that Bauer and Pretnar establish for effect handler programs [2].

While Lawvere theories were originally Plotkin's inspiration for studying algebraic effects, the effect handlers community has for a long time paid little attention to them. Yet, recently Lukšič and Pretnar [10] have investigated a framework for encoding axioms (or "effect theories") 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. This type system would allow us to express at the type-level that our handler interprets local state in terms of global state.

7 Conclusions

Starting from Gibbons and Hinze's observation [4] that the interaction between state and nondeterminism can be characterized axiomatically in multiple ways, we explored the differences between local state semantics (as characterised by Gibbons and Hinze) and global state semantics (for which we gave our own non-monadic characterisation).

In global state semantics, we find that we may use $(\![])$ to simulate sequencing, and that the idea can be elegantly packaged into commands like put_{R} and $modify_{\mathsf{R}}$. 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 *global state law*, plus some additional laws, which the semantics should satisfy. We verified (with the help of the Coq proof assistant) that there is an implementation satisfying these laws.

Using the n-queens puzzle as an example, we showed that one can end up in a situation where a problem is naturally expressed with local state semantics, but the greater degree of control over resources that global state semantics offers is desired. We then describe a technique to systematically transform a monadic program written against the local state laws into one that, when interpreted under global state laws, produces the same results as the original program. This transformation can be viewed as a handler (in the algebraic effects sense): it implements the interface of one effect in terms of the interface of another. We also verified the correctness of this transformation in Coq.

Acknowledgements We would like to thank Matija Pretnar, the members of IFIP WG 2.1, the participants of Shonan meeting 146 and the MPC reviewers for their insightful comments. We would also like to thank the Flemish Fund for Scientific Research (FWO) for their financial support.

References

1. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction (01 1991)
2. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. Logical Methods in Computer Science **10**(4) (2014), https://doi.org/10.2168/LMCS-10(4:9)2014
3. Gale, Y.: ListT done right alternative. https://wiki.haskell.org/ListT_done_right_alternative (2007)
4. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Danvy, O. (ed.) International Conference on Functional Programming. pp. 2–14. ACM Press (2011)
5. Gill, A., Kmett, E.: The monad transformer library. https://hackage.haskell.org/package/mtl (2014)
6. Hinze, R.: Prolog's control constructs in a functional setting - axioms and implementation. Int. J. Found. Comput. Sci. **12**(2), 125–170 (2001), https://doi.org/10.1142/S0129054101000436
7. Hutton, G., Fulger, D.: Reasoning about effects: seeing the wood through the trees. In: Symposium on Trends in Functional Programming (2007)
8. Kiselyov, O.: Laws of monadplus. http://okmij.org/ftp/Computation/monads.html#monadplus (2015)
9. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: Reppy, J.H. (ed.) Symposium on Haskell. pp. 94–105. ACM Press (2015)
10. Lukšič, v., Pretnar, M.: Local algebraic effect theories (2019), submitted
11. Moggi, E.: Computational lambda-calculus and monads. In: Parikh, R. (ed.) Logic in Computer Science. pp. 14–23. IEEE Computer Society Press (1989)
12. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: Castagna, G. (ed.) European Symposium on Programming. pp. 80–94. No. 5502 in Lecture Notes in Computer Science (2009)
13. Volkov, N.: The list-t package. http://hackage.haskell.org/package/list-t (2014)
14. Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Program Design Calculi: Marktoberdorf Summer School. pp. 233–264. Springer-Verlag (1992)
15. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Voigtländer, J. (ed.) Symposium on Haskell. pp. 1–12. ACM Press (2012)